Deterministic Replay for MCAPI Programs

Mohamed Elwakil
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008
mohamed.elwakil@wmich.edu

Zijiang Yang
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008
zijiang.yang@wmich.edu

ABSTRACT
The Multicore Communications API (MCAPI) is a new message passing API that was released by the Multicore Association. MCAPI provides an interface designed for closely distributed embedded systems with multiple cores on a chip and/or chips on a board. Similar to concurrent programs in other domains, debugging MCAPI programs is a challenging task due to their non-deterministic behavior. In this paper we present a tool that is able to deterministically replay the executions of MCAPI programs, which provides valuable insight for MCAPI developers in case of failure.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Reliability

Keywords
MCAPI, message race, multicore programs, debugging, deterministic replay

1. INTRODUCTION
The Multicore Association has developed the Multicore Communications API (MCAPI) specification and its reference implementation to address inter-core communication needs in multi-core programs. MCAPI is different from the Message Passing Interface (MPI) [1], a widely used standard for managing coarse-grained concurrency on distributed computers. While MPI is intended for inter-computer communication and needs to be installed on top of an operating system, MCAPI is intended for inter-core communication and can be installed on top of an operating system or an extremely thin run-time environment such as a hypervisor. A major design goal of MCAPI is to function as a low-latency interface benefiting from efficient on-chip interconnects in a multi-core chip. Thus, MCAPI is a light weight API that delivers high performance and needs a tiny memory footprint that is significantly lower than that of MPI [2].

In the traditional approach to debugging, a program is repeatedly executed under the control of a debugger to allow the user to obtain more information about the program states and intermediate results. This debugging approach is not viable for MCAPI programs as they suffer from the irreproducibility effect [3] due to their intrinsic non-determinism. The fact that two subsequent runs under the same input are not guaranteed to behave the same makes debugging MCAPI programs a challenging task. Such challenges facing MCAPI developers, however, are certainly not new. Similar to MCAPI, MPI programs exhibit non-deterministic behaviors. A common technique for aiding debugging MPI programs is using record and replay tools. During the recording phase, a program execution is monitored to record information about the execution. If an execution fails, the recorded data is retrieved to replay the program such that the behavior of the program during the replay phase is equivalent to the behavior observed in the recording phase.

MPI record and replay tools typically fall into two categories: data-replay [4] and order-replay [5]. During the recording phase, data-replay tools record the contents of all received messages at all processes in a trace. During the replay phase, some processes are run while others are simulated. The messages sent by the simulated processes originate from the trace file. Data-replay allows replaying one process, rather than all processes of a program, at a cost of large trace files. On the other hand, order-replay tools record the outcomes of non-deterministic operations and enforce these outcomes during replay. In the order-replay approach, all processes must be running during a replay. Since order-replay tools only record the outcomes of non-deterministic operations, it records far less data than data-replay tools.

In this paper, we present DR-MCAPI, which allows Deterministic Replay of MCAPI programs. DR-MCAPI adopts the order-replay approach due to the following two reasons: (1) because of the unpredictable nature of MCAPI programs executions, recording is needed in all executions including successful ones. In comparison, replaying is needed only when errors have been discovered and the developers need to scrutinize the details. Therefore, recording is more sensitive to performance. In fact, the record and replay technique is most useful if it could be used in production runs. This can only happen if the overhead of recording become negligible; (2) for most concurrency bugs it is desirable to replay all relevant processes with corresponding message exchanges in correct total order instead of replaying each process in isolation. In fact, inferring correct total orders has been the focus of recent studies on hardware based execution replay tools for concurrent systems (c.f. [6] and [7]).

Although there exist several record and replay tools for MPI programs, deterministic replay for MCAPI is different due to the different semantics of its communication primitives. For example, there is no notion of message source, but only destination endpoints in MCAPI. Thus unlike MPI applications it is not possible to identify the source of a received message. To our best knowledge, DR-MCAPI is the first such tool for MCAPI programs. The rest of the paper is organized as follows. In section 2, we briefly explain the MCAPI syntax and highlight the sources of non-determinism in a MCAPI program. Then in section 3, we describe the design of DR-MCAPI. The experimental results are presented in section 4, followed by the related works in section 5. Finally, section 6 concludes the paper.

2. NON-DETERMINISM IN MCAPI PROGRAMS

In MCAPI programs, a core is referred to as a *node*. Communication between nodes occurs through *endpoints*. A node may have one or more endpoints and an endpoint is uniquely defined by a node identifier and a port number. The MCAPI specification supplies APIs for initializing nodes, creating and deleting endpoints, obtaining addresses of remote endpoints, and sending and receiving messages. The functions used for sending and receiving messages are: mcapi_msg_send, mcapi_msg_send_i, mcapi_msg_recv, and mcapi_msg_recv_i.

The functions mcapi_msg_send and mcapi_msg_send_i send messages, asynchronously, between two endpoints. While mcapi_msg_send blocks till the message has been copied from a program buffer to the MCAPI runtime buffers, mcapi_msg_send_i is non-blocking and returns immediately before that data has been completely copied. The functions mcapi_msg_recv and mcapi_msg_recv_i retrieve messages from the MCAPI runtime buffers. The function mcapi_msg_recv blocks till a message has been retrieved from the runtime buffer, while mcapi_msg_recv_i is non-blocking and returns immediately even if there are no messages in the buffer. MCAPI receive calls are called *promiscuous* as they permit receiving messages from any source endpoint.

The MCAPI specification provides request variables and the mcapi_wait, mcapi_wait_any and mcapi_test functions to track the status of a non-blocking function call. A non-blocking function takes a request variable as an input and initializes it by setting its value to pending. Calling mcapi_wait with a request variable *R* blocks execution till the non-blocking operation that initialized *R* has completed. Invoking mcapi_wait_any with an array of request variables *Rs* blocks execution till any of the operations in *Rs* has completed. The mcapi_wait_any returns the index of the completed request. The mcapi_test function succeeds (i.e. returns true) if its input request has completed, fails (i.e. returns false) otherwise.

There are two rules that govern the order of messages arrivals at a destination endpoint: 1) Messages sent from the same source endpoint to the same destination endpoint are guaranteed to arrive at their destination according to their transmission order and 2) Messages sent from different source endpoints will arrive at their destination in any order, even if these source endpoints belong to the same node. The second rule combined with the fact that mcapi_msg_recv and mcapi_msg_recv_i calls don't specify the source endpoint, make it possible for *message races* to take place. Two or more messages are said to be racing if their order of arrival at a destination (i.e. an endpoint) is non-deterministic [8].

Figure 1 shows a MCAPI program with message races. A node creates a single endpoint (line 3) and sends messages to all other nodes (lines 4-8) and is expecting to receive a message from all other nodes (lines 9-10). Assuming there are N nodes, any node should receive N-1 messages that are racing with each other. The orders of message arrivals can change across different executions of the program (i.e. the final values in the Buffer array will be different with different executions); leading to the irreproducibility effect.

```
1    N=NodesCount();
2    mcapi_init_node(ThisNode);
3    LocalEP=mcapi_create_ep(ThisNode,1);
4    for (Index=0;Index<N;Index++){
5      if (Index==ThisNode) continue;
6      RemoteEP=mcapi_get_ep(Index,1);
7     mcapi_msg_send(LocalEP,RemoteEP,&Data[Index]);
8    }
9    for (Index=0;Index<N-1; Index++)
10     mcapi_msg_recv(LocalEP,&Buffer[Index]);
11   mcapi_delete_ep(LocalEP);
12   mcapi_finalize_node(ThisNode);
```

Figure 1. A MCAPI program with message races

Another source of non-determinism in MCAPI programs is the mcapi_wait_any call. In Figure 2, a node has two endpoints and is expecting to receive a message at each endpoint (lines 4-5). mcapi_wait_any blocks execution until either one of the two messages is received. Depending on which endpoint receives a message first, the value of ReqIndex may be different across executions of the program, which results in different branches of the switch being selected in different executions.

```
1    mcapi_init_node(ThisNode);
2    LocalEP1=mcapi_create_ep(ThisNode,1);
3    LocalEP2=mcapi_create_ep(ThisNode,2);
4    mcapi_msg_recv_i(LocalEP1,&Buffer1,Requests[0]);
5    mcapi_msg_recv_i(LocalEP2,&Buffer2,Requests[1]);
6    ReqIndex=mcapi_wait_any(Requests);
7    switch (ReqIndex) {
8      case 0: ...
9      case 1: ...
10   }
11   mcapi_delete_ep(LocalEP1);
12   mcapi_delete_ep(LocalEP2);
13   mcapi_finalize_node(ThisNode);
```

Figure 2. A MCAPI program with mcapi_wait_any

Using the non-blocking mcapi_test introduces non-determinism as well. In Figure 3, a node is expecting to receive a message at a local endpoint. The function mcapi_test is used to determine whether the expected message has arrived. The number of times mcapi_test returns false, and consequently the value of variable A at line 5, is dependent on uncontrollable factors such as the current core workload and the inter-core communication latency.

```
mcapi_init_node(ThisNode);
LocalEP=mcapi_create_ep(ThisNode,1);
mcapi_msg_recv_i(LocalEP,&Buffer,Request);
while (!mcapi_test(Request)) A++;
func_call(A);
mcapi_delete_ep(LocalEP);
mcapi_finalize_node(ThisNode);
```

Figure 3. A MCAPI program with mcapi_test

In summary, besides user inputs and random value generations, there are three additional sources of non-determinism in MCAPI programs: 1) mcapi_msg_recv and mcapi_msg_recv_i calls, 2) mcapi_wait_any calls and 3) mcapi_test calls. Such inherent non-deterministic behavior does not permit repeated execution as a reliable mean of debugging MCAPI programs. Hence, introducing the ability to replay an observed MCAPI program execution can significantly help MCAPI programs developers.

3. DR-MCAPI

Figure 4 depicts the workflow of our tool for deterministic replay of MCAPI programs. DR-MCAPI consists of two parts: a source code instrumenter and a MCAPI library wrapper (DR-MCAPI Library). The instrumenter replaces calls to the MCAPI library routines in an input program with calls to the DR-MCAPI library. Figure 5 shows the result of instrumenting the program in Figure 1. We use the ROSE compiler [9] to automate the instrumentation process. It is possible to avoid the instrumentation process by modifying the MCAPI library itself. However, such approach reduces the portability of DR-MCAPI and makes it implementation-specific. That defeats one of the goals of MCAPI of being an API standard with different implementations for different platforms. For example, DR-MCAPI, without any changes, is compatible with the new OpenMCAPI implementation [10].

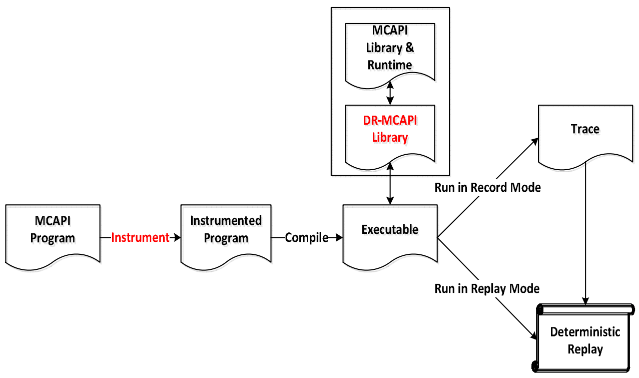


Figure 4: DR-MCAPI Workflow

The instrumented program is then compiled into an executable, which runs in one of two possible modes: record mode or replay mode. While a program is running in the record mode, calls to the DR-MCAPI library routines will use internal data structure to record certain information in addition to invoking the corresponding MCAPI library routines. For example, a call to dr_create_ep will add a new endpoint to a list of endpoints maintained for every node before invoking mcapi_create_ep. The recorded information is eventually stored to the disk.

```
N=NodesCount();
dr_init_node(ThisNode);
LocalEP= dr_create_ep(ThisNode,1);
for (Index=0;Index<N;Index++){
  if (Index== ThisNode) continue;
  RemoteEP= dr_get_ep(Index,1);
  dr_msg_send(LocalEP,RemoteEP,&Data[Index]);
}
for (Index=0;Index<N-1; Index++)
  dr_msg_recv(LocalEP,&Buffer[Index]);
dr_delete_ep(LocalEP);
dr_finalize_node(ThisNode);
```

Figure 5. An instrumented MCAPI program

If a MACPI program is executed in the replay mode, the trace information is loaded into memory and used by DR-MCAPI library to enforce an execution that is equivalent to the one observed when the program was running in the record mode. The replay mode is described in section 3.2

3.1 The Trace Structure

When an instrumented program is run in the record mode, a separate trace is generated for every MCAPI node. A node's trace contains a sequence of records that captures the program behavior during the execution. There are 7 types of records that are recorded in the trace.

A $Recv$ record originates from a msg_recv call and is defined as a tuple: $Recv \in Port \times RecvOrder \times Hash$. $Port$ is the port number of the receiving endpoint, $RecvOrder$ is the invocation order of this particular msg_recv call among other msg_recv calls at the same node, and $Hash$ is a hash-code of the received data calculated using the CRC-32 algorithm [11]. Figure 6 depicts three programs and their respective traces. The letters prefixed to trace records are shorthand for the records types. As shown in Figure 6a, endpoint with port number 1 at node 3 receives two messages (lines 8-9) that are sent from node 1 and node 2, respectively. The two records in the trace correspond to the arrival of two messages. In the first trace record (R:1/0/C0), 1 is the port number of the receiving endpoint, 0 is the order of the msg_recv call with respect to other msg_recv calls with the same endpoint and C0 signifies the hash-code of the received data.

A $Wait$ record originates from a wait call whose input request variable was initialized by a call to the function msg_recv_i. It is defined as $Wait \in ReqInitOrder \times Hash$. $ReqInitOrder$ is the initialization order of the wait function input request variable at the current node. In the program shown in Figure 6b, the request variable R1 was the first request variable initialized in this node (line 9) and hence its initialization order ($ReqInitOrder$) is 0. The trace record (W:1/C1), corresponding to line 19 in the source code indicates that a wait call retrieved a message with hash-code C1. In this record 1 is the initialization order of the request variable (R2).

```
Node 1:
1  LocalEP=create_ep(1,1);
2  RemoteEP=get_ep(3,1);
3  msg_send(LocalEP,RemoteEP,M1);

Node 2:
4  LocalEP=create_ep(2,1);
5  RemoteEP=get_ep(3,1);
6  msg_send(LocalEP,RemoteEP,M2);

Node 3:
7  LocalEP=create_ep(3,1);
8  msg_recv(LocalEP,&D1);
9  msg_recv(LocalEP,&D2);

Trace:
 R:1/0/C0
 R:1/1/C1
```

(a)

```
Node 1:
 1  LocalEP=create_ep(1,1);
 2  RemoteEP=get_ep(3,1);
 3  msg_send(LocalEP,RemoteEP,M1);

Node 2:
 4  LocalEP=create_ep(2,1);
 5  RemoteEP=get_ep(3,1);
 6  msg_send(LocalEP,RemoteEP,M2);

Node 3:
 7  LocalEP=create_ep(3,1);
 8  A=0;
 9  msg_recv_i(LocalEP,&D1,&R1);
10  msg_recv_i(LocalEP,&D2,&R2);
11  while(!test(R1))
12   {do_something();}
13  while(A<5)
14   {
15     if (test(R2)) break;
16     do_something();
17     A++;
18   }
19  if (A==5) wait(R2);

Trace:
T:0/10/C0
N:1/5
W:1/C1
```

(b)

```
Node 1:
 1  LocalEP=create_ep(1,1);
 2  RemoteEP=get_ep(3,1);
 3  msg_send(LocalEP,RemoteEP,M1);

Node 2:
 4  LocalEP=create_ep(2,1);
 5  RemoteEP=get_ep(3,1);
 6  msg_send(LocalEP,RemoteEP,M2);

Node 3:
 7  LocalEP=create_ep(3,1);
 8  msg_recv_i(LocalEP,&D1,Rs[0]);
 9  msg_recv_i(LocalEP,&D2,Rs[1]);
10  msg_recv_i(LocalEP,&D3,&R);
11  Index=wait_any(Rs);
12  if (Index==0)
13   wait(Rs[1]);
14  else
15   wait(Rs[0]);
16  wait(R);

Trace:
 A:0/1/C0
 W:0/C1
 W:2/C2
```

(c)

Figure 6. MCAPI programs and their traces

A $RecvTest$ record originates from a sequence (one or more) of test calls whose input request variable was initialized by a msg_recv_i call and does retrieve a message from the runtime buffers. It is defined as $RecvTest \in ReqInitOrder \times Count \times Hash$, such that $Count$ is the number of times the test call had failed, before succeeding and retrieving a message. In Figure 6b, the trace record (T:0/10/C0) means that a message with hash-code C0 was retrieved by the 11th invocation of a test call whose input request variable was the first to be initialized at this node.

Similarly, a $NonRecvTest$ record stems from a sequence of test calls. However, the $NonRecvTest$ record indicates that no messages were retrieved from the runtime buffers. That occurs when the input request variable was initialized by a non-blocking function other than msg_recv_i or when the input request variable was initialized by a msg_recv_i call and the sequence of test calls doesn't retrieve a message from the runtime buffers. In the second trace record in Figure 6b (N:1/5), number 5 means that a test call was invoked five times and number 1 means that the test call input request variable was the second request variable to be initialized at this node.

A Recv$Wany$ record comes from a wait_any call that returned the index of a request variable that was initialized by a msg_recv_i call and is defined as: $RecvWany \in WanyOrder \times Index \times Hash$. $WanyOrder$ is the order of this particular wait_any call among other wait_any calls at this node. $Index$ is the index returned by the wait_any call. The trace record (A:0/1/C0) in Figure 6c is explained as follows: number 0 means that this record comes from the first invocation of a wait_any call at this node, number 1 means that the wait_any call returned 1 and C0 signifies the hash-code of the retrieved message.

The record $NonRecvWany$ is defined as: $NonRecvWany \in WanyOrder \times Index$ and indicates that a wait_any call returned the index of a request variable that was initialized by a non-blocking function other than msg_recv_i.

A $Rand$ record represents a single invocation of the rand function and is defined as: $Rand \in RandOrder \times Value$. $RandOrder$ is the invocation order of this particular rand call among other rand calls at this node. $Value$ is the random number returned by the rand call.

Note that the traces in Figure 6 belong to the third node in the three programs. Trace records correspond to 1) retrieving a message from the runtime buffers, 2) test invocations, or 3) wait_any invocations. None of these appears in nodes 1 and 2.

3.2 The Replay Mode

To realize a correct replay of a program, it is necessary to associate endpoints, request variables and certain calls that were observed during the record mode with their counterparts in the replay mode. An endpoint observed in the replay mode is associated with the corresponding endpoint in the record mode via the node identifier and the port number, both of which remain the same across executions. Request variables are tracked across executions in the record mode and the replay mode using their order of initialization in a node. For example, in Figure 6b, the request variable that was initialized first will always be the request variable that is passed to the test call in line 11. Similarly, msg_recv, wait_any and rand calls are tracked by their invocation order with respect to other msg_recv, wait_any and rand calls, respectively.

To enable the replay mode, we maintain three data structures. 1) $Records$: a list of the trace records (e.g. $Recv$, $Wait$…). 2) $RequestVariables$: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly during the execution in replay mode. When a request variable is initialized (by being passed to a non-blocking call), a new item is appended to this list. If the request variable is initialized by a non-blocking receive call, we keep track of the receiving endpoint and the destination buffer pointer. If the trace indicates that test calls were used to check the status of this request in the record mode, then the number of failed tests is retrieved from the trace and associated with that request. All newly initialized requests are flagged as incomplete. 3) $RecievedMessages$: messages that arrive earlier than expected are stored in this list along with their hash-codes.

The algorithm in Figure 7 handles dr_msg_recv calls. First RecvCalls is incremented (line 1). RecvCalls keeps track of the number of dr_msg_recv function invocations at the node. Second, the GetRecvRecord procedure looks up the $Records$ list to fetch the $Recv$ record with $RecvOrder$=RecvCalls (line 2). Third, the hash-code of the expected message is retrieved (line 3). Next, $RecievedMessages$ is looked up for a message whose hash-code matches the expected hash-code. If such a message is found, then its data is copied to the program buffer (line 7) and then removed from $RecievedMessages$ (line 8). Otherwise, the mcapi_msg_recv is repeatedly invoked till it retrieves a message whose hash-code matches the expected hash-code (lines 11-22). When the excepted message arrives, it is copied to the program buffer (line 16) and then removed from $RecievedMessages$ (line 17). All other messages and their hash-codes are appended to $RecievedMessages$ (line 21).

```
dr_msg_recv(Endpoint, &Buffer)
 1   RecvCalls++;
 2   RecvRecord=GetRecvRecord(RecvCalls);
 3   ExpectedCRC=RecvRecord.Hash;
 4   for Index=0 to RecievedMessages.size do
 5     if (RecievedMessages[Index].CRC==ExpectedCRC)
 6      then
 7       copy(Buffer, RecievedMessages[Index]);
 8       free(RecievedMessages[Index]);
 9       return;
10    end-if
11   while(true) do
12     mcapi_msg_recv(Endpoint,&TempBuffer);
13     ArrivedCRC=CalculateCRC(TempBuffer);
14     if (ArrivedCRC==ExpectedCRC)
15      then
16       copy(Buffer, TempBuffer);
17       free(RecievedMessages[Index]);
18       return;
19      end-if
20     else
21      RecievedMessages.Append(TempBuffer, ArrivedCRC);
22   end-while
```

Figure 7. The dr_msg_recv procedure

In the program in Figure 6a, node 3 receives two messages. Let's assume that when running that program in the record mode, it generates the trace in the figure (i.e. the first msg_recv call retrieves a message with hash-code *C0* and the second msg_recv call retrieves a message with hash-code *C1*). Let's assume that during running the program in the replay mode, the message with hash-code *C1* arrives first. During the replay mode execution, when dr_msg_recv is invoked for the first time, the $RecievedMessages$ list will be empty. Hence, the while loop (lines 11-22) will iterate twice. In the first iteration, the mcapi_msg_recv call will retrieve the message with hash-code *C1*. Since the retrieved message is not the excepted one, it will be added to the $RecievedMessages$ list (line 21). In the second iteration, the mcapi_msg_recv call will retrieve the message with hash-code *C0*, which is the expected message and it will be delivered to the program (line 16). When dr_msg_recv is invoked for the second time, the $RecievedMessages$ list will contain the expected message (with hash-code *C1*) and it will be returned to the program (line 7).

Figure 8 shows the algorithm that handles a dr_wait call whose input request variable was initialized by a msg_recv_i call. This algorithm depends on the $RequestVariables$ list that links a request variable with the endpoint and the program buffer pointer that were passed to the msg_recv_i call.

```
dr_wait(Request)
 1   if not IsRecvRequest(Request) then
 2     return mcapi_wait(Request);
 3   end-if
 4   InitOrder=GetInitOrder(Request);
 5   WaitRecord=GetWaitRecord(InitOrder);
 6   ExpectedCRC=WaitRecord.Hash;
 7   BufferPtr=GetBufferPtr(Request);
 8   Endpoint=GetEndpoint(Request);
 9   Requests=GetRequests(CurrentNode);
10   for Index=0 to Requests.size() do
11    if (Requests[Index].isComplete)  then continue;
12    mcapi_wait(Requests[Index]);
13    ArrivedData=GetData(Requests[Index]);
14    ArrivedCRC=CalculateCRC(ArrivedData);
15    RecievedMessages.Append(ArrivedData,ArrivedCRC);
16    Requests[Index].setComplete();
17    end-for
18   for Index=0 to RecievedData.size() do
19    if (RecievedMessages[Index].CRC==ExpectedCRC)
20      then
21      copy(BufferPtr, RecievedMessages[Index]);
22      free(RecievedMessages[Index]);
23      return;
24    end-if
25   end-for
```

Figure 8. The dr_wait procedure

First, if the input request was not initialized by a dr_msg_recv_i call, then it is forwarded to the MCAPI library (lines 1-3). Otherwise, the hash-code of the expected message, the endpoint and the program buffer pointer associated with the input request variable are retrieved (lines 4-8). Second, mcapi_wait is invoked for all initialized (but not completed) requests at that node and retrieved messages and their hash-codes are appended to $RecievedMessages$ (lines 9-17). Finally, $RecievedMessages$

is looked up for a message whose hash-code matches the expected hash-code. When such message is found, it is copied to the buffer associated with the input request variable (line 21) and then removed from $RecievedMessages$ (line 22).

Figure 9 describes how DR-MCAPI handles dr_wait_any calls. First, WaitanyCalls is incremented (line 1). WaitanyCalls keeps track of the number of dr_wait_any function invocations at the node. If the current dr_want_any call retrieves a message, then the GetRecvWanyRecord procedure looks up the $Records$ list to fetch the $RecvWany$ record with $WanyOrder$ =WaitanyCalls (line 3). In line 4, the $Index$ in the $RecvWany$ record is retrieved and the request in the Requests array at $Index$ will be forwarded to dr_wait (line 5). If the current dr_want_any call doesn't retrieve a message, then the GetNRecvWanyRecord procedure looks up the $Records$ list to fetch the $NonRecvWany$ record with $WanyOrder$ =WaitanyCalls (line 7). In line 8, the $Index$ in the $NonRecvWany$ record is retrieved and the request in the Requests array at $Index$ will be forwarded to mcapi_wait (line 9). Finally, $Index$ is returned to the application (line 11).

```
dr_wait_any(Requests)
1   WaitanyCalls++;
2   if RecvWany(WaitanyCalls) then
3    RecvWanyRecrd=GetRecvWanyRecord(WaitanyCalls);
4    Index=RecvWanyRecord.Index;
5    dr_wait(Requests[Index]);
6   else
7    NRecvWanyRecrd=GetNRecvWanyRecord(WaitanyCalls);
8    Index=NRecvWanyRecrd.Index;
9    mcapi_wait(Requests[Index]);
10  end-if
11  return Index;
```

Figure 9. The dr_wait_any procedure

In the program in Figure 6c, node 3 receives three messages. Let's assume that when running that program in the record mode, it generates the trace in the figure (i.e. the wait_any call returns 1 and retrieves a message with hash-code *C0*, the wait call in line 15 retrieves a message with hash-code C1, and the wait call in line 16 retrieves a message with hash-code *C2*). Let's assume that during replay, the messages arrive with a different order (C2, C0 then C1). When dr_wait_any is invoked, it is going to determine that the request at index 1 of the array $Requests$ was initialized by a msg_recv call and will forward this request to dr_wait. In dr_wait, the first loop (lines 10-17) will retrieve the three messages via three calls to mcapi_wait (line 12) and they will be added to the $RecievedMessages$ list (line 15). The second loop in dr_wait (lines 18-25) will iterate through the $RecievedMessages$ list and will return the message with hash-code C0 to the application. When dr_wait is invoked to handle the wait calls at lines 15 and 16 (in Figure 6c), the messages with hash-codes C1 and C2 will be already in the $RecievedMessages$ list and will be returned to the application at the correct order.

A dr_test call is handled by the algorithm in Figure 10. First, the initialization order of the input request variable (Request) is retrieved (line 1). If that request variable is associated with an $ArrivalTest$ record, then the $Count$ of this record is reduced by one (line 4). If $Count$ reaches zero, the request is forwarded to dr_wait and true is returned to the application (lines 8-9). If that request variable is associated with a $NonArrivalTest$ record, then the $Count$ of this record is reduced by one (line 13). If $Count$ reaches zero, the request is passed to mcapi_wait and true is returned to the application (lines 17-18).

In the program in Figure 6b, node 2 receives two messages. Let's assume that when run in the record mode, this program generates the trace in the figure (i.e. the test call in line 11 fails 10 times then succeeds and retrieves a message with hash-code C0 at the 11th invocation, the test call in line 15 fails 5 times and the wait call in line 12 retrieves a message with hash-code C1). Let's assume that during replay, the messages arrive with a different order (C1 then C0). When dr_test is invoked to handle the test call at line 11, it will return *false* for 10 times (lines 5-6) and after that, it will invoke dr_wait (line 8) to retrieve the message with hash-code C0. When dr_test is invoked to handle the test call at line 15, it will return *false* for 5 times. Finally, the dr_wait call in (line 19 in Figure 6b) will retrieve the message with hash-code C1.

```
bool dr_test(Request)
1   InitOrder=GetInitOrder(Request);
2   if ArrivalTest(InitOrder) then
3    ArrivalTestRecord=GetArrivalTestRecord(Order);
4    ArrivalTestRecord.Count--;
5    if ArrivalTestRecord.Count>0 then
6     return false;
7    else
8     dr_wait(Request);
9     return true;
10   end-if
11  else
12   NArrivalTestRecord=GetNArrivalTestRecord(Order);
13   NArrivalTestRecord.Count--;
14   if NArrivalTestRecord.Count>0 then
15    return false;
16   else
17    mcapi_wait(Request);
18    return true;
19   end-if
20  end-if
```

Figure 10. The dr_test procedure

4. EXPERIMENTAL EVALUATION

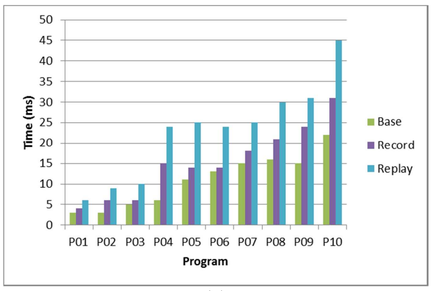
Due to the lack of publicly available MCAPI benchmarks, we performed experiments on three sets of MCAPI programs developed internally and a set of programs obtained from an external source [12]. Our experiments were conducted on a machine with Core 2 Duo 1.4 GHz CPU and 4GB RAM using MCAPI runtime V1.063. We evaluate DR-MCAPI using the following set of programs:

1. Binary tree benchmark: This is a set of 10 programs that create networks of nodes with sizes from 3 nodes to 21 nodes. Each two nodes send a message to the same parent node forming a binary tree in which messages travel from the leaves to the root node. The smallest tree has 3 nodes and exchanges 20 messages. The largest one has 21 nodes

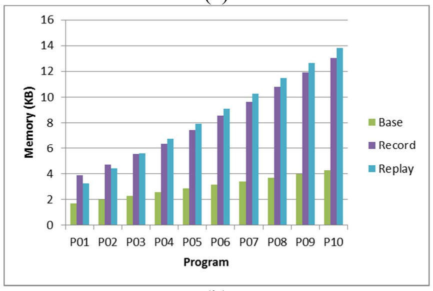
and exchanges 155 messages. This benchmark has a master/slave communication pattern.

2. Complete graph benchmark: This is a set of 10 programs that create networks of nodes with increasing sizes from 2 nodes to 13 nodes. All nodes send and receive messages to/from each other forming a complete graph. The number of exchanged messages is between 20 message (for a 2 nodes graphs) and 1560 messages (for a 13 nodes graph). This benchmark has an all-to-all communication pattern.
3. 10-nodes benchmark: In this benchmark, the number of nodes is fixed to 10, but the number of messages exchanged increases with every program. This allows us to isolate the effect of the number of messages on performance. There are 10 programs in this benchmark.
4. Bully benchmark: This is a MCAPI implementation of the Bully leader selection algorithm [13] provided by the V&V research group at Brigham Young University.

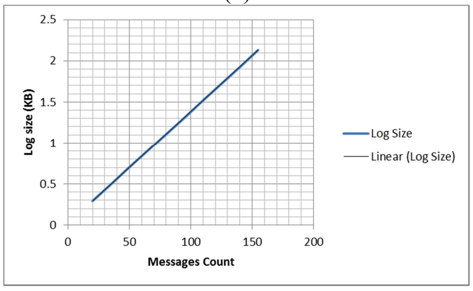
Figure 11 shows the results of the binary tree benchmark. Figure 11a compares the recording, replay time and baseline time of running the programs of the benchmark. The average percentage increase in recording time is 49% while the average percentage increase in replay time is 127%. Figure 11b compares the memory usage while recording, replaying and in a baseline execution. The average percentage increase in memory usage in the recording mode is 167% while it is 174% during replay mode. Figure 11c shows the size of the trace file with respect to the number of exchanged messages.



(a)



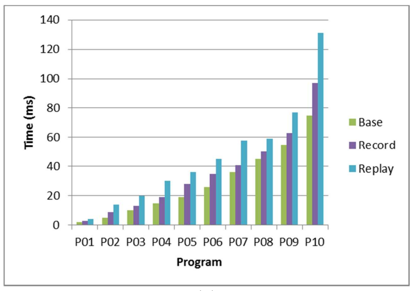
(b)



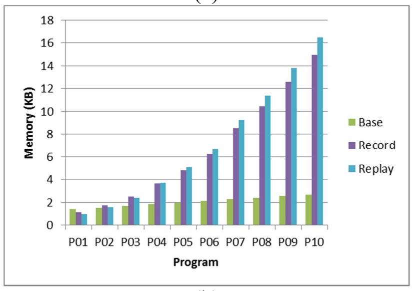
(c)

Figure 11. Results of the binary tree benchmark

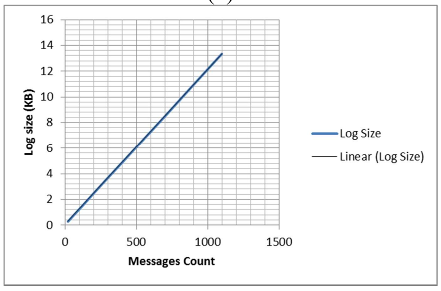
Figure 12 shows the results of the complete graph benchmark. The average percentage increase in recording time is 33% while the average percentage increase in replay time is 84%. The average percentage increase in memory usage in the recording mode is 193% while it is 211% during replay mode.



(a)



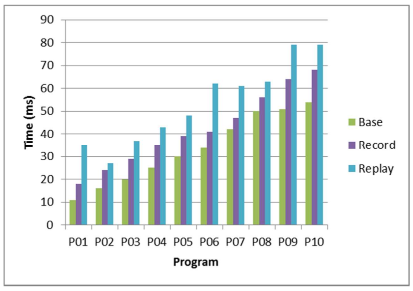
(b)



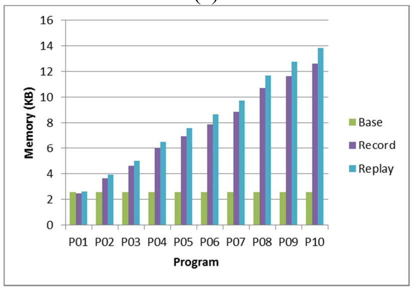
(c)

Figure 12. Results of the complete graph benchmark

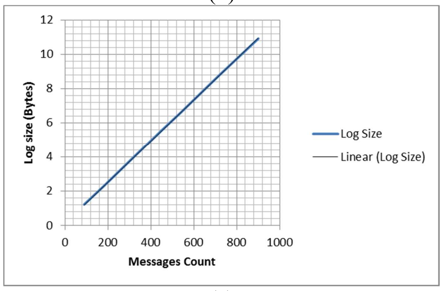
Figure 13 shows the results of the 10-nodes benchmark. The average percentage increase in recording time is 32% while the average percentage increase in replay time is 75%. The average percentage increase in memory usage in the recording mode is 195% while it is 222% during replay mode.



(a)



(b)



(c)

Figure 13. Results of the 10-nodes benchmark

The single program in the Bully benchmark had a 22% and 51% time increase in the recording mode and replay mode, respectively and 99% and 117% memory usage increase in the recording mode and replay mode, respectively. The experiments show that the average time increase is 38% in the recording mode and 96% in the replay mode and that the average memory usage increase in 184% in the recording mode and 203% in the replay mode. We are experimenting with check-pointing to reduce the memory usage increase. In all benchmarks, the log size increases linearly with respect to the number of messages.

5. RELATED WORK

MPI [1] has been the dominating message-passing software development standard. Hence, the current literature on replaying message-passing software is almost limited to MPI programs. In [4], Kranzlmuller et al. present a record and replay mechanism for MPI that adopts the order-replay approach and handles both promiscuous receive calls and test operations. Their approach is based on modifying the MPICH library source code. Different than MCAPI, not all MPI receive calls are promiscuous. MPI receive calls have a source parameter that can be used to state a specific sender process. If the source parameter is set to MPI_ANY_SOURCE, then the receive call may receive a message from any process allowing message races, otherwise, no message races can take place. Receive calls with MPI_ANY_SOURCE are handled by storing the identifier of the source process of the message that was received during the recording phase. During replay, when the source parameter of a receive call is MPI_ANY_SOURCE, it is replaced with the source process identifier obtained during the recording phase. This approach is not applicable to MCAPI programs since their receive calls (msg_recv and msg_recv_i) do not specify a source parameter. Test operations are handled by counting the number of consecutive failing test operations associated with the same request variable during the recording phase. In the replay phase, test operations are forced to fail (i.e. return false) till the recorded number of failed tests has reached. They report a 200% time increase in the recording phase. Also, this approach is library-dependent (based on the MPICH library) which limits its portability to other MPI implementations.

In [5], the authors disabuse the impracticality of data-replay and argue that the ability to replay one process justifies the excessive logging overhead. They implement their data-replay mechanism as a layer between the application and the MPI library. Recorded data includes: MPI function calls return values and the contents and the source processes identifiers of received messages. During replay, when the application posts a receive call; the data-replay layer returns the data recorded at the corresponding receive call during the recording phase. In other words, receive calls are *simulated* rather than being executed. As expected, the log size is 100's of times larger than when order-replay is used. In one experiment, the data log was 907MB while an order-replay would produce 0.84MB for the same program. The disk space requirement of this approach is prohibitively large for long-running applications. Unfortunately the approaches described in [4] and [5] don't capture all forms of non-determinism in MPI programs, making it difficult to ensure a completely faithful replay.

The authors of [14] propose subgroup-reproducible replay (SRR) which combines order-replay and data-replay. During the recording phase, disjoint groups of processes are formed and the contents of messages crossing group boundaries are recorded. The contents of the messages that are sent and received within a group are not recorded, but the order of arrival of such messages is recorded. This approach allows replaying a specific group of processes independently of other groups. During replay of a group, messages coming from outside that group are reproduced from the log; inter-group messages are produced through direct execution. Setting the size and the membership of groups can be done manually by the user or automated based on communication locality. Performance evaluation of the SRR approach shows that it increases the runtime by an average of 120% during the recording phase and generates a log that is half the size of the log generated by a pure data-replay approach. Also this work handles all non-determinism sources in MPI programs.

Another related tool is MCC [15] which implements an automated approach for verifying MCAPI programs. MCC creates a scheduling layer above the MCAPI runtime layer that allows intercepting MCAPI calls and discovering potentially matching send/receive ones. This allows MCC to explore all possible execution scenarios resulting from different orders of messages' arrival. MCC uses DPOR [16] technique to reduce the number of examined execution scenarios. MCC handles only promiscuous receive calls making it unsuitable for any programs using mcapi_test and mcapi_wait_any calls.

6. CONCLUSION

To the best of our knowledge, DR-MCAPI is the first replay tool that considers all non-determinism sources in MCAPI programs. DR-MCAPI introduces deterministic replay capabilities to MCAPI developers, thus, allowing detecting bugs induced by message-races and time-dependent events via cyclic debugging [17]. During a recording phase, an unobstructed execution of the input program is monitored to produce a trace that contains the outcomes of non-deterministic MCAPI operations. During a replay phase, the stored trace is used to enforce an execution that is equivalent to the one observed in the recording phase. This allows the programmer to repeatedly execute the program under supervision of a debugger to catch flaws. Our approach imposes an average of 38% and 96% time overhead in the recording phase and replay phase, respectively. We are currently developing an Eclipse plugin that uses the replay mechanism as a back-end to allow the user to perform interactive debugging. Also, we plan to reduce the memory usage by using check-pointing and compression techniques.

ACKNOWLEGEMENT: We would like to thank Dr. Eric Mercer and his group for providing us the Bully benchmark. The work was supported in part by NSF under contract CCF-0811287, and by ONR under Grant N000140910740.

7. REFRENCES

[1] "MPI: A Message-Passing Interface Standard," http://www.mpi-forum.org/docs/mpi-2.2/index.htm, [Online; accessed 28-April-2011].

[2] S. Brehmer, "The Multicore Association Communications API," http://www.multicore-association.org/workgroup/mcapi.php, March 2010, [Online; accessed 28-April-2011].

[3] D. F. Snelling and G.-R. Hoffmann, "A comparative study of libraries for parallel processing," *Parallel Computing*, vol. 8, no. 1-3, pp. 255 – 266, 1988, proceedings of the International Conference on Vector and Parallel Processors in Computational Science III.

[4] D. Kranzlmüller, C. Schaubschläger, and J. Volkert, "An integrated record&replay mechanism for nondeterministic message passing programs," in *Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK: Springer-Verlag, 2001, pp. 192–200.

[5] M. Maruyama, T. Tsumura, and H. Nakashima, "Parallel program debugging based on data-replay." in *IASTED PDCS'05*, 2005, pp. 151–156.

[6] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira, "Offline symbolic analysis for multi-processor execution replay," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 564–575. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669182

[7] D. Lee, M. Said, S. Narayanasamy, and Z. Yang, "Offline symbolic analysis to infer total store order," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 357 –358.

[8] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal, "Debugging race conditions in message-passing programs," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, ser. SPDT '96. New York, NY, USA: ACM, 1996, pp. 31–40.

[9] D. J. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[10] M. Levy, "Mentor Releases Open Source of MCAPI for Multicore," http://www.multicore-association.org/press/030211.html, March 2011, [Online; accessed 28-April-2011].

[11] Ross Williams, "A Painless Guide to CRC Error Detection Algorithms V3.00," http://www.ross.net/crc/crcpaper.html, April 2010, [Online; accessed 28-April-2011].

[12] E. Mercer, "Verification and Validation Laboratory at Brigham Young University," http://facwiki.cs.byu.edu/vv-lab/index.php, April 2011, [Online; accessed 28-April-2011].

[13] H. Garcia-Molina, "Elections in a distributed computing system," *IEEE Trans. Comput.*, vol. 31, pp. 48–59, January 1982.

[14] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "MPIWiz: subgroup reproducible replay of MPI applications," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 251–260.

[15] S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt, "MCC: A runtime verification tool for MCAPI user applications," in *FMCAD*. IEEE, 2009, pp. 41–44.

[16] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, pp. 110–121, January 2005.

[17] M. Ronsse, M. Christiaens, and K. D. Bosschere, "Cyclic debugging using execution replay," in *Proceedings of the International Conference on Computational Science-Part II*, ser. ICCS '01. London, UK, UK: Springer-Verlag, 2001, pp. 851–860.