

DETERMINISTIC REPLAY FOR MESSAGE-PASSING BASED CONCURRENT PROGRAMS

MOHAMED ELWAKIL, Western Michigan University

ZIJIANG YANG, Western Michigan University

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 parallel 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 capable of deterministically replaying MCAPI programs executions, which provides valuable insight for MCAPI developers in case of failure.

This is an expanded version of a paper accepted at the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD-IX) 2011. The current article extends the previous paper by (1) presenting two new techniques for replaying MCAPI programs: data-replay in Section 3.2 and sender-based order-replay in Section 3.3.1, (2) differentiating our work from comprehensive related work in Section 6.

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

General Terms: Reliability

Additional Key Words and Phrases: MCAPI, message race, multicore programs, debugging, deterministic replay

ACM Reference Format:

Elwakil, M., and Zijiang, Yang. 2011. Deterministic Replay for MCAPI Programs. *ACM Trans. on Design Automation of Electronic Syst.* 9,4, Article 00 (September 2011), 30 pages.

DOI = 10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

1. INTRODUCTION

If two executions of a program exhibit the same set of instructions with each instruction computing the same results and producing the same final values in memory, then these two executions are said to be logically equivalent [17]. A deterministic replay of a program is a controlled execution that is logically equivalent to a previous execution of interest. Deterministic replay has various applications such as cyclic debugging, fault tolerance and intrusion analysis [26].

In cyclic 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 [22]. Cyclic debugging assumes that different executions of the same program with the same input will be equivalent. Different executions of a concurrent program are not guaranteed to be equivalent as concurrent programs suffer from the irreproducibility effect [27] due to their intrinsic non-determinism. The fact that two subsequent runs of the same program with the same input are not guaranteed to behave the same or produce the same output, makes cyclic debugging of concurrent

This work is supported by the National Science Foundation, contract CCF-0811287. We would like to thank Dr. Eric Mercer and his group for providing us the Bully benchmark

Author's addresses: M. Elwakil, Department of Computer Science, Western Michigan University; Z. Yang Department of Computer Science, Western Michigan University.

Permission to make digital or hardcopies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credits permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

@2011 ACM 1539-9087/2010/03-ART39 \$10.00

DOI10.1145/0000000.0000000 <http://doi.acm.org/10.1145/0000000.0000000>

programs a challenging task. Cyclic debugging is the most prominent application of deterministic replay and is called Deterministic Replay Debugging (DRD) [17].

Within the context of fault-tolerance, deterministic replay has been used to detect hardware design faults by scrutinizing the variances between a replayed execution on a machine and an original execution on another machine [14]. Also, in the case of a program failure, a replayed execution can be used to reconstruct the most recent program state [25]. The ReVirt system [5] shows that deterministic replay is useful for intrusion analysis. ReVirt allows replaying the execution of a whole computer system before, during, and after a system has been compromised facilitating post-attack analysis.

As depicted in Fig. 1, the deterministic replay process consists of two phases: recording and replay. During the recording phase, a program execution is monitored by a recording environment to record information about the execution in a trace file. When a replay is needed, the data in the trace file is used to replay the program within a replay environment such that the behavior of the program during the replay phase is logically equivalent to the behavior observed in the recording phase.

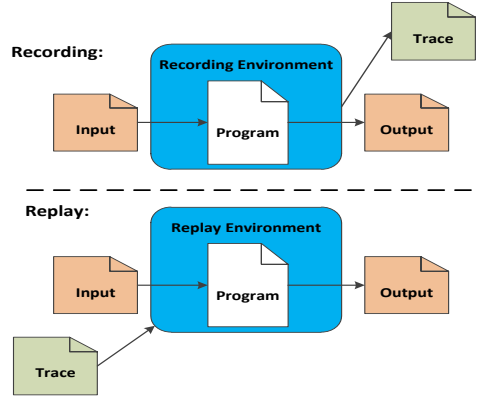


Fig. 1. The two phases of deterministic replay

In 2008, the Multicore Association [2], a consortium of major corporations and leading research centers, published the first version of the Multicore Communications API (MCAPI) [3]. MCAPI is a new message-passing API that is intended for systems with multiple cores on a chip and/or chips on a board. MCAPI 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 [1]. Currently, there are two implementations of the MCAPI specification; the standard implementation provided by the MCA and a relatively newer implementation provided by Mentor Graphics [13]. Both implementations are C libraries.

An MCAPI program execution is inherently nondeterministic, thus repeatedly executing the same program on the same input may yield different results, making debugging extremely difficult. Providing a deterministic replay capability allows developers to observe the same execution making finding failure source an easier task. In this paper, we present DR-MCAPI, the first tool for deterministically replaying MCAPI programs executions. DR-MCAPI works by monitoring a program execution to generate a trace. If the program fails, the trace can be used to produce an execution that is logically equivalent to the one that had failed. DR-MCAPI implements two replay approaches: data-replay and order-replay. Each approach has its own particular strengths and weaknesses. Section 2 provides an overview of the MCAPI specification and the sources of non-determinism in MCAPI programs. Section 3 describes the workflow of DR-MCAPI and the data-replay and order-replay approaches. Section 4 shows the results of experiments conducted using four benchmarks and compares our MCAPI programs replay techniques in terms of memory and time overheads. Section 5 discusses the features of DR-MCAPI in terms of usability, portability and scalability. In Section 6 we present a comprehensive overview of related work and we conclude and present future directions in Section 7.

2. NON-DETERMINISM IN MCAPI PROGRAMS

In MCAPI programs, a CPU 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 until the message has been copied from a program buffer to the MCAPI runtime buffers, `mcapi_msg_send_i` is non-blocking and returns immediately after initiating the copying process. The functions `mcapi_msg_recv` and `mcapi_msg_recv_i` retrieve messages from the MCAPI runtime buffers. The function `mcapi_msg_recv` blocks until 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 receives 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 (e.g. `mcapi_msg_recv_i`) takes a request variable as an extra input parameter and initializes it by setting its value to pending. Calling `mcapi_wait` with a request variable *R* blocks execution until the non-blocking operation that initialized *R* has completed. Invoking `mcapi_wait_any` with an array of request variables *Rs* blocks execution until 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, and 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 [19]. Fig. 2 shows an MCAPI program with message races. A node creates a single endpoint and sends messages to all other nodes (lines 1-8) 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 messages arrivals can change across consecutive executions of the program leading to the irreproducibility effect (i.e. the final values in the **Buffer** array will not be the same with different executions). It should be noted that, in contrast to multithreaded programs data races, the mere existence of a message race is not an error condition by itself, but a message race *may* lead to an error condition. This fact makes debugging message passing programs even harder, as one need not only to detect message races, but also to examine the consequences of a message race.

```
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);
```

Fig. 2. An MCAPI program with message races

Another source of non-determinism in MCAPI programs is the `mcapi_wait_any` call. In Fig. 3, 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 not be the same across consecutive executions of the program, which results in varying 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_rcv_i(LocalEP1,&Buffer1,Requests[0]);
5  mcapi_msg_rcv_i(LocalEP2,&Buffer2,Requests[1]);
6  ReqIndex=mcapi_wait_any(Requests,2);
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);

```

Fig. 3. An MCAPI program with `mcapi_wait_any`

Using the non-blocking `mcapi_test` introduces non-determinism as well. In Fig. 4, a node is expecting 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.

```

1  mcapi_init_node(ThisNode);
2  LocalEP=mcapi_create_ep(ThisNode,1);
3  mcapi_msg_rcv_i(LocalEP,&Buffer,Request);
4  while (!mcapi_test(Request)) A++;
5  func_call(A);
6  mcapi_delete_ep(LocalEP);
7  mcapi_finalize_node(ThisNode);

```

Fig. 4. An MCAPI program with `mcapi_test`

In summary, besides user inputs and random numbers generators, there are three additional sources of non-determinism in MCAPI programs: 1) `mcapi_msg_rcv` and `mcapi_msg_rcv_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 an MCAPI program developer.

3. DR-MCAPI: DETERMINISTIC REPLAY FOR MCAPI PROGRAMS

Fig. 5 depicts the workflow of our tool for deterministic replay of MCAPI programs. DR-MCAPI consists of two parts: a source code instrumenter and an 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 by replacing the prefix “`mcapi`” with “`dr`”. Also, the instrumenter adds code that initializes and tears down the DR-MCAPI library.

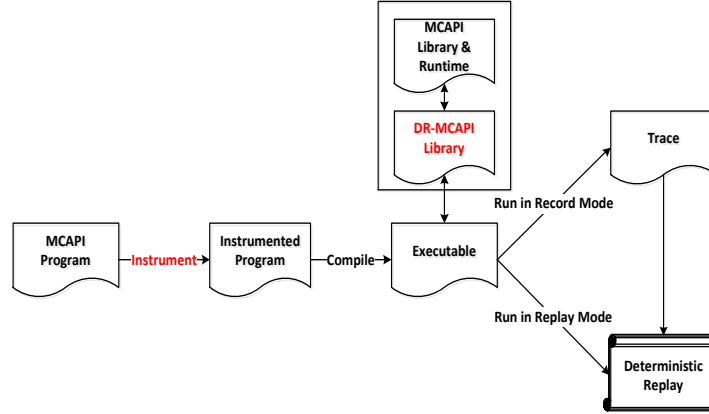


Fig. 5. DR-MCAPI Workflow

Fig. 6 shows the result of instrumenting the program in Fig. 2. We use the ROSE compiler [21] 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 [13].

```

1  N=NodesCount();
2  dr_init_node(ThisNode);
3  LocalEP= dr_create_ep(ThisNode,1);
4  for (Index=0;Index<N;Index++){
5      if (Index== ThisNode) continue;
6      RemoteEP= dr_get_ep(Index,1);
7      dr_msg_send(LocalEP,RemoteEP,&Data[Index]);
8  }
9  for (Index=0;Index<N-1; Index++)
10     dr_msg_rcv(LocalEP,&Buffer[Index]);
11  dr_delete_ep(LocalEP);
12  dr_finalize_node(ThisNode);
  
```

Fig. 6. An instrumented MCAPI program

The DR-MCAPI library acts as a layer between the program and the MCAPI library as shown in Fig. 7. When an instrumented program is run, the program invokes the DR-MACAPI routines which will carry out some processing and call the original MCAPI routine. For example, a call to **dr_create_ep** will add a new endpoint to a list of endpoints maintained for every node by DR-MCAPI and then **mcapi_create_ep** will be invoked.

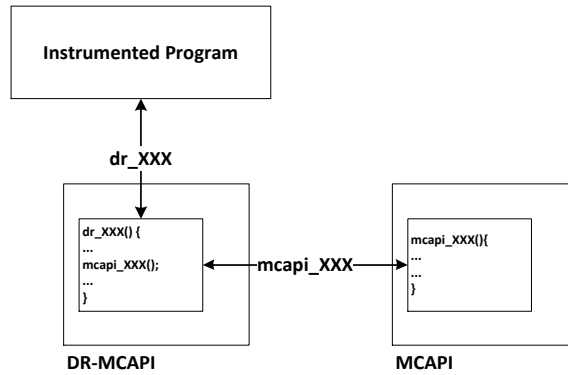


Fig. 7. DR-MCAPI is a layer between the MCAPI library and the instrumented program

3.1 Operating Modes

An instrumented program can run in one of two possible operating modes: recording mode or replay mode. While a program is running in the recording mode, calls to the DR-MCAPI library routines record certain information in addition to invoking MCAPI library routines. When the program execution ends (either normally or by a failure), the recorded information is stored to the disk as a trace. During the recording mode, DR-MCAPI doesn't affect the outcomes of non-deterministic operations.

When run in the replay mode, the trace information is loaded into memory and is used by DR-MCAPI library to force an execution that is equivalent to the one observed when the program was running in the recording mode.

Replay tools for message-passing programs typically fall into two categories: data-replay and order-replay. In data-replay, the contents of all received messages at all nodes are stored during the recording phase, while in order-replay, only the outcomes of non-deterministic operations are recorded. DR-MCAPI supports both data-replay and order-replay. Section 3.2 describes how DR-MCAPI implements data-replay and section 3.3 shows how DR-MCAPI implements order-replay.

Fig. 8 depicts the pseudocode of an instrumented MCAPI program with five nodes. Nodes 1 and 3 send three messages to nodes 2, 4 and 5, each. Node 2 uses the data in the two received messages to calculate a new value and send it to nodes 4 and 5. Hence, nodes 4 and 5 are expecting to receive three messages, each. We will be using this program as an ongoing example throughout this article.

Node 1	Node 3
1 dr_initialize(1);	12 dr_initialize(3);
2 X=1;	13 Y=10;
3 EP=dr_create_endpoint(1);	14 EP=dr_create_endpoint(1);
4 N2EP=dr_get_endpoint(2,1);	15 N2EP=dr_get_endpoint(2,1);
5 N4EP=dr_get_endpoint(4,1);	16 N4EP=dr_get_endpoint(4,1);
6 N5EP=dr_get_endpoint(5,1);	17 N5EP=dr_get_endpoint(5,1);
7 dr_msg_send(EP,N2EP,X);	18 dr_msg_send(EP,N2EP,Y);
8 dr_msg_send(EP,N4EP,X);	19 dr_msg_send(EP,N4EP,Y);
9 dr_msg_send(EP,N5EP,X);	20 dr_msg_send(EP,N5EP,Y);
10 dr_delete_endpoint(EP);	21 dr_delete_endpoint(EP);
11 dr_finalize();	22 dr_finalize();
Node 2	Node 4
23 dr_initialize(2);	39 dr_initialize(4);
24 EP=dr_create_endpoint(1);	40 EP=dr_create_endpoint(1);
25 N4EP=dr_get_endpoint(4,1);	41 dr_msg_rcv_i(EP,&D,&R1);
26 N5EP=dr_get_endpoint(5,1);	42 dr_msg_rcv_i(EP,&E,&R2);
27 dr_msg_rcv_i(EP,&A,Rs[0]);	43 dr_msg_rcv(EP,&F);
28 dr_msg_rcv_i(EP,&B,Rs[1]);	44 while(!dr_test(R1));
29 if(dr_wait_any(Rs,2)==0)	45 dr_wait(R2);
30 dr_wait(Rs[1]);	46 dr_delete_endpoint(EP);
31 else	47 dr_finalize();
32 dr_wait(Rs[0]);	
33 dr_assert(A>B)	Node 5
34 C=Func(A,B);	48 dr_initialize(5);
35 dr_msg_send(EP,N4EP,C);	49 EP=dr_create_endpoint(1);
36 dr_msg_send(EP,N5EP,C);	50 dr_msg_rcv(EP,&G);
37 dr_delete_endpoint(EP);	51 dr_msg_rcv(EP,&H);
38 dr_finalize();	52 dr_msg_rcv(EP,&I);
	53 dr_delete_endpoint(EP);
	54 dr_finalize();

Fig. 8. Pseudocode of an MCAPI Program

3.2 DR-MCAPI Data-replay

During a recording execution, the contents of all received messages at all nodes are stored. During a replay execution, it is possible that some nodes are run while others are simulated. The messages sent by the simulated processes originate from the trace and *not* from the program execution. The data-replay approach generates a huge trace. However, it allows replaying one or more specific nodes. First, we describe the data-relay trace structure in section 3.2.1 and then the data-replay replay mechanism in section 3.2.2.

3.2.1 The Trace Structure

When an instrumented program P is run in the recording mode, a separate trace is generated for each MCAPI node:

$Trace_P = \{Trace^1, \dots, Trace^N\}$, where N is the number of nodes in program P .

A node's trace contains a list of records:

$Trace^n = \{Record^1, \dots, Record^{|Trace^n|}\}$, where n is a node identifier.

A trace record may be any of six types:

$Record \in Recv \cup Wait \cup RecvWany \cup NonRecvWany \cup ArrivalTest \cup NonArrivalTest$

A *Recv* record originates from a **dr_msg_rcv** call and is defined as a tuple: $Recv \in Port \times RecvOrder \times Data$. *Port* is the port number of the receiving endpoint. *RecvOrder* is the invocation order of this particular **dr_msg_rcv** call among other **dr_msg_rcv** calls at this node. *Data* is the payload of the received message.

A *Wait* record originates from a **dr_wait** call whose input request variable was initialized by a **dr_msg_rcv_i** call and is defined as tuple: $Wait \in ReqInitOrder \times Data$. *ReqInitOrder* is the initialization order of the input request variable at the current node.

A *RecvWany* record comes from a **dr_wait_any** call that returned the index of a request variable that was initialized by a **dr_msg_rcv_i** call and is defined as tuple: $RecvWany \in WanyOrder \times Index \times Data$. *WanyOrder* is the invocation order of this particular **dr_wait_any** call among other **dr_wait_any** calls at this node. *Index* is the index returned by the **dr_wait_any** call.

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

An *ArrivalTest* originates from a sequence (one or more) of **dr_test** calls whose input request variable was initialized by a **dr_msg_rcv_i** call and does retrieve a message from the runtime buffers. It is defined as tuple $ArrivalTest \in ReqInitOrder \times Count \times Data$, such that *ReqInitOrder* is the initialization order of the input request variable at the current node and *Count* is the number of times the **dr_test** call had failed, before succeeding and retrieving a message.

Similarly, $NonArrivalTest \in ReqInitOrder \times Count$ record stems from a sequence of **dr_test** calls. However, the *NonArrivalTest* 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 **dr_msg_rcv_i** or when the input request variable was initialized by a **dr_msg_rcv_i** call and the sequence of **dr_test** calls doesn't retrieve a message from the runtime buffers. Table 1 shows a trace of the example program in Fig. 8:

Table 1. Data-replay trace of the program in Fig. 8

Node 2	Node 4	Node 5
A:1/0/D1	R:1/1/D1	R:1/1/D1
W:2/D2	T:1/10/D2	R:1/2/D2
	W:2/D3	R:1/3/D3

Since nodes 1 and 3 don't receive any messages, they don't produce traces. The trace record (A:1/0/D1) is described as follows: 'A' indicates a *RecvWany* record. Number 1 is the invocation order of the **dr_wait_any** call that generated *RecvWany* record. Number 0 is the value that

dr_wait_any returned. D1 signifies the payload of the retrieved message. The fields of the trace record (W:2/D1) are described as follows: ‘W’ indicates a *Wait* record. Number 2 is the initialization order of the **dr_wait** input request variable. D2 signifies the payload of the retrieved message.

The fields of the trace record (R:1/1/D1) are described as follows: ‘R’ indicates a *Recv* record. The first number 1 is the port number of the receiving endpoint. The second number 1 is the invocation order of the **dr_msg_recv** call that generated this *Recv* record. D1 signifies the payload of the received message.

The fields of the trace record (T:1/10/D2) are described as follows: ‘T’ indicates an *ArrivalTest* record. Number 1 is the initialization order of the **dr_test** input request variable. Number 10 is the number of failed **dr_test** calls. D2 signifies the payload of retrieved message.

3.2.2 The Replay Mechanism

We now describe how a trace is used to replay an execution. To achieve a correct and deterministic replay of a program, it is necessary to associate endpoints, request variables and certain calls that were observed during the recording mode with their counterparts in the replay mode. An endpoint that is observed in the replay mode is associated with an endpoint that is observed in the recording mode via the node identifier and the port number; both remain the same across executions.

Request variables are tracked across an execution in the recording mode and an execution in the replay mode using their order of initialization in a node. Similarly, **dr_msg_recv** and **dr_wait_any** calls are tracked by their invocation order with respect to other **dr_msg_recv** and **dr_wait_any** calls, respectively, in the same node.

During replay, DR-MCAPI maintains two data structures for each node: 1) *Records*: a list of trace records (i.e. *Recv*, *Wait*, *RecvTest*, *RecvWany*, *ArrivalTest*, *NonArrivalTest* and *NonRecvWany*). This list is constructed directly from the trace. 2) *RequestVariables*: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly. 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 was initialized by a non-blocking receive call, then we keep track of the receiving endpoint and the destination buffer pointer. If the trace indicates that **dr_test** calls were used to check the status of this request in the recording 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. All MCAPI routine calls that introduce non-determinism are handled by the DR-MCAPI library, rather than the MCAPI library.

The algorithm in Fig. 9 shows how **dr_msg_recv** calls are handled during replay. First, **RecvCalls** is incremented. **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**. Finally, the message payload in the *Recv* record is copied to the program buffer (lines 4-5). The fact that the **GetRecvRecord** call determines which message will be delivered next is what allows **dr_msg_recv** to be deterministic.

```
dr_msg_recv(Endpoint, &Buffer){
1  RecvCalls++;
2  PortNum=GetPortNumber(Endpoint);
3  RecvRecord=GetRecvRecord(RecvCalls);
4  Data=RecvRecord.Data;
5  copy(Buffer,&Data);
6  return;
}
```

Fig. 9. Handling dr_msg_recv calls

The algorithm in Fig. 10 shows how DR-MCAPI handles **dr_wait** calls. 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 initialization order and a pointer to the program buffer of this request are retrieved (lines 4-5).

Next, the **GetWaitRecord** procedure looks up the *Records* list to fetch the *Wait* record with *ReqInitOrder=Order*. Finally, the message payload in the *Wait* record is copied to the program buffer (lines 7-8).

```

dr_wait(Request) {
1  if not IsRecvRequest(Request) then
2    return mcapi_wait(Request);
3  end-if
4  Order=GetOrder(Request);
5  DataPtr=GetDataPtr(Request);
6  WaitRecord=GetWaitRecord(Order);
7  Data=WaitRecord.Data;
8  copy(DataPtr,&Data);
9  return;
}

```

Fig. 10. Handling dr_wait calls

Fig. 11 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_wait_any** call doesn't retrieve a message, then the **GetNonRecvWaitanyRecord** procedure looks up the *Records* list to fetch the *NonRecvWany* record with *WanyOrder=WaitanyCalls* (line 3). In line 4, the *Index* in the *NonRecvWany* is retrieved and the request in the **Requests** array at *Index* will be forwarded to the MCAPI library (line 5). Finally, *Index* is returned to the program (line 6).

If the current **dr_wait_any** call does retrieve a message, then the **GetRecvWaitanyRecord** procedure looks up the *Records* list to fetch the *RecvWany* record with *WanyOrder=WaitanyCalls* (line 8). In line 9, the *Index* in the *RecvWany* is used to retrieve the program data pointer associated with the request in the **Requests** array at *Index*. Finally, the message data in the *RecvWany* record is copied to the program buffer (lines 11-12) and *Index* is returned to the program (line 13).

```

dr_wait_any(Requests, RsCount){
1  WaitanyCalls++;
2  if not RecvWany(WaitanyCalls) then
3    NonRecvWanyRecord=GetNonRecvWaitanyRecord(WaitanyCalls);
4    Index=NonRecvWanyRecord.Index;
5    mcapi_wait(Requests[Index]);
6    return Index;
7  else
8    RecvWanyRecord=GetRecvWaitanyRecord(WaitanyCalls);
9    Index=RecvWanyRecord.Index;
10   DataPtr=GetDataPtr(Requests[Index]);
11   Data=RecvWanyRecord.Data;
12   copy(DataPtr,&Data);
13   return Index;
14 end-if
}

```

Fig. 11. Handling dr_wait_any calls

A **dr_test** call is handled by the algorithm in Fig. 12. First, the initialization order of the input request variable (**Request**) is retrieved (line 1). If that request variable is associated with a *NonArrivalTest* record, then the *Count* of this record is reduced by one (line 4). If *Count* reaches zero, the request is forwarded to the MCAPI runtime and true is returned to the program (lines 8-9). If that request variable is associated with a *ArrivalTest* record, then the *Count* of this record is reduced by one (line 13). If *Count* reaches zero, the request is passed to **dr_wait** and true is returned to the program (lines 17-18). As long as the value of *Count* is greater than zero, false is returned.

```

bool dr_test(Request){
1  Order=GetOrder(Request);
2  if not ArrivalTest(Order) then
3    NonArrivalTest=GetNonArrivalTest(Order);
4    NonArrivalTest.Count--;
5    if NonArrivalTest.Count>0 then
6      return false;
7    else
8      mcapi_wait(Request);
9      return true;
10   end-if
11  else
12    ArrivalTest=GetArrivalTest(Order);
13    ArrivalTest.Count--;
14    if ArrivalTest.Count>0 then
15      return false;
16    else
17      dr_wait(Request);
18      return true;
19    end-if
20  end-if
}

```

Fig. 12. Handling dr_test calls

3.3 DR-MCAPI Order-replay

In order-replay, the outcomes of non-deterministic operations are recorded during a recording execution and are enforced during the replay execution. All nodes must be running during replay. Since in order-replay only the outcomes of non-deterministic operations are recorded, far less data than data-replay is recorded. We have developed two techniques for realizing order-replay: sender-based order-replay (described in section 3.3.1) and receiver-based order-replay (described in section 3.3.2).

3.3.1 Sender-based Order-replay

Sender-based order-replay works by capturing the total order of messages arrival during the recorded execution and enforcing this order during the replay execution by changing the order of dispatching **dr_msg_send** (and **dr_msg_send_i**) calls to the MCAPI runtime. First, we describe the sender-based order-replay trace structure in section 3.3.1.1 and then its replay mechanism in section 3.3.1.2.

3.3.1.1 The Trace Structure

In the sender-based order-replay technique, a single trace is generated for the whole program: $Trace = \{Record^1, \dots, Record^{|Trace|}\}$ and there are three record types in the trace:

$$Record \in Send \cup Wany \cup Test$$

A *Send* record represents sending a message between two endpoints and is defined as tuple: $Send \in Node \times Port \times SendOrder \times UAO$. *Node* is the identifier of the sending node. *Port* is the port number of the sending endpoint. *SendOrder* is the invocation order of the particular **dr_msg_send** (or **dr_msg_send_i**) call among other **dr_msg_send** (and **dr_msg_send_i**) calls at the same node. *UAO* stands for Unique Order of Arrival which is a global number assigned to every received message and it establishes a total order of arrivals among all received messages in a program. A *Send* record is constructed in two steps:

- 1) When a **dr_msg_send** (or a **dr_msg_send_i**) is invoked, the message payload is augmented with the triple $\{Node, Port, SendOrder\}$.
- 2) When a message is retrieved from the runtime buffers (by **dr_msg_recv**, **dr_wait**, **dr_wait_any** or **dr_test** call), it is assigned the *UAO* number. *UAO* is monotonically increasing with every received message throughout the program.

A *Wany* record stems from a **dr_wait_any** call and is defined as $Wany \in Node \times Order \times Index$. *Node* is the identifier of the current node. *Order* is the invocation order of this particular **dr_wait_any** call among other **dr_wait_any** calls at the same node. *Index* is the index returned by the **dr_wait_any** call.

A *Test* record originates from a sequence (one or more) of **dr_test** calls and is defined as $Test \in Node \times ReqInitOrder \times Count$, such that *Node* is the identifier of the current node, *ReqInitOrder* is the initialization order of the input request variable at the current node and *Count* is the number of times the **dr_test** call had failed. Table 2 shows a trace of the example program in Fig. 8.

Table 2. Sender-based order-replay trace of the program in Fig. 8

Node 1	Node 2	Node 3	Node 4
S:1/1/1/1	A:2/1/0	S:3/1/1/2	T:4/1/10
S:1/1/2/3	S:2/1/1/5	S:3/1/2/4	
S:1/1/3/7	S:2/1/2/6	S:3/1/3/8	

Since node 5 neither sends messages, nor has **dr_test** or **dr_wait_any** calls, it doesn't contribute to the trace. The trace record (S:1/1/1/1) was generated by the **dr_msg_send** call in line 7 and the **dr_wait_any** call at line 29. Its fields are described as follows: 'S' indicates a *Send* record. The first number 1 is the node identifier. The second number 1 is the port number of the sending endpoint. The third number 1 is the invocation order of the **dr_msg_send** call. The fourth number 1 is the UAO which indicates that the message sent by this **dr_msg_send** call was the first to be received in the recorded execution.

The trace record (A:2/1/0) was generated by the **dr_wait_any** call at line 29. Its fields are described as follows: 'A' indicates a *Wany* record. Number 2 is the node identifier. Number 1 is the invocation order of this **dr_wait_any** call. Number 0 is the value returned by the **dr_wait_any** call.

The trace record (T:4/1/10) was generated by the **dr_test** call in line 44 and its fields are described as follows: 'T' indicates a *Test* record. Number 4 is the node identifier. Number 1 is the **dr_test** call input request variable initialization order. Number 10 is the number of times the **dr_test** call was invoked.

3.3.1.2 The Replay Mechanism

We now describe how a trace is used to replay an execution. When a program is run in the replay mode, four data structures are created:

- 1) *SendRecords*: a list of all *Send* records from the trace.
- 2) *RequestVariables*: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly.
- 3) *WanyRecords*: a list of *Wany* records that are obtained from the trace
- 4) *TestRecords*: a list of *Test* records that are retrieved from the trace.

In the replay mode, sending a message is a three step process as depicted in Fig. 13:

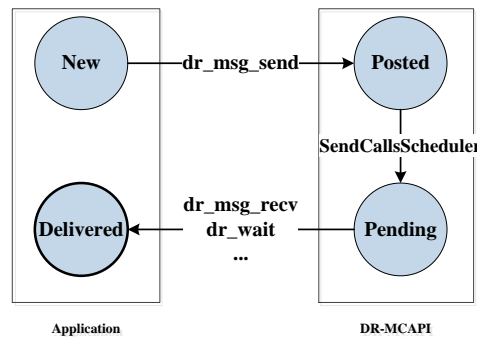


Fig. 13. The three steps of sending a message

- 1) When a **dr_msg_send** (or a **dr_msg_send_i**) is invoked by the program, the corresponding *Send* record in the *SendRecords* list is set to *Posted*.
- 2) The algorithm *SendCallsScheduler* in Fig. 14 continuously monitors the *SendRecords* list. If *SendCallsScheduler* finds a *Send* record whose state is *Posted* and whose *UAO* equals to **LatestUAO**, this record state is set to *Pending* and a corresponding **dr_msg_send** (or **dr_msg_send_i**) is invoked to actually send a message.
- 3) When a message is received (via a call to **dr_msg_rcv**, **dr_wait**, **dr_test**, or a **dr_wait_any**), its state is set to *Delivered*.

```

SendCallsScheduler(){
1  LatestUAO=0;
2  Max=Size(SendRecords);
3  while(LatestUAO<Max) do
4    if (there are no pending send calls) then
5      S = SendRecords.GetRecord(UAO);
6      if (S is posted)
7        LatestUAO++;
8        Set S state to pending;
9        Forward S to the MCAPI runtime;
10     end-if
11   end-if
12 end-while
}

```

Fig. 14. The SendCallsScheduler Algorithm

Fig. 15 shows how **dr_wait_any** calls are handled during replay. First, **WaitanyCalls** is incremented. **WaitanyCalls** keeps track of the number of **dr_wait_any** function invocations at the node. Second, the **GetWaitanyRecord** procedure looks up the *WanyRecords* list to fetch the *Wany* record with *Order*=**WaitanyCalls** (line 2). In line 3, the *Index* in the *Wany* is retrieved and the request in the **Requests** array at *Index* will be forwarded to the MCAPI library (line 4). Finally, *Index* is returned to the program.

```

dr_wait_any(Requests, RsCount){
1  WaitanyCalls++;
2  WanyRecord=GetWaitanyRecord(WaitanyCalls);
3  Index=WanyRecord.Index;
4  mcapi_wait(Requests[Index]);
5  return Index;
}

```

Fig. 15. Handling dr_wait_any calls

A **dr_test** call is handled by the algorithm in Fig. 16. First, the initialization order of the input request variable (**Request**) is retrieved (line 1). Then, the **Test** record associated with this initialization order is retrieved from the *TestRecords* list (line 2). Next, the *Count* of this record is reduced by one (line 3). When *Count* reaches zero, the request is passed to **mcapi_wait** and true is returned to the program (lines 7-8).

```

bool dr_test(Request){
1  Order=GetOrder(Request);
2  Test=GetTest(Order);
3  Test.Count--;
4  if Test.Count>0 then
5    return false;
6  else
7    mcapi_wait(Request);
8    return true;
9  end-if
}

```

Fig. 16. Handling dr_test calls

We use Table 3 and Table 4 to illustrate how the sender-based order-replay works. Table 3 shows a list of the `dr_msg_send` calls that appear in the example program and has three columns. The Line column shows the line numbers of the `dr_msg_send` calls. The Name column assigns names to the `dr_msg_send` calls. We use these names for brevity. The UAO column lists the UAO associated with the `dr_msg_send` calls according to the trace in Table 2.

Table 3. Sender-based order-replay example

Line	Name	UAO
7	S1	1
8	S2	3
9	S3	7
18	S4	2
19	S5	4
20	S6	8
34	S7	5
35	S8	6

Now, it is possible that in an execution the `dr_msg_send` calls are invoked according to this order: S4, S5, S1, S2, S3, S6, S7, and finally S8. Table 4 describes how the DR-MCAPI library will handle the `dr_msg_send` calls such that the messages order of arrival that was observed in the recording mode will be exhibited during the replay mode.

Table 4. Handling `dr_msg_send` example

Program Event	DR-MCAPI Library Action
S4 is invoked	<ol style="list-style-type: none"> 1) S4 <i>Send</i> record is set to <i>Posted</i>. 2) S4 will be blocked by the SendCallsScheduler procedure, since there are other <code>dr_msg_send</code> calls with smaller UAO (S1) that were not delivered yet.
S5 is invoked	<ol style="list-style-type: none"> 1) S5 <i>Send</i> record is set to <i>Posted</i>. 2) S5 will be blocked by the SendCallsScheduler procedure, since there are other <code>dr_msg_send</code> calls with smaller UAO (S1, S2, and S4) that were not delivered yet.
S1 is invoked	<ol style="list-style-type: none"> 1) S1 <i>Send</i> record is set to <i>Pending</i>. 2) S1 message is forwarded to the MCAPI runtime.
S1 message is received	<ol style="list-style-type: none"> 1) S1 <i>Send</i> record is set to <i>Delivered</i>. 2) S4 <i>Send</i> record is set to <i>Pending</i>. 3) S4 message is forwarded to the MCAPI runtime.
S4 message is received	<ol style="list-style-type: none"> 1) S4 <i>Send</i> record is set to <i>Delivered</i>. 2) S5 is still blocked since S2 (which has a smaller UAO) is not delivered yet.
S2 is invoked	<ol style="list-style-type: none"> 1) S2 <i>Send</i> record is set to <i>Posted</i>. 2) S2 message is forwarded to the MCAPI runtime.
S2 message is received	<ol style="list-style-type: none"> 1) S2 <i>Send</i> record is set to <i>Delivered</i>. 2) S5 <i>Send</i> record is set to <i>Pending</i>. 3) S5 message is forwarded to the MCAPI runtime.
S5 message is received.	<ol style="list-style-type: none"> 1) S5 <i>Send</i> record is set to <i>Delivered</i>.
S3 is invoked	<ol style="list-style-type: none"> 1) S3 <i>Send</i> record is set to <i>Posted</i>. 2) S3 will be blocked by the SendCallsScheduler procedure, since there are other <code>dr_msg_send</code> calls with smaller UAO (S7 and S8) that were not delivered yet.
S6 is invoked	<ol style="list-style-type: none"> 1) S6 <i>Send</i> record is set to <i>Posted</i>. 2) S6 will be blocked by the SendCallsScheduler procedure, since there are other <code>dr_msg_send</code> calls with smaller UAO (S3 and S8) that were not delivered yet.

S7 is invoked	1) S7 <i>Send</i> record is set to <i>Pending</i> . 2) S7 message is forwarded to the MCAPI runtime.
S7 message is received	1) S7 <i>Send</i> record is set to <i>Delivered</i> .
S8 is invoked	1) S8 <i>Send</i> record is set to <i>Posted</i> . 2) S8 message is forwarded to the MCAPI runtime.
S8 message is received	1) S8 <i>Send</i> record is set to <i>Delivered</i> . 2) S8 <i>Send</i> record is set to <i>Pending</i> . 3) S3 message is forwarded to the MCAPI runtime.
S3 message is received	1) S3 <i>Send</i> record is set to <i>Delivered</i> . 2) S6 <i>Send</i> record is set to <i>Pending</i> . 3) S6 message is forwarded to the MCAPI runtime.
S6 message is received	1) S6 <i>Send</i> record is set to <i>Delivered</i> .

Table 4 shows that regardless to the order of **dr_msg_send** invocations observed in the replay executions, messages will be delivered according to the order observed in the recorded execution.

3.3.2 Receiver-based Order-replay

Maintaining the UAO variable and the *SendRecords* list requires shared-memory and synchronizing access to it via the MRAPI API [10]. This leads to excessive time overhead as shown by the experiments in section 4.3. The receive-based order-replay described in this section requires less time overhead than sender-based order-replay. Receiver-based order-replay works by capturing the order of messages arrival at a specific node during the recording phase and enforcing this order during the replay phase by manipulating the order of the messages retrieved from the runtime buffers. The order of messages arrival is established using a hash-code of the message's payload.

First, we describe the receiver-based order-replay trace structure in section 3.3.2.1 and then its replay mechanism in section 3.3.2.2.

3.3.2.1 The Trace Structure

When an instrumented program P is run in the recording mode, a separate trace is generated for each MCAPI node:

$$Trace_P = \{Trace^1, \dots, Trace^N\}, \text{ where } N \text{ is the number of nodes in program } P.$$

A node's trace contains a list of records:

$$Trace^n = \{Record^1, \dots, Record^{|Trace^n|}\}, \text{ where } n \text{ is a node identifier.}$$

There are six types of records:

$$Record \in Recv \cup Wait \cup RecvWany \cup NonRecvWany \cup ArrivalTest \cup NonArrivalTest$$

A *Recv* record originates from a **dr_msg_recv** call and is defined as 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 **dr_msg_recv** call among other **dr_msg_recv** calls with the same endpoint. *Hash* is a hash-code of the received message data and is calculated using the CRC-32 algorithm [23].

A *Wait* record originates from a **dr_wait** call whose input request variable was initialized by a **dr_msg_recv_i** call and is defined as $Wait \in ReqInitOrder \times Hash$. *ReqInitOrder* is the initialization order of the input request variable at the current node.

A *RecvWany* record comes from a **dr_wait_any** call that returned the index of a request variable that was initialized by a **dr_msg_recv_i** call and is defined as: $RecvWany \in WanyOrder \times Index \times Hash$. *WanyOrder* is the invocation order of this particular **dr_wait_any** call among other **dr_wait_any** calls at this node. *Index* is the index returned by the **dr_wait_any** call. *Hash* is a hash-code of the received message data.

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

An *ArrivalTest* record originates from a sequence (one or more) of *test* calls whose input request variable was initialized by a **dr_msg_rcv_i** call and does retrieve a message from the runtime buffers. It is defined as $ArrivalTest \in ReqInitOrder \times Count \times Hash$, such that *ReqInitOrder* is the initialization order of the input request variable at the current node, *Hash* is a hash-code of the received message data and *Count* is the number of times the **dr_test** call had failed, before succeeding and retrieving a message.

Similarly, *NonArrivalTest* record stems from a sequence of **dr_test** calls. However, the *NonArrivalTest* 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 **dr_msg_rcv_i** or when the input request variable was initialized by a **dr_msg_rcv_i** call and the sequence of **dr_test** calls doesn't retrieve a message from the runtime buffers.

Table 5. Receive-based order-replay trace

Node 2	Node 4	Node 5
A:1/0/C1	R:1/1/C1	R:1/1/C1
W:2/C2	T:1/10/C2	R:1/2/C2
	W:2/C3	R:1/3/C3

Since nodes 1 and 3 don't receive any messages, they don't produce traces. The trace record (A:1/0/C1) is described as follows: 'A' indicates a *RecvWany* record. Number 1 is the invocation order of the **dr_wait_any** call that generated *RecvWany* record. Number 0 is the value that **dr_wait_any** returned. C1 signifies the hash-code of the payload of the retrieved message.

The fields of the trace record (W:2/C1) are described as follows: 'W' indicates a *Wait* record. Number 2 is the initialization order of the **dr_wait** input request variable. C2 signifies the hash-code of the payload of the retrieved message.

The fields of the trace record (R:1/1/C1) are described as follows: 'R' indicates a *Recv* record. The first number 1 is the port number of the receiving endpoint. The second number 1 is the invocation order of the **dr_msg_rcv** call that generated this *Recv* record. C1 signifies the hash-code of the payload of the retrieved message.

The fields of the trace record (T:1/10/C2) are described as follows: 'T' indicates an *ArrivalTest* record. Number 1 is the initialization order of the **dr_test** input request variable. Number 10 is the number of failed **dr_test** calls. C2 signifies the hash-code of the payload of the retrieved message.

3.3.2.2 The Replay Mechanism

We now describe how a trace is used to replay an execution. To support the replay mode, we maintain three data structures:

- 1) *Records*: a list of records (e.g. *Recv*, *Wait*...) that are retrieved from the trace.
- 2) *RequestVariables*: a list of request variables per node. This list combines data from the trace and data that are obtained on-the-fly.
- 3) *RecievedMessages*: messages that arrive earlier than expected are stored in this list along with their hash-codes.

The algorithm in Fig. 17 handles **dr_msg_rcv** calls. First **RecvCalls** is incremented (line 1). **RecvCalls** keeps track of the number of **dr_msg_rcv** 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). Otherwise, the **mcapi_msg_rcv** is repeatedly invoked until it retrieves a message whose hash-code matches the expected hash-code (lines 10-20). When the expected message arrives, it is copied to the program buffer (line 15). All other messages and their hash-codes are appended to *RecievedMessages* (line 19).

```

dr_msg_rcv(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              return;
9          end-if
10 while(true) do
11     mcapi_msg_rcv(Endpoint,&TempBuffer);
12     ArrivedCRC=CalculateCRC(TempBuffer);
13     if (ArrivedCRC==ExpectedCRC)
14         then
15             copy(Buffer, TempBuffer);
16             return;
17         end-if
18     else
19         RecievedMessages.Append(TempBuffer, ArrivedCRC);
20 end-while
}

```

Fig. 17. Handling `dr_msg_rcv` calls

In the program in Fig. 8, node 5 receives two messages. Let's assume that when running that program in the recording mode, it generates the trace in Table 5 (i.e. the order of messages arrival is C1, C2, and then C3) and that during running the program in the replay mode, the messages arrive with a different order: C2, C3, and then C1.

When `dr_msg_rcv` is invoked for the first time, the *RecievedMessages* list will be empty. Hence, the while loop (lines 10-20) will iterate thrice. In the first iteration, the `mcapi_msg_rcv` call will retrieve the message with hash-code C2. Since the retrieved message is not the expected one, it will be added to the *RecievedMessages* list (line 19).

In the second iteration, the `mcapi_msg_rcv` call will retrieve the message with hash-code C3 and, it will be added to the *RecievedMessages* list as well. In the third iteration, the message with hash-code C1 will be retrieved. So, this message will be delivered to the program (line 15). When `dr_msg_rcv` is invoked for the second and third times, the *RecievedMessages* list will contain the expected messages and they will be returned to the program in the correct order (lines 4-9).

Fig. 18 shows the algorithm that handles a `dr_wait` call whose input request variable was initialized by a `msg_rcv_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_rcv_i` call.

First, if the input request was not initialized by a `dr_msg_rcv_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).

Fig. 19 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_wait_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).

```

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       return;
23   end-if
24 end-for
}

```

Fig. 18. Handling dr_wait calls

If the current **dr_wait_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 program (line 11).

```

dr_wait_any(Requests, RsCount) {
1  WaitanyCalls++;
2  if RecvWany(WaitanyCalls) then
3    RecvWanyRecrd=GetRecvWanyRecord(WaitanyCalls);
4    Index=RecvWanyRecrd.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;
}

```

Fig. 19. Handling dr_wait_any calls

In the program in Fig. 8 , node 2 receives two messages. Let's assume that when running that program in the recording mode, it generates the trace in the Table 5 (i.e. the order of messages arrival is C1 then C2 and that **wait_any** call returns 0) and that during running the program in the replay mode, the messages arrive with a different order: C2 then C1. When **dr_wait_any** is invoked, it is going to determine that the request at index 0 of the array *Requests* was initialized by a **dr_msg_recv** call and will forward this request to **dr_wait**. In **dr_wait**, the first loop (lines 10-17) will retrieve the two messages via three calls to **mcapi_wait** (line 12) and they will be added to the *RecievedMessages* list (line 15). The second loop (lines 18-24) will iterate through the *RecievedMessages* list and will return the message with hash-code C1 to the program. When

dr_wait is invoked to handle the **wait** call at line 30 in Figure 10, the message with hash-code C2 will be already in the *RecievedMessages* list and will be returned to the program.

A **dr_test** call is handled by the algorithm in Fig. 20. 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 program (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 program (lines 17-18).

```

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
}

```

Fig. 20. Handling dr_test calls

In the program in Fig. 8, node 4 receives three messages. Let's assume that when run in the recording mode, this program generates the trace in Table 5 (i.e. three messages are retrieved with order: C1, C2, and then C3 and that the **dr_test** call at line 44 retrieves the messages with hash-code C2 at the 11th invocation) . Let's assume that during replay, the messages arrive with a different order (C2, C1, and then C3). When the **dr_msg_rcv** call at line 43 is invoked, messages with hash-codes C1 and C2 will be retrieved from the runtime buffers and C1 will be returned to the program. When the **dr_test** call at line 43 is invoked, it will return *false* for 10 times and at the 11th invocation, it will invoke **dr_wait**. **dr_wait** will find the message with hash-code C2 in the *RecievedMessages* list. When the **dr_wait** call at line 45 is invoked, it will retrieve the message with hash-code C3.

4. EXPERIMENTAL RESULTS

In this section we analyze the performance of the replay approaches in terms of trace size, memory usage and runtime overheads.

4.1 Methodology

We performed experiments on three sets of MCAPI programs developed by ourselves (BT, CG and TN) and a set of programs (Bully) obtained from an external source [16]. 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 (BT): 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 (CG): This is a set of 10 programs that create networks of nodes with increasing sizes from 2 nodes to 11 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 graph) and 1100 messages (for a 11 nodes graph). This benchmark has an all-to-all communication pattern.
 - 3) 10-nodes benchmark (TN): This is a set of 10 programs that create networks of nodes with a fixed size of 10 nodes. However, the number of messages exchanged among the nodes increases monotonically. The number of messages exchanged is between 90 and 900. This benchmark allows us to isolate the effect of the number of messages on performance.
 - 4) Bully benchmark (Bully): This is a set of 10 programs that create of networks of nodes with different sizes and use the Bully leader selection algorithm [9] to select a leader node. The number of exchanged messages is between 35 messages (for a 3 nodes network) and 314 messages (for a 12 nodes network). This benchmark was provided by the V&V research group at Brigham Young University.

In all benchmarks, except the Bully benchmark, the message size is 50 bytes. The Bully benchmark message size is 4 bytes.

To analyze the runtime and memory usage, a given program is executed three times: 1) without the DR-Library, 2) with the DR-MCAPI library in recording mode and 3) with the DR-MCAPI library in replay mode. These three executions are repeated forty times and the results are averaged. We use a pair of `gettimeofday` function calls; when a program starts execution and when it ends execution to calculate total runtime and use the Massif [18] heap profiler to measure the heap memory used by a given execution. For the sake of brevity, we refer to data-replay as D-Replay, sender-based order-replay as S-Replay, and receiver-based order-replay as R-Replay.

4.2 Log size

Our first analysis is for the trace size. Fig. 21 shows the trace size relative to the number of exchanged messages using the D-Replay, S-Replay and R-Replay techniques. The x-axis is the number of messages and the y-axis is the trace size in kilobytes.

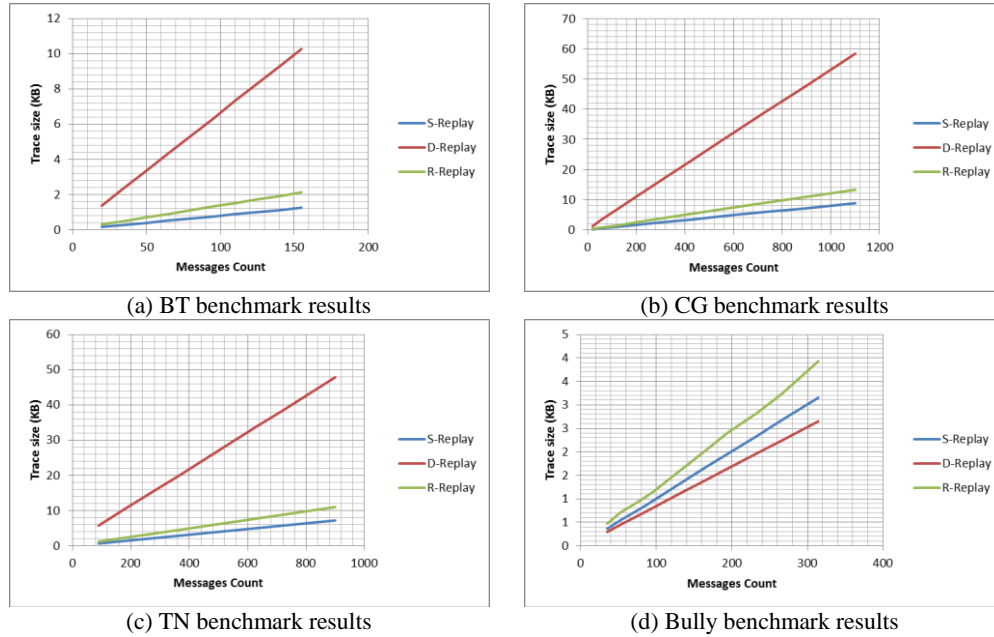


Fig. 21. Comparing the trace size among replay techniques

D-Replay produces a large trace compared to R-Replay and S-Replay in the benchmarks BT, CG, and TN. However, it is the opposite with the Bully benchmark. This is due to the small size of the messages exchanged in the Bully benchmark (4 bytes) compared to the other benchmarks (50 bytes). Table 6 shows the typical record size in the three replay techniques.

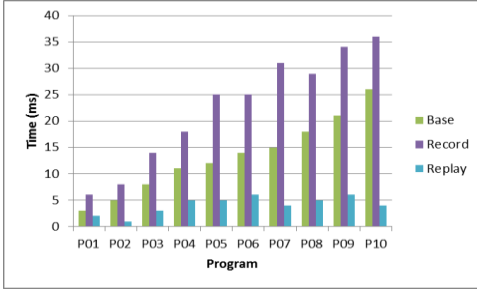
Table 6. Records structures and sizes in D-Replay, S-Replay and R-Replay

Technique	Typical Record Structure	Record Size (Bytes)
D-Replay	$Recv \in Port \times RecvOrder \times Data$	$1+4+sizeof(Data)$
S-Replay	$Send \in Node \times Port \times SendOrder \times UAO$	$1+1+4+4$
R-Replay	$Recv \in Port \times RecvOrder \times Hash$	$1+4+8$

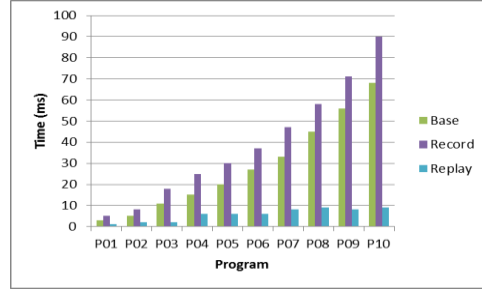
In D-Replay, the record size is 5 bytes plus the size of the message payload. Hence, in the Bully benchmark, the size of a trace record is 9 bytes, which is less than 13 bytes and 10 bytes for the S-Replay and R-Replay, respectively.

4.3 Runtime Overhead

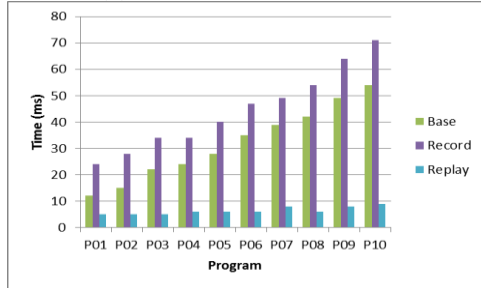
Fig. 22, Fig. 23 and Fig. 24 compare the running times of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the D-Replay, S-Replay and R-Replay techniques, respectively. The x-axis shows the program name and y-axis shows the time in milliseconds.



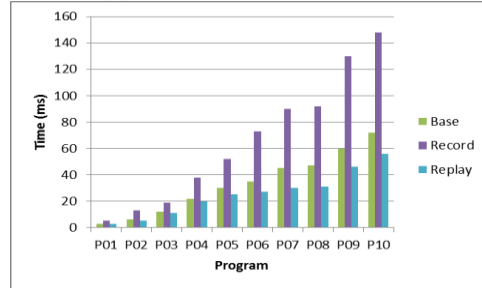
(a) BT benchmark results



(b) CG benchmark results



(c) TN benchmark results



(d) Bully benchmark results

Fig. 22. The runtime overhead in D-Replay

In D-Replay (Fig. 22), runtime overhead during recorded executions are 1.8x, 1.5x, 1.5x and 1.9x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.7x. However, replay executions runtime is *less* than baseline executions. This is due to two reasons: 1) during a replay execution, only one node is being replayed; 2) messages are not actually being sent but their arrival is being simulated. Hence, messages transfer time is eliminated.

In S-Replay (Fig. 23), runtime overhead during recorded executions are 2.6x, 3.3x, 4.2x and 2.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 3.0x. Runtime overhead during replay executions are 5.6x, 4.6x, 5.4x and 3.5x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 4.7x.

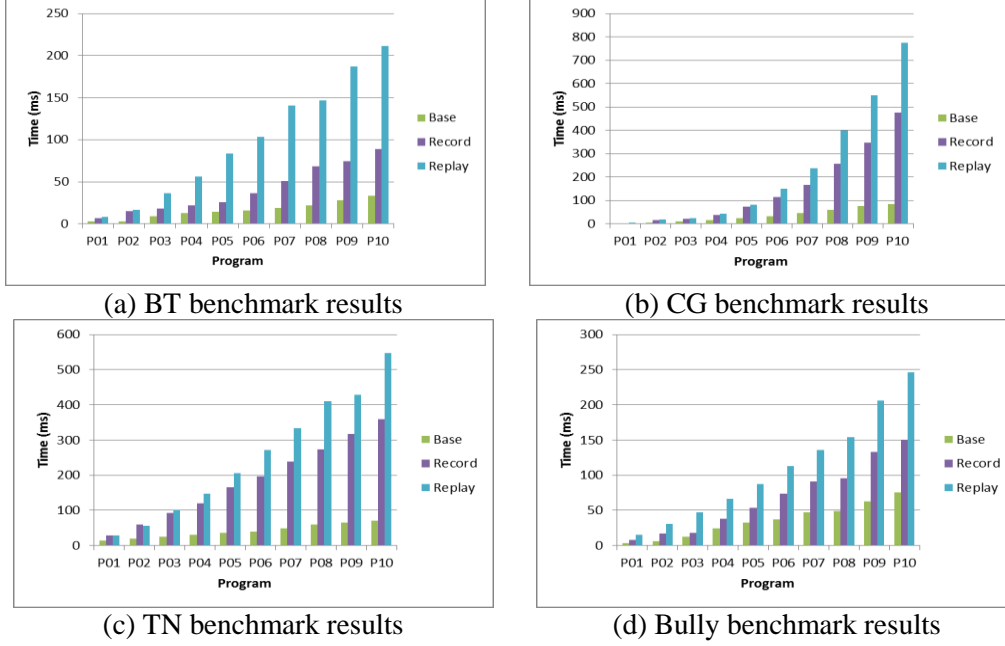


Fig. 23. The runtime overhead in S-Replay

In R-Replay (Fig. 24), runtime overhead during recorded executions are 1.5x, 1.3x, 1.3x and 1.4x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.4x. Runtime overhead during replay executions are 2.3x, 1.9x, 1.8x and 1.6x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 1.9x.

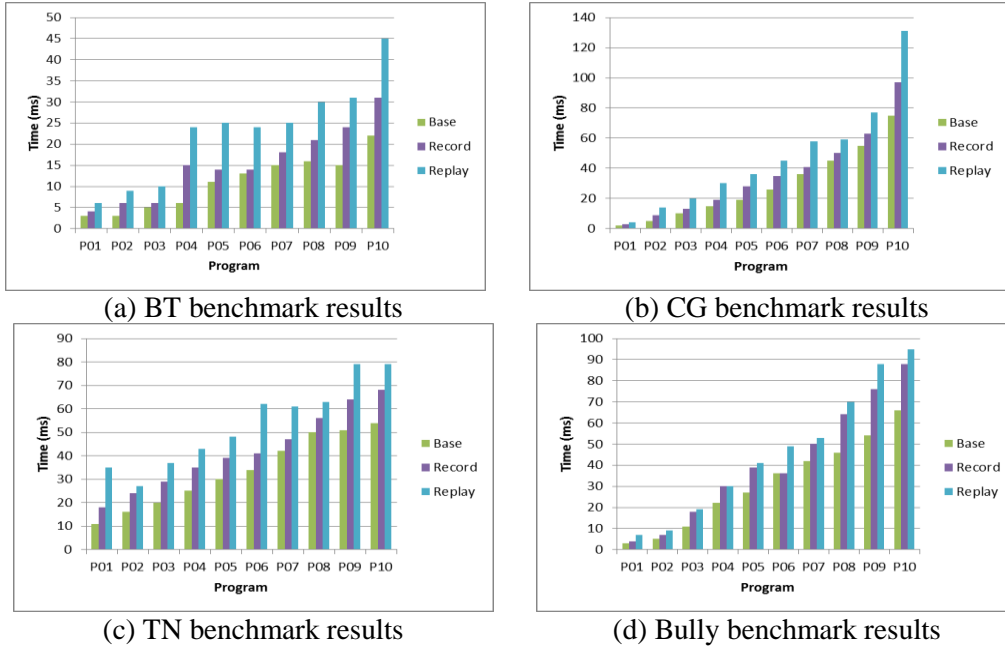
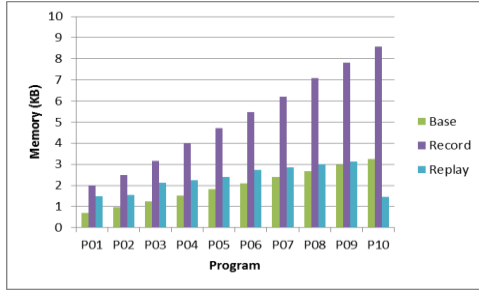


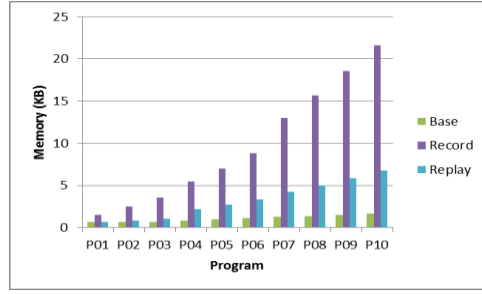
Fig. 24. The runtime overhead in R-Replay

4.4 Memory Usage Overhead

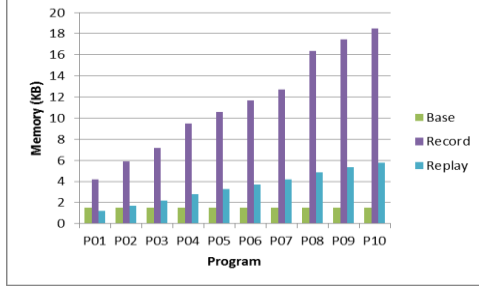
Fig. 25, Fig. 26 and Fig. 27 compare the memory usage of a baseline execution, a recorded execution and a replay execution for the four benchmarks when using the D-Replay, S-Replay and R-Replay techniques, respectively. The x-axis shows the program name and y-axis shows the memory usage in milliseconds.



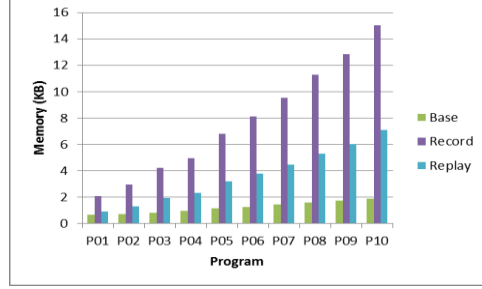
(a) BT benchmark results



(b) CG benchmark results



(c) TN benchmark results

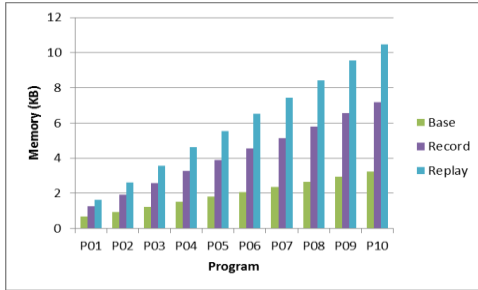


(d) Bully benchmark results

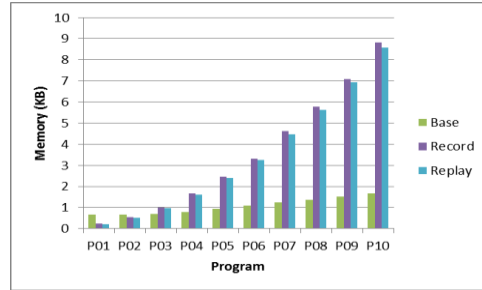
Fig. 25. The memory usage overhead in D-Replay

In D-Replay (Fig. 25), memory usage overhead during recorded executions are 2.6x, 8.0x, 7.5x and 6.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 6.0x. Memory usage overhead during replay executions are 1.3x, 2.7x, 2.3x and 2.7x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.2x.

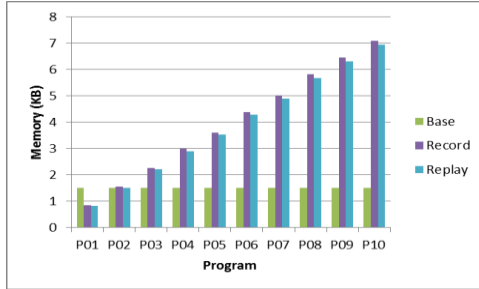
In S-Replay (Fig. 26), memory usage overhead during recorded executions are 2.1x, 2.8x, 2.6x and 2.0x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.4x. Memory usage overhead during replay executions are 3.0x, 2.8x, 2.6x and 2.6x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.7x.



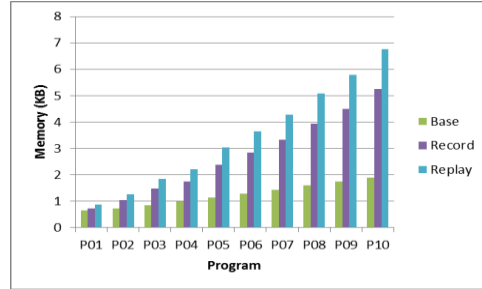
(a) BT benchmark results



(b) CG benchmark results



(c) TN benchmark results



(d) Bully benchmark results

Fig. 26. The memory usage overhead in S-Replay

In R-Replay (Fig. 27), Memory usage overhead during recorded executions are 2.7x, 2.9x, 3.0x and 2.8x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 2.8x. Memory usage overhead during replay executions are 2.7x, 3.1x, 3.2x and 2.9x in the BT, CG, TN and Bully benchmarks, respectively. The average runtime overhead is 3.0x.

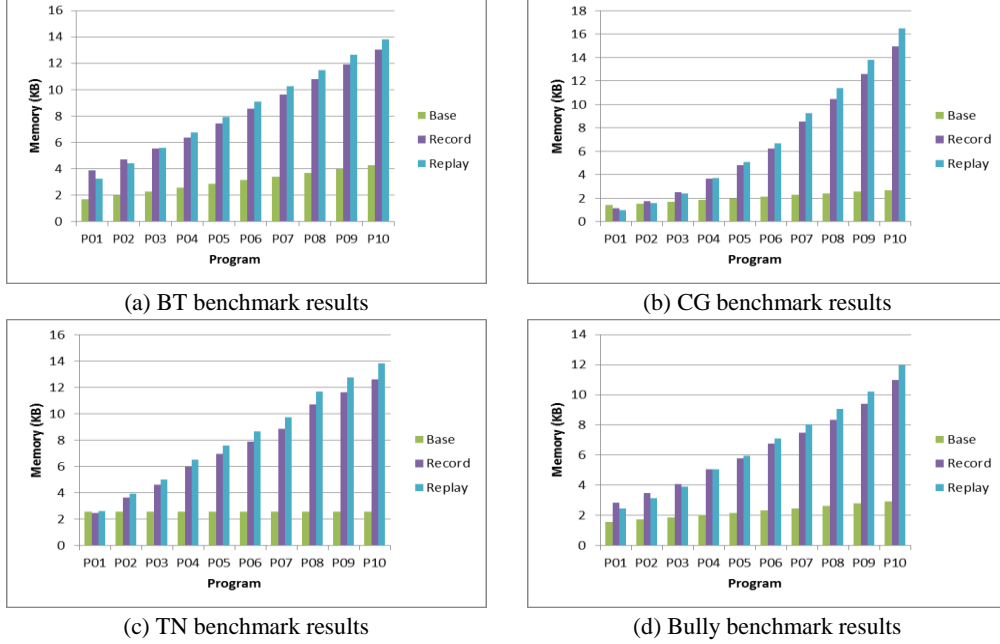


Fig. 27. The memory usage overhead in R-Replay

5. DISCUSSION

In this section we discuss the features of DR-MCAPI such as usability, portability and scalability.

5.1 Usability

DR-MCAPI is a push-button solution. The user needn't to change the source code or change/re-compile the MCAPI library. Using DR-MCAPI involves three steps: 1) instrumenting the source code, 2) compiling the instrumented program, and 3) running the generated executable. These steps are easily automated using a batch script or could be incorporated into the MCAPI compilation chain. DR-MCAPI handles program crashes arising from assertion failures. Assert calls are instrumented such that when the assertion expression evaluates to false during the recording phase, the logged trace is dumped to the disk before allowing the program to crash.

5.2 Portability

DR-MCAPI doesn't require hardware amendments and since it sits as layer between the program and the MCAPI library, it is portable across different implementations of the MCAPI specification. For example, DR-MCAPI is usable with the OpenMCAPI [13] implementation without any changes.

5.3 Scalability

Now, we discuss the scalability of DR-MCAPI in terms of the trace size, runtime and memory overheads. The trace size scales linearly with the number of messages exchanged. However, since in D-Replay the message payload itself is stored in the trace, the sizes of the messages affect the trace size resulting in larger traces. The S-Replay and R-Replay trace record sizes are independent of the message payload size.

A major factor affecting the scalability of DR-MCAPI is the time overhead, especially during a recorded execution. A replay execution is only needed when a failure takes place and the

developer needs to scrutinize the details. S-Replay encounters high runtime overhead during a recorded execution since all messages payloads are modified before being sent and are unpacked and processed after being received. On the other side, R-Replay has the least runtime overhead during recording since messages are accessed once (at the receiving node) to calculate the hash-code. S-Replay exhibits a very high runtime overhead during a replay execution since it manipulates the orders of executing `dr_msg_send` (and `dr_msg_send_i`) calls across the whole program and not within a node similar to R-Replay. D-Replay incurs the least replay time overhead because message transfer time is eliminated since messages are not actually being sent but their arrival is being simulated.

The memory overhead is due to the DR-MCAPI data structures. D-Replay memory overhead in the recording mode is the largest, since it buffers the contents of all messages exchanged until the trace is written to the disk. R-Replay requires more memory than S-Replay in the replay mode since it buffers messages received out of expected order.

5.4 Equivalent vs. Identical Replay

R-Replay guarantees an *equivalent* replay of the recorded execution; however S-Replay produces an *identical* replay. We use the program in Fig. 28 to demonstrate the difference.

<p>Node 1</p> <pre> 1 dr_initialize(1); 2 X=1; 3 EP=dr_create_endpoint(1); 4 N2EP=dr_get_endpoint(2,1); 5 N4EP=dr_get_endpoint(4,1); 6 dr_msg_send(EP,N2EP,X); 7 dr_msg_send(EP,N4EP,X); 8 dr_delete_endpoint(EP); 9 dr_finalize(); </pre>	<p>Node 3</p> <pre> 10 dr_initialize(3); 11 Y=10; 12 EP=dr_create_endpoint(1); 13 N2EP=dr_get_endpoint(2,1); 14 N4EP=dr_get_endpoint(4,1); 15 dr_msg_send(EP,N2EP,Y); 16 dr_msg_send(EP,N4EP,Y); 17 dr_delete_endpoint(EP); 18 dr_finalize(); </pre>
<p>Node 2</p> <pre> 19 dr_initialize(2); 20 EP=dr_create_endpoint(1); 21 dr_msg_rcv(EP,&A); 22 dr_msg_rcv(EP,&B); 23 dr_delete_endpoint(EP); 24 dr_finalize(); </pre>	<p>Node 4</p> <pre> 25 dr_initialize(4); 26 EP=dr_create_endpoint(1); 27 dr_msg_rcv(EP,&C); 28 dr_msg_rcv(EP,&D); 29 dr_delete_endpoint(EP); 30 dr_finalize(); </pre>

Fig. 28. Equivalent replay vs. identical replay

Let's assume that in a recording session of the program in Fig. 28, the order of arrival of the message was as in Table 7.

Table 7. An order of arrival in a recorded execution

Message	Destination Node	Arrival Order at destination node	Total Arrival Order
Sent at line 6	2	1	1
Sent at line 7	4	1	2
Sent at line 15	2	2	3
Sent at line 16	4	2	4

When using R-Replay, the *local* order of messages arrival at a given node during a replay execution is guaranteed to be the same as in the recorded execution. However, the *total* order of arrival of messages is not guaranteed to be the same. During an R-Replay replay session, it is possible to have the order of message arrival as in Table 8 which is equivalent to the one in Table 7, but *not* identical to it.

It is worth mentioning that D-Replay also produces an identical replay execution. S-Replay guarantees a replay session that adheres to both the local and total orders of messages arrival.

Table 8. Equivalent but not identical order of arrival

Message	Destination Node	Arrival Order at destination node	Total Arrival Order
Sent at line 6	2	1	2
Sent at line 7	4	1	1
Sent at line 15	2	2	4
Sent at line 16	4	2	3

5.5 D-Replay vs. S-Replay vs. R-Replay

In this section we compare the three replay techniques. As shown in Table 9, R-Replay exhibits better performance than S-Replay and D-Replay. D-Replay allows replaying some nodes (rather than all nodes) and in this case other nodes are simulated. This is useful as it speeds up the replay process, in case of many nodes. In order-replay, all processes *must* be replayed. The developer using DR-MCAPI should decide which nodes to replay. For example, if s/he suspects that a specific node is buggy. S-Replay is useful when identical replay is needed.

Table 9 D-Replay vs. S-Replay vs. R-Replay

Criteria	D-Replay	S-Replay	R-Replay
Trace Size	Worst	Good	Best
Recording time overhead	Good	Worst	Best
Replay time overhead	Best	Worst	Good
Recording memory overhead	Worst	Good	Good
Replay memory overhead	Best	Good	Worst
Replay specific nodes	Yes	No	No
Identical replay	Identical	Identical	Equivalent

5.6 The probe-effect

Due to logging, recorded executions may suffer from the probe effect [8] leading to changes in the relative timing of events. The probe effect may hide bugs that will otherwise be apparent. The probe effect can't be entirely avoided but it could be reduced by lessening the amount of data collected. DR-MCAPI is engineered to collect the minimum amount of data during the recording mode.

6. RELATED WORK

MPI [1] has been dominating message-passing software development for a long time. Hence, the current literature on replaying message-passing software is almost limited to MPI programs. In [11], Kranzlmueller 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 record 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 record phase. This approach is not applicable to MCAPI programs since their receive calls (msg_rcv and msg_rcv_i) do not specify a source parameter. Thus unlike MPI applications it is not possible to identify the source of a received message. Test operations are handled by counting the number of consecutive failing test operations associated with the same request variable during the record phase. In the replay phase, test operations are forced to fail (i.e. return false) until the recorded number of failed tests has reached. They report a 200% time increase during the record phase. Also, this approach is library-dependent (based on the MPICH library) which limits its portability.

In [15], 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 program 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 program posts a receive call; the data-replay layer returns the data recorded at the corresponding receive call during the record 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 [15] and [11] don't capture all forms of non-determinism in MPI programs, making it difficult to ensure a completely faithful replay.

The authors of [28] propose subgroup-reproducible replay (SRR) which combines order-replay and data-replay. During the record 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 [24] which implements an automated approach for verifying MCAPAPI programs. MCC creates a scheduling layer above the MCAPAPI runtime layer that allows intercepting MCAPAPI 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 Dynamic Partial Order Reduction (DPOR) [7] 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.

7. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

With the current trend of increasing a processor performance by adding more cores rather than increasing the clock speed, we may have processors with 10s or 100s cores in the near future. Currently, only a handful of applications can exploit the potentials of these multicore processors since only the very skilled programmers can develop applications for these processors. This must change. Every programmer should be able to write programs that take advantage of the multicore era processors. Hence, it is important to develop programming practices and tools that support multicore development. Providing a deterministic replay capability to multicore-specific standards such as MCAPAPI will greatly improve the debugging process. This is both an important and challenging problem. Any replay tool must be easy to use, scale well and handles all non-determinism sources in a program.

In this article, we presented DR-MCAPAPI. To the best of our knowledge, DR-MCAPAPI is the first replay tool that considers all non-determinism sources in MCAPAPI programs. The deterministic replay ability provided by DR-MCAPAPI allows a programmer to repeatedly execute the program under supervision of a debugger to catch flaws.

Currently, the trace scales linearly with the number of messages exchanged during the runtime of a program. Reducing the trace size will decrease both the time overhead and memory usage, hence improving the scalability our tool. We plan to investigate trace compression methods similar to the ones in [20] and [12] for DR-MCAPAPI traces. Check-pointing is a technique that allows recovery of a failed program to its state prior to failing [6]. Check-pointing works by periodically saving the state of a program to a stable storage during execution; when a failure takes place, the program is restarted from the last checkpoint [4]. We are exploring how to modify DR-MCAPAPI to support check-pointing for non-terminating MCAPAPI programs. Usability is of prime

importance to any tool. That is why are developing an Eclipse plugin that uses DR-MCAPI as a back-end to allow the user to perform interactive debugging.

APPENDIX

We summarize the trace records notations introduced in section 3 in Table 10.

Table 10 Trace records

Record	Structure	Source Call	Replay Technique
<i>Recv</i>	<i>Port</i> \times <i>RecvOrder</i> \times <i>Data</i>	<i>dr_msg_recv</i>	D-Replay R-Replay
<i>Wait</i>	<i>ReqInitOrder</i> \times <i>Data</i>	<i>dr_wait</i>	D-Replay R-Replay
<i>RecvWany</i>	<i>WanyOrder</i> \times <i>Index</i> \times <i>Data</i> .	<i>dr_wait_any</i>	D-Replay R-Replay
<i>NonRecvWany</i>	<i>WanyOrder</i> \times <i>Index</i>	<i>dr_wait_any</i>	D-Replay R-Replay
<i>ArrivalTest</i>	<i>ReqInitOrder</i> \times <i>Count</i> \times <i>Data</i>	<i>dr_test</i>	D-Replay R-Replay
<i>NonArrivalTest</i>	<i>ReqInitOrder</i> \times <i>Count</i>	<i>dr_test</i>	D-Replay R-Replay
<i>Send</i>	<i>Node</i> \times <i>Port</i> \times <i>SendOrder</i> \times <i>UAO</i>	<i>dr_msg_send</i> <i>dr_msg_send_i</i>	S-Replay
<i>Wany</i>	<i>Node</i> \times <i>Order</i> \times <i>Index</i>	<i>dr_wait_any</i>	S-Replay
<i>Test</i>	<i>Node</i> \times <i>ReqInitOrder</i> \times <i>Count</i>	<i>dr_test</i>	S-Replay

REFERENCES

- [1] MPI: A Message-Passing Interface Standard. <http://www.mpi-forum.org/docs/mpi-2.2/index.htm>. [Online; accessed 28-April-2011].
- [2] The Multicore Association. <http://www.multicore-association.org>. [Online; accessed 28-April-2011].
- [3] BREHMER, S. The Multicore Association Communications API. <http://www.multicore-association.org/workgroup/mcapi.php>, March 2010. [Online; accessed 28-April-2011].
- [4] CHEN, Y., PLANK, J. S., AND LI, K. CLIP: a checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1997), Supercomputing '97, ACM, pp. 1–11.
- [5] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), OSDI '02, ACM, pp. 211–224.
- [6] FERTRE, M., AND MORIN, C. Transparent Message-Passing Parallel Applications Checkpointing in Kerrighed. Research Report RR-5755, INRIA, 2005.
- [7] FLANAGAN, C., AND GODEFROID, P. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.* 40 (January 2005), 110–121.
- [8] GAIT, J. A probe effect in concurrent programs. *Softw. Pract. Exper.* 16 (March 1986), 225–233.
- [9] GARCIA-MOLINA, H. Elections in a distributed computing system. *IEEE Trans. Comput.* 31 (January 1982), 48–59.
- [10] HOLT, J., AND BREHMER, S. The Multicore Association Resource Management API. <http://www.multicore-association.org/workgroup/mrapi.php>, March 2011. [Online; accessed 28-April-2011].
- [11] KRANZLMÜLLER, D., SCHAUBSCHLÄGER, C., AND VOLKERT, J. 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, 2001), Springer-Verlag, pp. 192–200.

- [12] KRISHNAMOORTHY, S., AND AGARWAL, K. Scalable communication trace compression. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (Washington, DC, USA, 2010), CCGRID '10, IEEE Computer Society, pp. 408–417.
- [13] LEVY, M. Mentor Releases Open Source of MCAPI for Multicore. <http://www.multicore-association.org/press/030211.html>, March 2011. [Online; accessed 28-April-2011].
- [14] LUCCHETTI, D., REINHARDT, S. K., AND CHEN, P. M. Extravirt: detecting and recovering from transient processor faults. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 1–8.
- [15] MARUYAMA, M., TSUMURA, T., AND NAKASHIMA, H. Parallel program debugging based on data-replay. In *IASTED PDCS'05* (2005), pp. 151–156.
- [16] MERCER, E. Verification and Validation Laboratory at Brigham Young University. <http://facwiki.cs.byu.edu/vv-lab/index.php>, April 2011. [Online; accessed 28-April-2011].
- [17] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Software profiling for deterministic replay debugging of user code. In *Proceeding of the 2006 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the fifth SoMeT 06* (Amsterdam, The Netherlands, The Netherlands, 2006), IOS Press, pp. 211–230.
- [18] NETHERCOTE, N., WALSH, R., AND FITZHARDINGE, J. Building workload characterization tools with valgrind. In *IISWC* (2006), IEEE, p. 2.
- [19] NETZER, R. H. B., BRENNAN, T. W., AND DAMODARAN-KAMAL, S. K. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (New York, NY, USA, 1996), SPDT '96, ACM, pp. 31–40.
- [20] NOETH, M., MARATHE, J., MUELLER, F., SCHULZ, M., AND DE SUPINSKI, B. Scalable compression and replay of communication traces in massively parallel environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.
- [21] QUINLAN, D. J. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters* 10, 2/3 (2000), 215–226.
- [22] RONSSE, M., CHRISTIAENS, M., AND BOSSCHERE, K. D. Cyclic debugging using execution replay. In *Proceedings of the International Conference on Computational Science-Part II* (London, UK, UK, 2001), ICCS '01, Springer-Verlag, pp. 851–860.
- [23] 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].
- [24] SHARMA, S., GOPALAKRISHNAN, G., MERCER, E., AND HOLT, J. MCC: A runtime verification tool for MCAPI user applications. In *FMCAD* (2009), IEEE, pp. 41–44.
- [25] SHEGALOV, G. *Integrated Data, Message, and Process Recovery for Failure Masking in Web Services*. PhD thesis, Universität des Saarlandes, July 2005.
- [26] SMITH, DARRY, A. Efficient recording and replaying of non-deterministic instructions in a virtual machine and CPU, 2010.
- [27] SNELLING, D. F., AND HOFFMANN, G.-R. A comparative study of libraries for parallel processing. *Parallel Computing* 8, 1-3 (1988), 255 – 266. Proceedings of the International Conference on Vector and Parallel Processors in Computational Science III.
- [28] XUE, R., LIU, X., WU, M., GUO, Z., CHEN, W., ZHENG, W., ZHANG, Z., AND VOELKER, G. MPIWiz: subgroup reproducible replay of MPI applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2009), PPOPP '09, ACM, pp. 251–260.