Verification and Coverage of Message Passing Multicore Applications

ETEM DENIZ and ALPER SEN, Bogazici University
JIM HOLT, Freescale Semiconductor Inc.

We describe verification and coverage methods for multicore software that uses message passing libraries for communication. Specifically, we provide techniques to improve reliability of software using the new industry standard MCAPI by the Multicore Association. We develop dynamic predictive verification techniques that allow us to find actual and potential errors in a multicore software. Some of these error types are deadlocks, race conditions, and violation of temporal assertions. We complement our verification techniques with a mutation-testing-based coverage metric. Coverage metrics enable measuring the quality of verification tests. We implemented our techniques in tools and validated them on several multicore programs that use the MCAPI standard. We implement our techniques in tools and experimentally show the effectiveness of our approach. We find errors that are not found using traditional dynamic verification techniques and we can potentially explore execution schedules different than the original program with our coverage tool. This is the first time such predictive verification and coverage metrics have been developed for MCAPI.

Categories and Subject Descriptors: J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD); D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Verification, Reliability, Algorithms

Additional Key Words and Phrases: Multicore software, message passing communication, predictive verification, coverage, mutation testing

ACM Reference Format:
Deniz, E., Sen, A., and Holt, J. 2012. Verification and coverage of message passing multicore applications. ACM Trans. Des. Autom. Electron. Syst. 17, 3, Article 23 (June 2012), 31 pages.
DOI = 10.1145/2209291.2209296 http://doi.acm.org/10.1145/2209291.2209296

1. INTRODUCTION

Reliability of electronic systems is crucial since errors can result in loss of money, time, and even human life. Many domains require reliable software and hardware. Reliability is especially crucial for safety-critical embedded multicore systems used in automobiles and medical instruments. The task of improving reliability has been complicated by the concurrent nature of multicore systems since concurrent systems can get into an exponential number of scenarios that cannot be completely analyzed. We need reliability techniques that can deal with concurrent multicore systems. In addition to the concurrent nature of hardware, concurrent software is also becoming common place. New multicore software formalisms are developed to exploit the performance

This research was supported by Semiconductor Research Corporation under task 2082.001, Marie Curie European Reintegration Grant within the 7th European Community Framework Programme, Bogazici University Research Fund 5483, and the Turkish Academy of Sciences.
Authors' addresses: E. Deniz and A. Sen (corresponding author), Department of Computer Engineering, Bogazici University, 34342, Istanbul, Turkey; email: alper.sen@boun.edu.tr; J. Holt, Freescale Semiconductor Inc., 7700 W. Parmer Lane, Austin, TX 78729.

DOI 10.1145/2209291.2209296 http://doi.acm.org/10.1145/2209291.2209296

available in multicore hardware. Improving the reliability of multicore software is also a big challenge due to concurrency.

Reliability is further reduced by the nondeterminism that is introduced by the concurrent software that uses the shared memory paradigm. Such software is also not scalable to heterogeneous embedded multicores with different types and number of cores, different operating systems, and physical transports. The message passing paradigm explicitly provides concurrency by using messages. This not only reduces the potential for nondeterminism but also is scalable. In the context of distributed systems and scientific programming, the Message Passing Interface standard (MPI) [MPI 2011] is widely used. The embedded system domain requires a standard with a smaller memory footprint than MPI and that exploits the properties of the domain. The Multicore Association has developed such an industry standard for multicore software development. The standard for message passing communication is called MCAPI [MCA 2011]. In this article, we provide reliability techniques for multicore software developed using MCAPI.

We use a twofold approach for improving reliability: verification and coverage. We develop a dynamic predictive verification technique that is a combination of formal methods and simulation techniques. In this technique, the designer can specify the assertions (properties) that the multicore software should satisfy. Some assertions are mutual exclusion, or deadlock and race conditions. Deadlocks and race conditions are common problems for concurrent systems. Hence, while we provide a general algorithm for checking designer-specified temporal assertions, we also provide specialized algorithms for deadlock and race condition detection. We improve the performance of our algorithms by developing enhanced dependency tracking techniques. In order to complement our verification efforts, we develop coverage metrics. When the verification process is complete, there is still a doubt whether enough properties have been written or enough scenarios have been explored. Coverage metrics allow us to measure the quality of verification efforts. We develop mutation-testing-based coverage techniques for multicore software using MCAPI. Specifically, we develop a set of mutation operators for the MCAPI standard that get inserted in programs and then we check what percentage of these mutations can be covered by verification tests. This is the first time such predictive verification and coverage metrics have been developed for the MCAPI standard.

We developed tools that implement our algorithms and experimented with multicore programs that use MCAPI. We verified and found errors in some programs that were not found using traditional dynamic verification techniques. This shows the predictive nature of our approach. Also, we show that our specialized algorithms for deadlock and race condition detection have better performance than temporal assertion verification. Our mutation-based coverage tool allows us to explore execution schedules different than the original program.

The article is organized as follows. We provide a detailed related work on reliability techniques for multicore software. Then, we describe the model that we use in this article. We describe our verification and coverage algorithms in Sections 4 and 5. The experimental section displays the effectiveness of our approach. Finally, we present our conclusions and future work.

2. RELATED WORK

There are several works that detect concurrency problems in MCAPI user applications. In Sharma et al. [2009a, 2009b] and Sharma and Gopalakrishnan [2009], the authors present the first dynamic verifier for MCAPI applications, called MCAPI Checker (MCC). Dynamic verification checks the behavior of the user application during its execution. MCC explores all possible interleavings of an MCAPI application by using the Dynamic Partial Order Reduction (DPOR) [Flanagan and Godefroid 2005] technique.

MCC handles MCAPI's connectionless send and receive functions and verifies assertions and checks for deadlocks. On the other hand, our tool handles both connection-oriented and connectionless sends and receives. MCC can insert wait in order to match receives with sends and this can potentially change the behavior of the application, as we show later in this article. MCC dynamically generates all possible execution paths by repeatedly executing the instrumented program. Although MCC guarantees to find all deadlocks and assertion violations for a given input, its overhead is high because it tries to explore all possible interleavings of a multicore application. On the other hand, our approach is orthogonal to the DPOR approach and does not suffer the overhead in DPOR.

Fault localization helps us to identify exactly where the bugs are in programs. In Elwakil and Yang [2010], a debugging tool used for detecting assertion failures that are caused by (connectionless) message races is presented. MCAPI guarantees that the messages sent from the same endpoint to a specific endpoint will arrive at the destination according to their transmission order. On the other hand, there is no rule about the arrival order of concurrent messages from different endpoints. Two or more messages can race for arriving at the same destination and in some cases, this nondeterminism can lead to assertion failures. Localization of the fault by finding the specific order of message arrivals that causes the assertion failure is as important as detecting the assertion failure. The tool presented in Elwakil and Yang [2010] symbolically explores all possible race conditions and then, by using an efficient SMT formula, it is decided whether there exists a particular order of message arrivals that results in an error state. Symbolic Debugger for MCAPI Applications (CRI) presented in Elwakil et al. [2010] is similar to the work in Elwakil and Yang [2010]. CRI instruments the MCAPI application source-code to be able to generate an execution trace of the application. The instrumented source-code is compiled and run, and then a trace is generated. The trace is encoded as an SMT formula where the formula is satisfiable, if there is a reachable error state. The last step in CRI is solving the formula by an SMT solver. CRI only supports connectionless message sends and receives. CRI finds violations of Boolean assertions but cannot find violations of temporal assertions. CRI explores all possible orders of message arrivals for a given input of an MCAPI application while deciding the satisfiability of a formula. We develop efficient race condition detection algorithms as well as verify temporal assertions.

Predictive Runtime Verification (PRV) offers a simple and efficient alternative over model checking the entire program with respect to the given specification. The PRV technique in Sen and Garg [2007] and Sen et al. [2008] is a dynamic technique where partial order traces are used instead of total order traces to model an execution and checks whether a temporal property is satisfied or violated on that partial order trace. The PRV technique has been shown to detect actual and potential errors in Java as well as in SystemC. The PRV technique catches some of the errors not exhibited in the observed total order trace, but only those that are in the partial order trace obtained from the observed total order trace. An example PRV tool, named BTV, is shown in Ogale and Garg [2007]. Our approach is similar to the work in Sen [2011] where predictive assertion verification and mutation testing have been applied to SystemC designs, which are used for hardware/software codesign. By contrast, our article targets message passing multicore applications and specifically applications written using the MCAPI standard. In this work, we present predictive deadlock and race condition detection algorithms that do not exist in Sen [2011]. We have also developed a new mutation library for message passing applications. Novel efficient vector clock algorithms that improve scalability are also new contributions of this article.

PRV is similar to Dynamic Partial Order Reduction (DPOR) in that both techniques are simulation based and apply a single input to the design, whereas in model checking all possible input combinations are applied. However, our work is also different from DPOR. In DPOR, the order of dependent transitions in the generated simulation trace is changed leading to the generation of new simulation traces until all possible changes are exhausted. This may lead to the state explosion problem for complex designs. However, we generate a partial order trace from a single simulation trace and do not modify the order of dependent transitions in order to generate new traces. DPOR is orthogonal to our approach and can be used in conjunction, where DPOR can provide all partial order traces for a given input, and our work can check the temporal properties on each partial order trace efficiently.

Checking concurrency problems such as deadlock and race conditions lends itself to algorithms with better performance than assertion checking. MCAPI is similar to the MPI standard although their target platforms are different. Hence, deadlock and race condition detection techniques that are developed for MPI can potentially also be applied to MCAPI applications.

In Hilbrich et al. [2009], the authors present a general deadlock model for MPI. They use the AND⊕OR Wait For Graph (WFG) while detecting deadlocks in MPI programs. Many MPI calls simply create a dependence on another and task dependencies must be met before the issuing task can proceed. For example, a message send call causes the task to wait for another task to post a matching receive. While all dependencies must be satisfied for the process to continue and a cycle in the WFG is a necessary and sufficient deadlock criterion for the AND model, a process may continue when any one of a set of dependencies is satisfied under the OR model. The reason why they use the AND⊕OR model instead of the AND or the OR model is that the AND model is sufficient for handling receive functions but not wildcard receive functions and the OR model is necessary for wildcard receives. Although the sender of a receive is specified for many cases, wildcard receive does not specify the sender and can be satisfied by a matching send from any task. Their detection mechanism does not detect all possible deadlocks because this consumes time and decreases performance. Instead of analyzing all potential matchings of wildcard receives, they only consider the matchings that actually occur. This approach reduces the overhead and decreases the number of false positives.

Message races can cause nondeterministic executions of concurrent programs. Park et al. [2007] present the MPIRace-Check tool, which is an on-the-fly detection tool for MPI programs written in C. MPIRace-Check finds all race conditions between message sends while the program is executed by checking the concurrent communication events between processes. They use vector clocks to determine the concurrency relation between events. The slowdown of MPIRace-Check is 26% for 10000 send/receive operations and 35% for the worst case. Our race condition detection is similar to this work but we have developed higher-performance techniques.

In Sharma et al. [2007], the authors conduct a survey of MPI debugging tools. For instance, MARMOT [Krammer et al. 2004] uses a time-out mechanism to conclude the presence of a deadlock and cannot detect even simple deadlocks. In a time-out mechanism, a blocking function call waits until a specified time and if this function is still waiting, a deadlock is reported. While MPI-SPIN, which is a model checker based on SPIN, is reliable and expandable, this suffers the state explosion problem. UMPIRE [Vetter and de Supinski 2000] dynamically analyzes MPI programming errors using a profiling interface. UMPIRE uses both a time-out mechanism and dependency graphs for detecting deadlocks. MARMOT and UMPIRE are purely runtime checking tools. On the other hand, Intel Message Checker (IMC) [Desouza et al. 2005] collects information for each MPI call in a trace file during execution and analyzes this file after the execution.

Coverage techniques have been developed in the literature. These include structural coverage, code coverage (lines, branches), and functional coverage. We are interested in fault-insertion-based coverage. This allows to measure the impact of faults in the system. Mutation testing is a fault-based software testing technique that provides a testing criterion that can be used to measure the effectiveness of a test set in terms of its ability to detect faults. Some mutation-based coverage metrics have been developed for applications written in different languages such as Java, C, and SystemC [Bradbury et al. 2006; Sen and Abadir 2010]. In Bradbury et al. [2006], the authors present a set of concurrent mutation operators after giving bug patterns for concurrent Java applications. These bug patterns are based on common mistakes that can be made by programmers in practice. Sen and Abadir [2010] developed a fault model for concurrent SystemC designs, where they define mutation operators for concurrent functions in SystemC. Our approach is similar to this approach but we developed a new mutation library that did not exist before for message passing programs.

3. MODEL

3.1. Background on Multicore Communication API (MCAPI)

Multicore Communication API (MCAPI) [MCA 2011] aims to supply communication and synchronization between closely distributed embedded systems. MCAPI is a message passing API like MPI but its target system and functionalities differ from MPI. MCAPI provides low latency and low overhead for heterogeneous platforms (in terms of types and number of cores, different operating systems, and physical transports). Shared memory used by multicore systems can lead to nondeterminism. Message passing reduces the potential for nondeterminism by explicit messages for communication. MCAPI has three fundamental communication types: connectionless datagrams for messages; connection-oriented, unidirectional, FIFO packet streams for packet channels; and connection-oriented single-word unidirectional, FIFO packet streams for scalar channels. Channels require opening before communication and closing after communication completes. Basic elements of the MCAPI topology are nodes, which can be a process, a thread, a hardware accelerator, etc. Communication occurs between endpoints, which are termination points and created on nodes on each side of the communication. More than one endpoint can be set up on each node and endpoints are identified with unique identification numbers. Both connectionless and connection-oriented communications take place between endpoints.

Connectionless messages can be sent or received in either blocking or nonblocking fashion. The blocking send function (*mcapi_msg_send*) in our MCAPI library will block if there is insufficient memory space available at the system buffer. When sufficient memory space becomes available, the function will complete. Current implementation of MCAPI library by the Multicore Association does not support this kind of blocking send function, instead it returns immediately with an error even if there is no memory space. This is similar to the nonblocking send function (*mcapi_msg_send_i*), which returns immediately even if there is no memory space available. MCAPI stores messages in a queue at the receiver endpoint and the size of the queue can be configured according to the user's demands. The blocking receive function (*mcapi_msg_recv*) returns once a message is available in the endpoint's message queue, whereas a nonblocking receive function (*mcapi_msg_recv_i*) returns immediately even if there is no message available. Message receive functions do not specify the sender endpoint and can match any of the senders depending on the execution schedule. These are also called wildcard receives. Packet channels use connection-oriented communication. They use FIFO order and they can have blocking or nonblocking send and receive functions. Scalar channels are aimed at systems that have hardware support for sending small amounts of data (for example, a hardware FIFO) and scalar channels have only blocking functions due to high performance.

Table I. Some MCAPI Functions

Type	MCAPI Function	Description
General	*mcapi_initialize*	Initializes an MCAPI node
	mcapi_finalize	Finalizes an MCAPI node
Endpoints	*mcapi_endpoint_create*	Creates an endpoint
	mcapi_endpoint_get *mcapi_endpoint_get_i*	Obtains the endpoint associated with a given tuple in blocking/non-blocking fashion
Messages	*mcapi_msg_send* *mcapi_msg_send_i*	Sends a blocking/non-blocking (connectionless) message from a send endpoint to a receive endpoint.
	mcapi_msg_recv *mcapi_msg_recv_i*	Receives in blocking/non-blocking fashion a (connectionless) message from a receive endpoint
Packet Channels	*mcapi_pktchan_connect_i*	Connects send and receive endpoints
	mcapi_pktchan_recv_open_i	Creates a typed and directional, local representation of the channel on the sender side
	mcapi_pktchan_send_open_i	Creates a typed and directional, local representation of the channel on the receiver side
	mcapi_pktchan_send *mcapi_pktchan_send_i*	Sends a blocking/non-blocking data packet on a (connected) channel
	mcapi_pktchan_recv *mcapi_pktchan_recv_i*	Receives in blocking/non-blocking fashion a data packet on a (connected) channel
	mcapi_pktchan_release	Releases a packet buffer obtained from a *mcapi_pktchan_recv()*
	mcapi_pktchan_recv_close_i	Closes the receive side of the channel
	mcapi_pktchan_send_close_i	Closes the send side of the channel
Scalar Channels	*mcapi_sclchan_send_uint64*	Sends a 64-bit scalar on a (connected) channel
	mcapi_sclchan_recv_uint64	Receives a 64-bit scalar on a (connected) channel
Non-blocking operations	*mcapi_test*	Tests if a non-blocking operation has completed
	mcapi_wait	Waits for a non-blocking operation to complete
	mcapi_wait_any	Waits for any non-blocking operation in a list to complete
	mcapi_cancel	Cancels an outstanding non-blocking operation

For nonblocking function requests, the user program receives a handle for each request and can then use the nonblocking management functions to test if the request has completed with *mcapi_test* function, or wait for it either singularly with *mcapi_wait* or wait for any one of requests in an array of requests with the *mcapi_wait_any* function. The user program can also cancel nonblocking function calls using the *mcapi_cancel* function.

MCAPI provides sufficient number of functionalities while hiding or minimizing communication overhead to get better performance. Table I contains a list of MCAPI functions. Apart from MCAPI library implementation by Multicore Association, Open-MCAPI [Open MCAPI 2011], created by Mentor Graphics, is also an open-source implementation of the MCAPI standard.

Both MCAPI and MPI have similar functions for exchanging messages. For example, the following function pairs (mpi function – mcapi function) have similar behaviors: *mpi_send* – *mcapi_msg_send*, *mpi_isend* – *mcapi_msg_send_i*, *mpi_recv* – *mcapi_msg_recv*, and *mpi_irecv* – *mcapi_msg_recv_i*.

We show an example multicore program that uses the MCAPI library in Figure 1. The program has two concurrent threads (*Thread1* and *Thread2*) communicating through connectionless nonblocking message exchange. Each thread initializes the MCAPI environment and then creates an endpoint to communicate with the other thread using *mcapi_endpoint_create*. *Thread1* gets *Thread2*'s endpoint by using the *mcapi_endpoint_get* function. *Thread1* then sends a message to *Thread2* and finalizes the MCAPI environment before exiting. *Thread2* receives the message from *Thread1* using the nonblocking message receive function. In concurrent programs, the order in which threads are scheduled is nondeterministic. If Thread1 executes *mcapi_msg_send_i* before Thread2 executes *mcapi_msg_recv_i* then Thread2 receives the message from Thread1. However, if *Thread2* executes *mcapi_msg_recv_i* before

```
#define DOMAIN 1
#define NODE1 1
#define NODE2 2
#define PORT_NUM 100
#define NUM_THREADS 2

void* run_thread_1 (void *t) {
  ...
  mcapi_boolean_t cs1 = MCAPI_TRUE;
  mcapi_initialize(DOMAIN,NODE1,&parms,&version,&status);

  ep1 = mcapi_endpoint_create(PORT_NUM,&status);   /* e1 */
  ep2 = mcapi_endpoint_get(DOMAIN,NODE2,PORT_NUM,MCA_INFINITE,&status); /*e2*/
  cs1 = MCAPI_FALSE;   /* e3 */
  mcapi_msg_send_i(ep1,ep2,''MCAPI'',size,priority,&request,&status);  /*e4*/

  mcapi_finalize(&status);
  ...
}

void* run_thread_2 (void *t) {
  ...
  buffer = '''';
  mcapi_boolean_t cs2 = MCAPI_FALSE;
  mcapi_initialize(DOMAIN,NODE2,&parms,&version,&status);

  ep2 = mcapi_endpoint_create(PORT_NUM,&status);   /* f1 */
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  ...
  cs2 = MCAPI_TRUE;   /* f2 */
  ...
  mcapi_wait(&request,&recv_size,MCA_INFINITE,&status);   /* f3 */

  mcapi_finalize(&status);
  ...
}

int main () {
  ...
  /* run all threads */
  pthread_create(&threads[0],NULL,run_thread_1,NULL);
  pthread_create(&threads[1],NULL,run_thread_2,NULL);
  /* wait for all threads */
  for (t = 0; t < NUM_THREADS; t++) {
      pthread_join(threads[t],NULL);
  }
  ...
}
```

Fig. 1. Example multicore program using MCAPI.

Thread1 executes *mcapi_msg_send_i*, *Thread2* returns from *mcapi_msg_recv_i* without receiving a message since there is no message available in its receive queue. *Thread2* then waits until the message is received by using the *mcapi_wait* function.

3.2. Trace Model

A multicore system consists of a collection of distinct endpoints which communicate with one another by message exchanges or shared memory. We consider a multicore system composed of a collection of sequential endpoints $\{ep_1, ep_2, \ldots, ep_n\}$, and an MCAPI library capable of implementing communication between pairs of endpoints for message exchanges. Each endpoint ep_i has a local state, which is determined by the

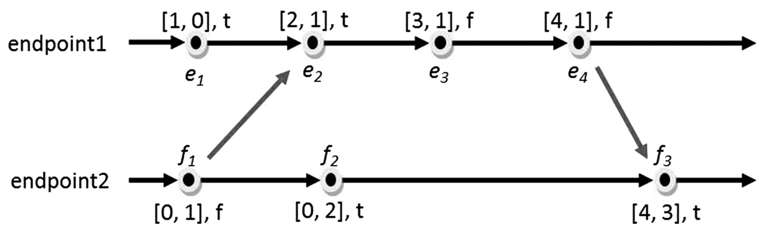

Fig. 2. A partial order trace of the example in Figure 1.

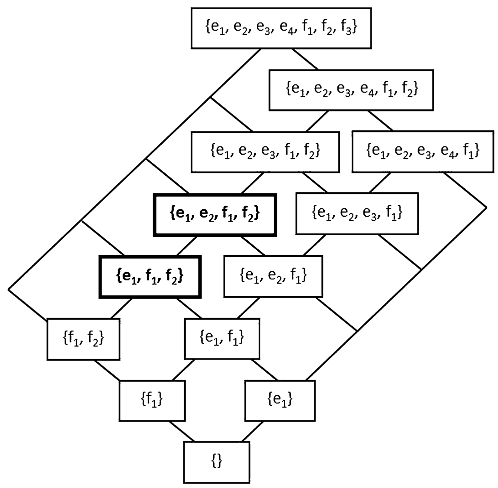

Fig. 3. State space of the partial order trace in Figure 2.

values of its local variables and events that are generated during an execution of a multicore program. Some example events are message send/receive and shared variable read/write. These events change the state of the multicore program.

An execution trace can be viewed as a partially ordered set of events called a partial order trace and we represent a partial order trace as a directed graph with vertices as the set of events and a set of edges. Figure 2 shows an example partial order trace of the example in Figure 1, when *Thread2* executes *mcapi_msg_recv_i* before *Thread1* executes *mcapi_msg_send_i*. The dots (vertices) are events and the arrows (edges) are dependencies. This partial order trace contains two endpoints (*endpoint1* and *endpoint2*), where *endpoint1* has four events which are e_1, e_2, e_3, and e_4 and *endpoint2* has three events which are f_1, f_2, and f_3. A global state is the state of the system and given by the set of events that have been executed from the beginning of the system to the current state by all endpoints. For example, $\{e_1, e_2, f_1, f_2\}$ is a global state of the partial order trace in Figure 2. We define a consistent global state on directed graphs as a subset of vertices such that, if a vertex is in the subset, then all incoming neighbors are also in the subset. In Figure 2, the global state $\{e_1, e_2\}$ is not a consistent global state because it includes $\{e_2\}$ but not $\{f_1\}$. However, $\{e_1, f_1\}$ and $\{f_1, f_2\}$ are consistent global states. Figure 3 shows the state space of the partial order trace in Figure 2 that contains all consistent global states of the trace starting from the initial state $\{\}$, and ending at the final state $\{e_1, e_2, e_3, e_4, f_1, f_2, f_3\}$ moving one event at a time. This model allows us to capture concurrency via interleaving. That is, from a given state we can obtain new states by the addition of concurrent events. For example, from state $\{e_1, f_1\}$ we can reach $\{e_1, f_1, f_2\}$ or $\{e_1, e_2, f_1\}$, since both e_2 and f_2 are concurrent as we will explain later.

3.3. Vector Clocks

There exist several techniques for tracking the concurrency information or the dependencies between events. Lamport's happened-before relation [Lamport 1978], which is a partial order relation, is used for capturing ordering between concurrent events. The happened-before relation ($\rightarrow$) is formally defined as the least-strict partial order on events such that:

—if events s and t occur on the same endpoint, $s \rightarrow t$ if the occurrence of event s preceded the occurrence of event t;
—if event s is the sending of a message and event t is the corresponding receipt of that message, $s \rightarrow t$.

We use vector clocks [Fidge 1991; Mattern 1989] to capture the happened-before relationship between events in a concurrent system. We associate a vector clock with every event. A vector clock (v) is an array of n nonnegative integers (one entry per endpoint), where $v_i[i]$ is the local clock for endpoint ep_i and for $i \neq j$, $v_i[j]$ represents endpoint ep_i's latest knowledge of endpoint ep_j's local clock.

For several applications such as predictive assertion verification, we need to track dependencies between only the relevant events. Relevant events are a subset of all the events generated during the execution, and we describe them in detail shortly for message passing and shared memory systems.

Algorithm 1 shows the details of the operations on vector clocks for both message passing and shared variables. The vector clock algorithm presented in Algorithm 1 is described by the initial conditions and the actions taken for each event type. For message passing systems, relevant events are endpoint create, get, message send, receive, test, wait, and cancel functions. Note that packet and scalar send, receive operations are also relevant events. Each endpoint sends its vector clock with outgoing messages. A message receiving endpoint receives the vector clock of the sender and updates its vector clock by taking a component-wise maximum with the vector clock included in the message.

For shared memory systems, the only relevant event is a shared variable write, where the variable is used in the property to be checked. In multicore programs, tasks (processes, threads, etc.) can communicate via a set of shared variables. Some variable updates can causally depend on others. For instance, if a task writes a shared variable x and then another task writes y due to a statement $y = x + 2$, then the update of y causally depends upon the update of x. We only consider read-write, write-read, and write-write causalities while updating vector clocks of shared variables, because the order of multiple consecutive reads of the same variable is not important. We have different vector clocks for writes and reads because changing the order of consecutive reads does not change the actual behavior of the program, whereas changing the order of write with other operations results in different behavior. We can extend the happened-before relation to read and write events of shared variables as in Rosu and Sen [2007]. For this, we use two additional n-dimensional vector clocks for each shared variable x. These vector clocks are called access and write vector clocks and we denote the access vector clock of shared variable x by $x.a$ and the write vector clock by $x.w$.

ALGORITHM 1: VectorClock

Input: an event s generated by endpoint ep_j
Output: updated vector clock v_j

<u>endpoint create event ():</u>
for $i = 1$ to n **do**
    $v_j[i] := 0$;
end for
$v_j[j] := 1$;
<u>endpoint get event (endpoint ep_k):</u>
reserve request r;
let $r.ep := ep_k$, $r.type := get$;
<u>send event (endpoint ep_j, endpoint ep_k, message m):</u>
$v_j[j] := v_j[j] + 1$;
reserve request r and buffer b;
let $r.b := b$, $r.type := send$, $r.completed := true$;
store m and v_j in b as $b.m$ and $b.vc$, respectively;
add b to the receive queue of ep_k;
<u>receive event (endpoint ep_j):</u>
if the receive queue of ep_j is not empty **then**
    receive the first request r from the receive queue of ep_j;
    $r.completed = true$;
else
    reserve request r;
    let $r.type := recv$, $r.completed := false$;
end if
<u>test event (request r):</u>
if $r.type = receive$ and $r.completed = true$ **then**
    receive buffer b of r;
    $v_j :=$ componentwiseMax(v_j, $b.vc$);
end if
if $r.type = get$ and endpoint $r.ep$ exists **then**
    $r.completed = true$;
    $v_j :=$ componentwiseMax(v_j, $v_r.ep$);
end if
$v_j[j] := v_j[j] + 1$;
<u>wait event (request r, timeout t):</u>
timeout $lt = 0$;
while $r.completed = false$ and $lt < t$ **do**
    call test event (r);
    $lt := lt + 1$;
end while
<u>shared variable read event (variable x):</u>
$v_j :=$ componentwiseMax(v_j, $x.w$);
$x.a :=$ componentwiseMax($x.a$, v_j);
<u>shared variable write event (variable x):</u>
$v_j :=$ componentwiseMax($x.a$, v_j);
$x.a := v_j$ and $x.w := x.a$;
if x is relevant to the *property* **then**
    $v_j[j] := v_j[j] + 1$;
end if

3.4. Efficient Vector Clocks for MCAPI

We now show some properties of Algorithm 1. The following relations are defined to compare two vector clocks, s.v and t.v, where they are the vector clocks assigned to the events s and t, respectively.

—$s.v = t.v \Leftrightarrow \forall x : s.v[x] = t.v[x]$
—$s.v \leq t.v \Leftrightarrow \forall x : s.v[x] \leq t.v[x]$
—$s.v < t.v \Leftrightarrow s.v \leq t.v \wedge \exists x : s.v[x] < t.v[x]$

We can define happened-before and concurrency relations between events by using vector clocks of the events as follows.

—$s \rightarrow t \Leftrightarrow s.v < t.v$
—$s||t \Leftrightarrow (\neg(s \rightarrow t) \wedge \neg(t \rightarrow s))$ (Concurrent, CC)

The last relation defined before states that events s the t are concurrent (or causally independent). In addition, if the endpoint at which an event occurred is known, the test to compare two vector clocks can be simplified and allows us to obtain performance gains. If events s and t occurred at endpoints ep_i and ep_j and are assigned vector clocks $s.v$ and $t.v$, respectively, then

—$s \rightarrow t \Leftrightarrow s.v[i] \leq t.v[i]$ (Efficient Happened Before, EHB)
—$s||t \Leftrightarrow s.v[i] > t.v[i] \wedge s.v[j] < t.v[j]$ (Efficient Concurrency, ECC).

We next show that the relations given earlier hold for our vector clock algorithm.

LEMMA 3.1. *Let s and t be events on endpoints ep_i and ep_j with vector clocks $s.v$ and $t.v$, respectively and $s \neq t$. Then, $\neg(s \rightarrow t) \Rightarrow t.v[i] < s.v[i]$.*

PROOF. We know that $\neg(s \rightarrow t)$. If $i = j$, then it follows that $t \rightarrow s$ because the local component of the vector clock is increased after each relevant event, hence $t.v[i] < s.v[i]$. If $i \neq j$, then we have two cases. The first case is $t \rightarrow s$. In this case, $s.v[i]$, which is the local clock of ep_i, is increased in s and we have $t.v[i] < s.v[i]$. The second case is $s||t$. We know that every endpoint has the most up-to-date knowledge of its local clock for concurrent events s and t and it follows that $t.v[i] < s.v[i]$. □

THEOREM 3.2. *Let s and t be events on endpoints ep_i and ep_j with vector clocks $s.v$ and $t.v$, respectively. Then, $s \rightarrow t$ if and only if $(s.v[i] \leq t.v[i])$.*

PROOF. We first show that $(s \rightarrow t)$ implies that $(s.v[i] \leq t.v[i])$. If $s \rightarrow t$, then there is a message path or shared variable read/write dependency path from s to t. Since every endpoint updates its vector clock on receipt of a message or on reading/writing a shared variable and this update is done by taking the component-wise maximum, we know the following holds: $\forall k : s.v[k] \leq t.v[k]$. Thus $(s \rightarrow t) \Rightarrow (s.v[i] \leq t.v[i])$. The converse, $s.v[i] \leq t.v[i] \Rightarrow (s \rightarrow t)$, follows from Lemma 3.1. □

Figure 4 shows our particular implementation of vector clocks in the MCAPI library for the *mcapi_msg_send_i* function. This function begins with locking the MCAPI database, which is shared between tasks, and ends with unlocking the database. All MCAPI functions need locking/unlocking operations since the database is in shared memory and accessing shared memory in multicore systems without a lock mechanism is not safe. The *mcapi_msg_send_i* function reserves a request and then checks the validity of the sender and receiver endpoints. If the request reservation is successful and endpoints are valid, then a free MCAPI buffer is found. Next, the sender endpoint increments its local clock and stores the message, its vector clock, and the clock index in the buffer. After preparing the buffer, the buffer is added to the receive queue of the receiver endpoint.

```
/* our vector clock extension for endpoints */
struct {
  uint16_t clock_index;
  uint16_t vector_clock[MCAPI_VECTOR_CLOCK_SIZE];
} vector_clock_ext;

void mcapi_trans_msg_send_i( mcapi_endpoint_t  send_ep,
  mcapi_endpoint_t  receive_ep, const char* buffer,
  size_t buffer_size, mcapi_request_t* request, mcapi_status_t* mcapi_status)
{
  struct vector_clock_ext* vc_ext;
  ...
  /* lock the database */
  assert(mcapi_trans_access_database_pre(global_rwl,MCAPI_TRUE));

  /* make sure we have an available request entry */
  if ( mcapi_trans_reserve_request_have_lock(&r)) {
    ...
    assert(mcapi_trans_decode_handle_have_lock(send_ep,&sd,&sn,&se));
    assert(mcapi_trans_decode_handle_have_lock(receive_ep,&rd,&rn,&re));
    ...

    /* find a free mcapi buffer */
    db_buff = &mcapi_db->buffers[i];     /* i.th buffer is available*/

    /* increment clock */
    vc_ext = &mcapi_db->domains[sd].nodes[sn].node_d.endpoints[se].vc_ext;
    vc_ext->vector_clock[vc_ext->clock_index] += 1;

    /* copy the buffer parm into a mcapi buffer */
    memcpy (db_buff->buff, buffer, buffer_size);

    /* store endpoint in mcapi buffer */
    db_buff->sender_clock_index = vc_ext->clock_index;

    /* store vector clock in mcapi buffer */
    memcpy(db_buff->vector_clock, vc_ext->vector_clock,
    mcapi_db->verifier_ext.num_clocks * (sizeof(uint16_t)));
    ...
    assert(setup_request_have_lock(&receive_ep, request, mcapi_status,
    completed, buffer_size, NULL, SEND, 0, 0, 0, r));
    /* the non-blocking request succeeded */
    ...
  }
  ...
  /* unlock the database */
  assert(mcapi_trans_access_database_post(global_rwl,MCAPI_TRUE));
}
```

Fig. 4. Vector clock implementation in *mcapi_msg_send_i* function.

4. PREDICTIVE VERIFICATION

In this section, we describe our predictive verification algorithms. Our algorithms are predictive because we not only find actual errors but potential errors that may result from an alternative execution of endpoints in the system. Also, our goal is to have a solution with high performance. We have two types of verification, assertion verification and deadlock/race condition detection. Given a multicore program and a property, our automated verification flow consists of the following steps, where we can turn on and off each type of verification or use them in conjunction.

(1) The property is read, and the variables are found.
(2) Tracing functions for relevant variables and shared variables are automatically added to the program.

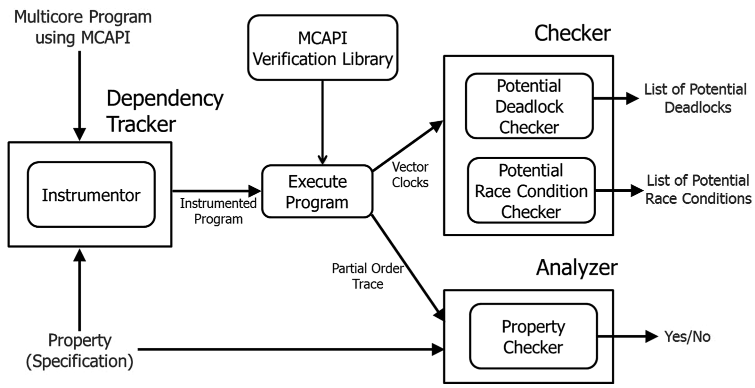

Fig. 5. Overview of predictive verification tool architecture.

(3) The instrumented program is compiled and executed with our MCAPI Verification Library, generating a partial order trace.
(4) The deadlocks and race conditions are detected during execution of the instrumented program.
(5) The resulting partial order trace and the property are passed to the BTV verifier tool, which determines whether the property is satisfied or not.

Our technique records MCAPI calls into a trace file and then we use this information to check the property given by the user. We not only detect actual errors but also potential errors with this technique. Our technique dynamically collects information about the communication and checks if deadlocks or race conditions exist. We handle both MCAPI connectionless and connection-oriented communication functions since both connectionless and connection-oriented functions create dependencies between endpoints. In addition to the communication functions, we handle endpoint operations, channel open/close/connect operations, and nonblocking operations that include wait, test, and cancel functions. In multicore programs using MCAPI, if a corresponding receive is not called for a send, we call such a send an *unmatched send* and if a corresponding send is not called for a receive, we call such a receive an *unmatched receive*. We detect unmatched sends, unmatched receives, and unclosed channels, as well.

The overall structure of our MCAPI Predictive Verification Tool (MPVT) is shown in Figure 5. The tool consists of 3 main modules: dependency tracker, analyzer, and checker. The dependency tracker module instruments the multicore user program in order to generate a partial order trace. The checker module dynamically checks deadlocks and race conditions and the analyzer module determines if the property is satisfied or not.

4.1. Instrumentation for Predictive Verification

The first step in predictive verification flow is instrumenting the multicore program. The dependency tracker module generates the execution trace of an MCAPI user program. We use vector clocks to obtain a partial order representation of traces. The partial order execution trace contains all states of endpoints and each state contains the values of variables relevant to the property.

For instrumenting the MCAPI library, instead of writing wrapper functions, we chose to modify the library functions and developed an MCAPI Verification Library. The handling of MCAPI function calls in wrapper functions can increase the execution time of the user program. In addition, current MCAPI implementation from the Multicore

Association uses shared memory to implement the MCAPI library, which may lead to race conditions in wrapper functions and preventing these using locks reduces performance. We keep the overhead low while making the solution robust. However, a wrapper function would be more beneficial for incorporating our verification algorithms to new MCAPI library implementations.

For instrumenting an MCAPI user program, the dependency tracker module automatically inserts code at appropriate locations in the user program to be monitored. The instrumented program outputs the values of variables relevant to the property given by the user. The instrumented program also updates vector clocks of endpoints for each relevant event according to the algorithm given in Algorithm 1. This is accomplished in the MCAPI Verification Library. In order to update vector clocks of shared variables, we enforce shared variable reads and writes via our verification library functions. We used shared variable instrumentation part of Inspect [Yang 2009] for instrumenting the user program. For each read/write access on variables that are shared among endpoints, Inspect intercepts the operations by adding a wrapper around it using C Intermediate Language [CIL 2011]. Upon running the instrumented multicore user program, a log file is generated. This log file consists of a sequence of events that a thread or process on which an endpoint is created goes through. Each event contains the values of variables relevant to the property being verified and a vector clock. Finally, this log file is used to obtain a partial order representation of the execution trace.

4.2. Predictive Assertion Verification

After the instrumented multicore program executes and generates a partial order trace, the analyzer module uses the Basis-based Trace Verifier (BTV) tool [Ogale and Garg 2007], to decide whether a given property is satisfied or not. BTV, which is an offline trace verifier, detects all temporal properties that can be expressed in Basic Temporal Logic (BTL). BTV can detect actual and potential errors due to a slicing technique that we developed earlier [Sen and Garg 2007]. A BTL property can have arbitrary negations, disjunctions, conjunctions, and the temporal possibly (EF) and temporal invariant (AG) operators. A property l in BTL is defined recursively as follows.

—$\forall l \in AP$ (AP is the set of atomic propositions)
—If p and q are BTL properties then the following are BTL properties, $p \vee q$, $p \wedge q$, $\neg p$, $EF(p)$

Notice that $AG(p)$ can be represented in BTL as $\neg EF(\neg p)$. A few examples of BTL properties are listed next.

—Violation of mutual exclusion: Two endpoints are in the critical section at the same time. $EF(critical_1 \wedge critical_2)$.
—Resettability: It is always possible to get to a reset state. $AG(EF(restart))$.
—All processes are never red concurrently at any future state and $process_0$ has the token: $\neg EF(red_0 \wedge red_1 \wedge \ldots) \wedge token_0$.
—It is possible to get to a state where started holds, but ready doesn't: $EF(started \wedge \neg ready)$.
—Received message size is never larger than the maximum message size defined in MCAPI: $AG(\neg(recv\_size > MCAPI\_MAX\_MSG\_SIZE))$.
—It is possible to get to a state where there is no request available: $EF(status == MCAPI\_ERR\_REQUEST\_LIMIT)$.

The core of the BTV technique is in computing a compact representation of states containing exactly those global states that satisfy the property. BTV uses a k-slicing algorithm while detecting temporal properties on a given partial order trace. The slice of a trace with respect to a property is a subtrace that contains all of the global states of the trace that satisfy the property such that it is computed efficiently and represented concisely [Sen and Garg 2007]. BTV can efficiently explore all possible traces generated from a partial order trace using slicing and without reexecution and without state space generation. BTV takes a partial order trace and a property and works recursively. The main idea behind the slicing algorithm is adding additional edges on the directed graph, which is the representation of the partial order trace. While finding the slice of p, which is a local property of endpoint ep , for each vertex that does not satisfy p, we add an additional edge to this vertex from the next vertex on ep and obtain a new graph. Notice that adding these new edges removes the states that do not satisfy p. Now, the states that do not satisfy p are not in consistent global states of the new graph since they have incoming edges from their next vertices. When the edge adding process completes, the directed graph output is the slice with respect to p. The other cases use an edge addition approach as well. While other temporal property detection techniques such as SPIN [Holzmann 1997] and JMPaX [Sen et al. 2005] have exponential-time complexity, BTV has polynomial-time complexity due to the slicing technique and restricting the subset of temporal properties; the proofs can be found in Sen and Garg [2007] and Ogale and Garg [2007]. This subset is useful to represent common concurrency properties.

4.2.1. Example. Figure 2 shows the partial order trace of an execution obtained by running the instrumented version of the example in Figure 1. This partial order trace is obtained for the observed execution schedule where *Thread2* executes *mcapi_msg_recv_i* before Thread1 executes *mcapi_msg_send_i*. The property to be checked is the mutual exclusion property, whether both endpoints can be in the critical section at the same time, that is, $EF(cs1 == MCAPI\_TRUE \wedge cs2 == MCAPI\_TRUE)$. Initially, vector clocks are all zeros and variable $cs1$ is *MCAPI_TRUE* and variable $cs2$ is *MCAPI_FALSE*. For the schedule, when *Thread2* execution is followed by *Thread1* execution, we have the following relevant operations. Relevant operations on the first endpoint are *mcapi_endpoint_create* (e_1), *mcapi_endpoint_get* (e_2), $cs1 = MCAPI\_FALSE$ (e_3), *mcapi_msg_send_i* (e_4). Relevant operations on the second endpoint are *mcapi_endpoint_create* (f_1), $cs2 = MCAPI\_TRUE$ (f_2), and *mcapi_wait* (f_3). Notice that, for the given execution schedule, the *mcapi_msg_recv_i* event is not in the relevant operations list of *endpoint2*. This is because since this is an unsuccessful function call which means that there is no message available in the receive queue and the function call returns immediately without creating any dependency. Writes to $cs1$ and $cs2$ variables generate events since these variables are relevant to the property. We assume that in the observed execution schedule the execution order of endpoints is as follows. $\{\}$, $\{e_1\}$, $\{e_1, f_1\}$, $\{e_1, e_2, f_1\}$, $\{e_1, e_2, e_3, f_1\}$, $\{e_1, e_2, e_3, e_4, f_1\}$, $\{e_1, e_2, e_3, e_4, f_1, f_2\}$, $\{e_1, e_2, e_3, e_4, f_1, f_2, f_3\}$. In Figure 2, events are also labeled with vector clocks and the values of local properties which are $true(t)$ when the local property is satisfied or $false(f)$, otherwise. Local properties of *endpoint1* and *endpoint2* are $(cs1 == MCAPI\_TRUE)$ and $(cs2 == MCAPI\_TRUE)$, respectively. The partial order trace is obtained from the observed execution schedule that has vector clocks associated with events. Figure 3 shows the state space of the partial order trace in Figure 2. When we use BTV, we find that there exists a state that satisfies the property. In fact, two states, $\{e_1, f_1, f_2\}$, $\{e_1, e_2, f_1, f_2\}$, represented as bold in Figure 3, satisfy the property. However, the observed execution schedule, which corresponds to a sequence of states in the state space that does not go through any bold state, does not satisfy the property. Hence, the error can be missed in the observed schedule but due to partial order traces we can capture this error.

It is important that MCAPI functions behave correctly and we do not force scheduler behaving in a specific way while checking a property. For instance, MCC [Sharma et al. 2009b] forces a task to wait until a nonblocking send matches with a receive or until a nonblocking receive matches with a send. Although the MCAPI standard allows the task to continue after a nonblocking operation, MCC forces the scheduler to behave differently leading to a reduced state space and potentially false positives. For instance, MCC inserts a wait after the *mcapi_msg_recv_i* function, which makes the aforesaid property unsatisfied, and misses the error. However, our algorithm finds the error.

4.3. Predictive Deadlock and Predictive Race Condition Detection

The predictive verification technique is very effective in finding bugs in concurrent programs. However, it requires user-defined properties. On the other hand, deadlocks and race conditions are undesirable for multicore programs and they can be detected automatically without any user-defined properties. We say that a *deadlock* occurs in a multicore program if two or more endpoints are each waiting for the other to complete before proceeding. If a deadlock occurs for an observed execution, we call it an *actual deadloc*k and if it does not occur for an observed execution but it can potentially occur for any of the other schedules, we call it a *potential deadlock*. We say that a *race condition* occurs in a multicore program if two or mode endpoints send a message to the same endpoint concurrently. In this case, the receiver endpoint can receive any of the sent messages and the received message can change the execution behavior. Note that all connectionless message receive functions in MCAPI are wildcard receives so multicore programs using MCAPI can potentially include many race conditions. In this work, we detect both actual and potential deadlocks and race conditions.

The checker module of MPVT contains our deadlock and race condition detection algorithms that are shown in Algorithm 2. For *deadlock detection*, we use a graph-based detection technique in order to detect actual and potential deadlocks. We dynamically build a relevant event dependency graph, which uses the AND model, and detects deadlocks. In the AND model, a vertex represents an endpoint and an edge represents the dependency between two endpoints. A cycle is sufficient to declare a deadlock with this model. When a new endpoint is created, our checker module adds a new vertex to the graph. We add a new edge from a sender endpoint to a receiver endpoint for each blocking message and packet send operation and we remove this edge when the receiver successfully receives the message or the packet. After adding a new edge, a deadlock is detected if any cycles are found in the graph and it is reported with endpoints that are in the cycles. An endpoint is allowed to make several send function calls, and it is blocked when the receive queue of the receiver is full. When we detect a deadlock, we call it an *actual deadlock*, if all the receive queues of endpoints, which are in a detected cycle, are full. If there is at least one endpoint with no full receive queue, we call it a *potential deadlock*. In potential deadlocks, it is possible that the receive queues of all endpoints may become full for other execution orders of send/receive calls. If the receive queues of all endpoints are never full for all execution orders of send/receive calls, we detect a false deadlock.

For *race condition detection*, we use a concurrency check mechanism in order to detect race conditions between message sends. We handle race condition detection in receive functions as seen in Algorithm 2. If there exists a previous receive operation on the receiver endpoint, we check the happened-before relation between the last send operation s_1 that matched with previous receive r_1 and the current send operation s_2 that matches with the current receive operation r_2 by using their vector clocks. If s_1 and s_2 are concurrent, we report a race condition. We later show that this can be more efficiently accomplished by checking whether s_1 did not happen before s_2. After checking the race condition, we store the current vector clock of the send operation in order to use it in next receive operation on the receiver endpoint. This mechanism is

ALGORITHM 2: DeadlockAndRaceCondDetection

Input: an event s generated by endpoint ep_j
Output: list of potential deadlocks and race conditions

```
endpoint create event ():
add vertex j to Dependency Graph (DG);

send event (endpoint ep_j, endpoint ep_k, message m):
add a new edge e from sender ep_j to receiver ep_k in DG;
call check_deadlock(e);
reserve buffer b;
store m in b;
store v_j and j in b as b.vc and b.ci, respectively;
add b to the receive queue of ep_k;

receive event (endpoint ep_j):
if the receive queue of ep_j is not empty then
   receive the first buffer b from the receive queue of ep_j;
   call check_race_condition(b);
   remove the edge from b.ci to ep_j in DG;
end if

check_race_condition(buffer b):
if lastsender exists then
   if lastsender_vc_j[lastsender_ci_j] > b.vc[lastsender_ci_j] then
      report race condition with receiver and senders;
   end if
end if
lastsender_vc_j := b.vc;
lastsender_ci_j := b.ci;

check_deadlock(edge e):
if e creates any cycles in DG then
   report deadlocks with the corresponding endpoints;
end if
```

very efficient in detecting race conditions because we can decide the happened-before relation by a single comparison.

We now prove that comparing single components of vector clocks is sufficient for reporting race conditions. Current implementation of MCAPI library by the Multicore Association guarantees that a receiver endpoint receives the messages in FIFO order even if they are sent from different endpoints since the library uses shared memory while exchanging messages. This system is a causally system defined as follows. Our example in Figure 6 will clarify these mechanisms further.

Definition 4.1 (*Causally Ordered*). Let any two send events s_1 and s_2 from any endpoints to the same endpoint in a multicore system be related such that s_1 happened before s_2. The corresponding receive events are r_1 and r_2, respectively. Then, the first message is received before the second message. Formally, we have the following.

$$s_1 \to s_2 \Rightarrow r_1 \to r_2 \qquad \text{(CO)}$$

THEOREM 4.2. *Let s_1 and s_2 be causally ordered send events such that s_1 did not happen before s_2 and r_1 and r_2 are the corresponding receive events, respectively. If s_1*

```
...
void* run_thread_1 (void *t) {
    ...
    ep1 = mcapi_endpoint_create(PORT_NUM,&status);   /* e1 */
    /* get other endpoints: ep2, ep3 */
    mcapi_msg_send(ep1,ep2,''msg12.1'',msgSize,priority,&status);    /* e2 */
    mcapi_msg_send(ep1,ep3,''msg13.1'',msgSize,priority,&status);    /* e3 */
    mcapi_msg_send(ep1,ep3,''msg13.2'',msgSize,priority,&status);    /* e4 */
    ...
}
void* run_thread_2 (void *t) {
    ...
    ep2 = mcapi_endpoint_create(PORT_NUM,&status);    /* f1 */
    /* get other endpoints: ep3 */
    mcapi_msg_send(ep2,ep3,''msg23.1'',msgSize,priority,&status);    /* f2 */
    mcapi_msg_recv(ep2,buffer,BUFF_SIZE,&recv_size,&status);    /* f3 */
    mcapi_msg_recv(ep2,buffer,BUFF_SIZE,&recv_size,&status);    /* f4 */
    ...
}
void* run_thread_3 (void *t) {
    ...
    ep3 = mcapi_endpoint_create(PORT_NUM,&status);    /* h1 */
    /* get other endpoints: ep2 */
    mcapi_msg_send(ep3,ep2,''msg32.1'',msgSize,priority,&status);    /* h2 */
    mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);    /* h3 */
    mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);    /* h4 */
    mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);    /* h5 */
    ...
}
int main () {
    ...
    /* run all threads */
    /* wait for all threads */
    ...
}
```

Fig. 6. Example multicore program for predictive deadlock and race condition detection.

did not occur before s_2 then they are concurrent. Formally, $(\neg(s_1 \rightarrow s_2) \wedge (r_1 \rightarrow r_2)) \Rightarrow s_1 \parallel s_2$.

Proof. We will use proof by contradiction. We assume that $\neg(s_1 \rightarrow s_2)$, $r_1 \rightarrow r_2$ and $\neg(s_1 || s_2)$. Using CC we have that $\neg((\neg(s_1 \rightarrow s_2)) \wedge (\neg(s_2 \rightarrow s_1)))$. Combining $\neg((\neg(s_1 \rightarrow s_2)) \wedge (\neg(s_2 \rightarrow s_1)))$ with $\neg(s_1 \rightarrow s_2)$ we have that $(FALSE) \vee ((s_2 \rightarrow s_1) \wedge (\neg(s_1 \rightarrow s_2)))$. Using CO this implies that $(r_2 \rightarrow r_1)$, whereas from the theorem we assume that $(r_1 \rightarrow r_2)$. This leads to a contradiction. □

4.3.1. Example. Figure 6 shows an example MCAPI user program which has a potential deadlock. The program has three threads and one endpoint for each thread. The first endpoint sends a message to the second endpoint and then sends two other messages to the third endpoint. The second endpoint sends a message to the third endpoint and then receives two messages. The third endpoint sends a message to the second endpoint and then receives thread message from any endpoint. For this example, we assumed that the receive queue size of an endpoint is 1. In other words, the receiver endpoint can store only one incoming message and the second send operation to this receiver endpoint is blocked until the receiver endpoint receives a message.

Our tool detects two race conditions during the execution of the multicore program in Figure 6. Figure 7 shows the generated partial order trace that has three endpoints *ep1*, *ep2*, and *ep3*. In the example, *ep2* receives the first message (f_3) from *ep3* (h_2). We do not check the race condition at this receive operation since there is no previous

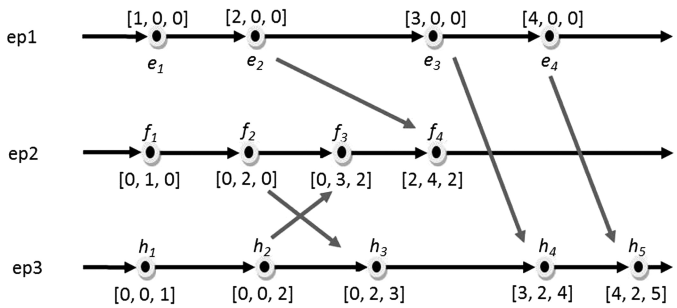

Fig. 7. Partial order trace of example in Figure 6.

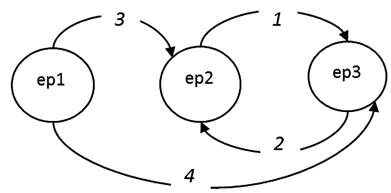

Fig. 8. Relevant event dependency graph.

receive operation on this endpoint. *ep2* receives the second message (f_4) from *ep1* (e_2). Since there exists a previous receive operation on *ep2*, we check the happened-before relation between the previous send operation (h_2) and the current send operation (e_2). We find that there is no happened-before relation and report this situation as a race condition. We detect a second race condition on *ep3*. When *ep3* receives a message (h_4) from *ep1* (e_3), we detect a race condition by finding that there is no happened-before relation between f_2 and e_3. When the third endpoint receives the second message (h_5) from *ep1* (e_4), we check for a race condition but it is clear that e_3 happened before e_4, therefore we do not report this situation as a race condition.

Figure 8 shows the relevant event dependency graph generated by the execution of the multicore program in Figure 6. Our deadlock detection mechanism runs dynamically and adds and removes edges between endpoints. First, we add the edge, represented as 1, when *ep2* sends a message (f_2) to *ep3*. Second, we add the edge (2), when *ep3* sends a message to *ep1* (h_2). We then check if a cycle exists on the graph. We find a cycle in the graph and report this as a potential deadlock. We add the third (3) and the fourth (4) edges when *ep1* sends a message to *ep2* (e_2) and to *ep3* (e_3), respectively. The detected deadlock caused by the cycle between *ep2* and *ep3* is not an actual deadlock since the receive queue of *ep3* is not full and *ep3* continues by receiving the incoming message from *ep2* (h_3) after sending a message to *ep2* (h_2). Next, *ep2* receives the message sent by *ep3* (f_3). *ep2* and *ep3* receive the remaining messages (f_4, h_4, h_5) and the execution completes. The execution in Figure 7 shows the observed execution but the order of message send and receive operations can change from one execution to the other. For example, if *ep1* sends a message to *ep2* (e_2) and to *ep3* (e_3), respectively, then *ep2* sends a message to *ep3* (f_2) and *ep3* sends a message to *ep2* (h_2). Notice that *ep3* is blocked in the third send operation (e_4) since the receive queue of *ep3* is full. *ep2* is also blocked (f_2) because the receive queue of *ep3* is already full. The only way to continue is that *ep3* receives at least one message and unblocks one of the send operations from *ep1* and *ep2* but *ep3* sends a message to *ep2* (h_2) and completes the cycle between *ep2* and *ep3* in the graph and causes a deadlock. The cycle in our case contains two endpoints; however, multicore programs can create large cycles which are detected by our detection mechanism efficiently. Our graph-based (potential) deadlock

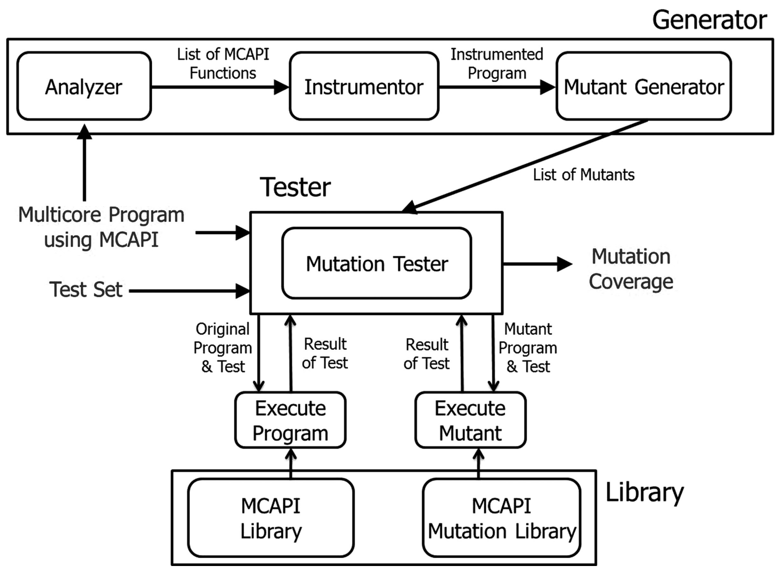

Fig. 9. Overview of mutation coverage tool architecture.

detection algorithm can report nondeadlocks as deadlock since the deadlock situation depends on the receive queue size of an endpoint.

5. VERIFICATION COVERAGE

Predictive verification aims to find errors in multicore programs. We also need a sufficient number of tests that cover all possible behaviors of the multicore program. We use mutation testing to check if the test set is sufficient for catching errors. Mutation testing is a software testing method that involves inserting faults into user programs and then rerunning a test set against the mutated program. A good test will detect the change in the program. Our aim is to check the adequacy of a test set developed for testing multicore programs that use the MCAPI library. Mutation testing allows us to have a verification coverage measure which we perform with the following steps.

(1) We create a set of mutant programs. In other words, each mutant program differs from the original program by one mutation, for example, one single syntax change made to one of the program statements.
(2) We run the original program and the mutant program with the same test set.
(3) We evaluate the results, based on the following set of criteria: If both the original program and the mutant program generate the same output, our test set is inadequate. Our test set is adequate if one of the tests in the test set detects the fault in our program, that is, one mutant program generates a different output than the original program.

We developed a tool as seen in Figure 9 for concurrent MCAPI programs to inject functional faults. Each change of the program by a mutation operator generates a mutant multicore program. We generate mutations based on our fault model and insert these mutations into a given MCAPI program to obtain a mutant. A mutant is *killed* (detected) by a test case that causes it to produce different output from the original multicore program. The ratio of the number of mutants killed to the number of all mutants is called *mutation coverage*.

Table II. Mutation Coverage Example

Test1 Outputs (buffer)		Test2 Outputs (request is valid)	
Original	Mutant	Original	Mutant
"MCAPI"	*"MCAPI"*	*FALSE*	*TRUE*

We illustrate the need to have a mutation coverage metric with a mutant obtained from the example in Figure 1. First, we generate a mutant program by removing the *mcapi_wait* function from the second thread's function body. Our test set has one test (Test1) which checks the value of the buffer variable. We run both the original and the mutant programs. If the first thread executes and exits and then the second thread executes and exits, both programs produce same value "*MCAPI*" for the *buffer* variable. This result shows that the test set is not sufficient and we add a new test (Test2) which checks the validity of the *request* variable in order to improve the test set. Note that a request is valid until the receive operation by *Thread2* completes and a test or wait operation is completed. Now, the original program produces *FALSE* and the mutant program produces *TRUE* for Test2 in the test set since the wait operation has been deleted. Table II summarizes the testing process.

5.1. Mutation Operators for MCAPI

In this section, we will identify some bug patterns in MCAPI and then develop mutation operators for MCAPI functions that will trigger these bugs. The following list contains our bug patterns. Notice that some errors in a multicore program can match with multiple bug patterns. Java concurrency bug patterns [Bradbury et al. 2006] and SystemC bug patterns [Sen and Abadir 2010] are some resources we used in developing our bug patterns.

(1) *Nondeterminism* (*ND*). Changing the timeout duration of a function, canceling an uncompleted operation may lead to a nondeterministic situation.
(2) *Deadlock* (*DL*). Insufficient system-side buffering causes deadlocks in send functions. An unmatched receive function also causes a deadlock. A task can get stuck in the *mcapi_endpoint_get* function if the endpoint that the task waits for is not created. If a task waits infinitely for a request that never completes, this causes a deadlock.
(3) *Race Condition* (*RC*). Sending two or more concurrent messages to the same endpoint causes message race conditions.
(4) *Starvation* (*SV*). A process may starve due to actions of other processes. If a task does not delete an endpoint when it is done and if other tasks try to create endpoints they can fail because of a lack of endpoints. Not closing a channel and not freeing a packet are other reasons of starvation.
(5) *Resource Exhaustion* (*RE*). A group of endpoints all have a finite set of resources, such as requests, and one of the endpoints needs a resource but none of the other endpoints gives up. In MCAPI, we use the *mcapi_test* or *mcapi_wait* function in order to remove a completed request from the system. If we do not use these functions, we may fail on new operations. Similarly, not freeing a packet even if we successfully received it may fail new message exchange operations.
(6) *Incorrect Parameters* (*IP*). This occurs when some of the parameters of an MCAPI function call are wrong. Initializing the MCAPI environment with a wrong domain identifier, creating an endpoint with a wrong node or port identifier, deleting a wrong endpoint, sending a message to the wrong endpoint, and connecting to a wrong endpoint lead to incorrect parameters bug patterns.
(7) *Forgetting Functions* (*FF*). Forgetting to call an MCAPI function causes this bug pattern. For instance, if we forget to free a packet on a memory-constrained device this causes resource leak. Forgetting to initialize or to finalize the MCAPI

Table III. Mutation Operators for MCAPI

Operator Category	Concurrency Mutation Operators for MCAPI	Description
Modify parameters of concurrency function	*MFT*	Modify Function Timeout
	MPF	Modify Parameter of Concurrent Function
Remove, replace, exchange, reorder concurrency function	*RCF*	Remove Concurrency Function
	EFC	Exchange Function Call with Another
	RTF	Replace Timed Function with Untimed Function

Table IV. MCAPI Bug Patterns and the Corresponding Mutation Operators

Index	MCAPI Bug Patterns	Mutation Operators
1	Nondeterminism (*ND*)	*MFT*, *RCF*, *EFC*, *RTF*
2	Deadlock (*DL*)	*MFT*, *MPF*, *RCF*, *EFC*, *RTF*
3	Race Condition (*RC*)	*MFT*, *RCF*, *EFC*, *RTF*
4	Starvation (*SV*)	*RCF*, *EFC*
5	Resource exhaustion (*RE*)	*RCF*
6	Incorrect parameter (*IP*)	*MPF*
7	Forgetting function (*FF*)	*MFT*, *RCF*, *EFC*, *RTF*
8	Incorrect function (*IF*)	*MPF*, *RCF*, *EFC*

environment, forgetting to establish a connection between two endpoints before transferring packet or scalar data are some examples of this pattern. If we forget to use *mcapi_test* or *mcapi_wait* after a nonblocking receive operation, we may be trying to use an unavailable data.

(8) *Incorrect Functions* (*IF*). Using a blocking function instead of a nonblocking function or vice versa causes this bug pattern. Sending or receiving a packet instead of scalar, sending or receiving a packet or scalar data on an unconnected channel, using *mcapi_test* instead of *mcapi_wait*, sending or receiving a message (not a scalar or packet data) on a connected channel are the other reasons for this bug pattern.

We present a set of mutation operators for MCAPI. These mutation operators aim to check concurrency in multicore programs. We also identify the set of MCAPI functions that a mutation operator can be applied to. Table III shows the mutation operators for MCAPI and Table IV relates them to the bug patterns previously described.

(1) *Modify Function Timeout* (*MFT*). This operator changes the timeout value of the function and can be applied to *mcapi_wait* and *mcapi_wait_any* functions since they are the only functions that have timeout parameters. We modify *mcapi_wait(time)* to *mcapi_wait(time*2)* or *mcapi_wait(time/2)* or *mcapi_wait(MCAPI_INFINITE)*. This modification may result in nondeterminism *ND*, deadlock *DL*, or race condition *RC*. For instance, when we modify the *mcapi_wait(10, request)* with *mcapi_wait(MCAPI_INFINITE, request)* in Figure 11, it results in a deadlock.
(2) *Modify the Parameter of Function* (*MPF*). This operator may lead to deadlock *DL*, or incorrect parameter *IP*.
(3) *Remove Concurrency Function* (*RCF*). This operator removes calls to concurrency functions in Table I. For example, if we remove *mcapi_wait* from the multicore program displayed in Figure 10, it leads to a race condition between *ep1* and *ep2*.
(4) *Exchange Function Call with Another* (*EFC*). This operator exchanges a function in Table I with another appropriate function. For example, we can exchange a blocking function with a nonblocking one such as *mcapi_msg_send* and *mcapi_msg_send_i*. This may lead to nondeterminism *ND*, deadlock *DL*, or starvation *SV*. If we exchange *mcapi_msg_recv_i* with *mcapi_msg_recv* in Figure 11, we cause a deadlock since *ep1* waits for *ep2* and *ep2* waits for *ep1*.
(5) *Replace Timed Function with Untimed Function* (*RTF*). This operator replaces a timed function with an untimed function. For example, when we replace *mcapi_wait*

```
void* run_thread_1 (void *t) { /* Thread1 has ep1 */
  ...
  mcapi_msg_send(ep1,ep3,''msg13'',msgSize,priority,&status);
  mcapi_msg_send_i(ep1,ep2,''msg12'',msgSize,priority,&request,&status);
  ...
}
void* run_thread_2 (void *t) { /* Thread2 has ep2 */
  ...
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  mcapi_wait(&request,&recv_size,MCAPI_INFINITE,&status); // mutation
  mcapi_msg_send(ep2,ep3,''msg23'',msgSize,priority,&status);
  ...
}
void* run_thread_3 (void *t) { /* Thread3 has ep3 */
  ...
  mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);
  ...
  mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);
  ...
}
```

Fig. 10. Inserting a mutation results in a race condition.

```
void* run_thread_1 (void *t) { /* Thread1 has ep1 */
  ...
  mcapi_msg_recv(ep1,buffer,BUFF_SIZE,&recv_size,&status);
  mcapi_msg_send(ep1,ep2,''msg1'',msgSize,priority,&status);
  ...
}
void* run_thread_2 (void *t) { /* Thread2 has ep2 */
  ...
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  mcapi_wait(&request,&recv_size,10,&status);          /* // original */
  /* mcapi_wait(&request,&recv_size,MCAPI_INFINITE,&status); // mutant */
  mcapi_msg_send(ep2,ep1,''msg2'',msgSize,priority,&status);
  ...
}
```

Fig. 11. Inserting a mutation results in deadlock.

with *mcapi_test*, function *mcapi_test* does not block the task and this situation may result in nondeterminism *ND*, deadlock *DL*, or race condition *RC*.

5.2. Mutation Coverage Tool for MCAPI

We have developed an automated tool that inserts relevant mutations to the multicore programs one by one and then checks if the mutant program is killed by any of the tests. The mutation coverage tool called MCAPI Mutation Coverage Tool (MTCT) consists of 3 main modules: generator, tester, and library. The generator has three submodules which are analyzer, instrumentor, and mutant generator.

The mutation testing process of MCAPI programs starts with program analysis. The analyzer records the locations (function name, source file path, and line number) of MCAPI functions by statically analyzing the source code and then the instrumentor module automatically replaces original function calls with wrapper function calls in order to handle mutation operations in wrappers. The instrumentor needs the location of the MCAPI functions in the program and this information is supplied by the analyzer. The mutant generator creates a list of mutants according to the function list and predefined mutation operators. For instance, we have two different mutation operators *MPF* and *RCF* for the *mcapi_endpoint_create* function.

The library module includes two libraries. The first library is the original MCAPI library and the second library is our mutation library. The mutation library contains

```
void mcapi_mut_msg_recv_i(char* file, mcapi_uint32_t line,
  mcapi_endpoint_t receive_ep, void* buffer, size_t buffer_size,
  mcapi_request_t* request, status_t* status)
{
  size_t received_size = 0;
  if (line == mut_line && strcmp(file, mut_file) == 0) {
    switch (mut_type) {
    case 1: /* remove */
      *status = MCAPI_ERR_MUTATION;
      return;
    case 2: /* exchange with blocking */
      mcapi_msg_recv(receive_ep, buffer, buffer_size, &received_size, status);
      break;
    default:
      mcapi_msg_recv_i(receive_ep, buffer, buffer_size, request, status);
      break;
    }
  } else {
    mcapi_msg_recv_i(receive_ep, buffer, buffer_size, request, status);
  }
}
```

Fig. 12. *mcapi_mut_msg_recv_i* function from our mutation library.

wrapper functions that handle mutation operations and then call the original library function. In each wrapper function, we check the mutation parameters (source file name, line number, mutation type) that are passed to the function and if they match with the current function then we activate the mutant, otherwise this function directly calls the original library function. Figure 12 shows part of the *mcapi_mut_msg_recv_i* function from our mutation library. In order to generate a mutant by exchanging the *mcapi_msg_recv_i* function with *mcapi_msg_recv* function, we set mut_file as the file name of the multicore program, *mut_line* as the line of this function, and *mut_type* as 4.

The last module, tester, activates mutants one at a time. The tester module then executes a mutant program with each test in the test set and checks if there exists a test that kills the mutant. This module returns the mutation coverage result of the test set. Higher coverage values indicate that the test set is capable of detecting concurrency bugs.

6. EXPERIMENTAL RESULTS

We have developed tools for both predictive verification and mutation testing. We obtained a scalable and fast solution that can be seamlessly integrated with current multicore programs. We tested our tools successfully on multicore programs supplied by MCAPI and developed by us because no publicly available benchmark using MCAPI is currently available. Table V shows the characteristics of multicore programs we used. The first five multicore programs are from MCAPI tests and the remaining multicore programs are developed by us. The first column in the table shows the name of the multicore program, the second column denoted by #line shows the number of lines in the multicore program, the column denoted by #ep shows the number of endpoints created during the multicore program execution, and the last column gives a brief description of the multicore program. These multicore programs cover message, packet channel, and scalar channel operations of MCAPI as well as blocking and nonblocking operation types. All the experiments were performed on a PC running Linux with a CPU of 800 MHz and 4GB of memory. The performance metrics we measured are running time (seconds) and memory usage (megabytes). The results represented in the tables are the average values that we got after running our tools one hundred times.

Table V. Characteristics of the Benchmarks

Multicore program	#line	#ep	Description
msg2	186	2	Tests blocking message send and receive calls between endpoints.
msg11	374	2	Tests non-blocking message send and receive calls between endpoints.
pkt5	402	2	The packet channel version of msg11. The order of the calls (send or receive) is chosen randomly as well as the number of packets sent or received each time.
scl1	451	8	Tests scalar channel send and receive calls.
multiMessage	419	12	A simple work pool multicore program that performs matrix multiplication and uses blocking message exchange operations.
pv1	200	16	Message exchanging between endpoints where each endpoint first sends messages and then receives the incoming messages.
pv2	156	2	Non-blocking message send and receive calls between endpoints as well as non-blocking operations such as wait. (Predictive assertion verification example in Fig. 1.)
drc1	183	32	Blocking message send/receive calls between endpoints. Each endpoint sends a message to a specific endpoint. The order of the calls generates a cycle that contains all endpoints.
drc2	189	3	Blocking message send/receive calls between endpoints. (Predictive deadlock and race condition detection example in Fig. 6.)
drc3	200	32	Multicore program pv1 with 32 endpoints.
rc1	233	3	Two endpoints concurrently sending messages to the same endpoint.

Table VI. Properties for the Benchmarks

Multicore program	Property
msg11	Overflow or underflow occurs at any time during execution. Overflow occurs when the number of the un-received messages is greater than 16 and underflow occurs when the number of un-sent message is greater than 16: $(EF((i\_s >= i\_r + 16)\|(i\_r >= i\_s + 16)))$
pkt5	Overflow, underflow, memory limit error in sender, or request limit error in receiver occurs at any time during execution: $(EF(((i\_s >= i\_r + 64)\|(i\_r >= i\_s + 64))\|((sender\_status == MCAPI\_ERR\_MEM\_LIMIT)\|(recvr\_status == MCAPI\_ERR\_REQUEST\_LIMIT))))$
scl1	The return codes (rc) of the function calls in main function are always true: $(AG(rc == MCAPI\_TRUE))$
pv1	Sent message size is greater than MCAPI_MAX_MSG_SIZE or received message is truncated at any time during execution: $(EF((s\_size > MCAPI\_MAX\_MSG\_SIZE)\|(recv\_status == MCAPI\_ERR\_MSG\_TRUNCATED)))$
pv2	Two endpoints are in critical section at the same time: $(EF((cs1 == MCAPI\_TRUE)\&(cs2 == MCAPI\_TRUE)))$
rc1	It is always true that if one of the senders sends a message, eventually the receiver receives the message: $AG(((is\_sender1\_turn == MCAPI\_TRUE)\|(is\_sender2\_turn == MCAPI\_TRUE)) \Rightarrow EF(is\_receiver\_turn == MCAPI\_TRUE))$

6.1. Predictive Verification Experiments

We have performed two experiments on multicore programs using our predictive verification tool. In the first set of experiments, we check assertion violations and detect deadlocks, race conditions, as well as unmatched calls. In the second set of experiments, we only detect deadlocks, race conditions, and unmatched calls. Unmatched calls contain unmatched sends and unmatched receives of messages, packets, and scalars, as well as unmatched channel open calls. If an opened channel is not closed with a channel close call, this causes an unmatched channel open call.

6.1.1. Predictive Assertion Verification Experiments. For the first set of experiments, we used six of the benchmarks for validating our predictive assertion verification tool. Table VI contains the multicore programs and the properties for each multicore

Table VII. Experimental Results of Predictive Assertion Verification

Multicore program	Satisfied	ORGtime	Itime	IRtime	TCtime	BTVtime	BTVmem	TotalTime
msg11	Yes	0.022	0.19	0.027	0.269	0.028	0.32	0.514
pkt5	Yes/No	0.022	0.20	0.031	0.517	0.098	0.32	0.846
scl1	Yes	0.021	0.21	0.025	0.317	0.010	0.01	0.562
pv1	No	0.102	0.18	0.118	0.105	0.004	0.32	0.573
pv2	Yes	0.010	0.17	0.013	0.092	0.001	0.01	0.276
rc1	Yes	0.049	0.20	0.058	0.269	0.014	0.32	0.541

program that we developed. Our properties are related with problems that occur in concurrent message passing systems, for instance, the size of the sent message being larger than the maximum message size defined in the MCAPI library or the received message being truncated since the size of the available buffer, which is used for storing the received message, is not sufficient. In addition, we can define properties for checking whether a specific status such as *MCAPI_ERR_MEM_LIMIT* or *MCAPI_ERR_REQUEST_LIMIT* returns from a MCAPI function call at any time during the execution of a multicore program.

Table VII shows our predictive assertion verification results. In the table, the column denoted by Satisfied represents whether the property given in Table VI is satisfied or not. We denote the running time of original multicore program in the column ORGtime and the running time of the instrumented multicore program in the IRtime column. The column denoted by Itime represents the time used by the instrumentor Inspect for shared variable instrumentation. The column denoted by TCtime represents the time used by our trace converter that converts a partial order trace generated by execution of the instrumented multicore program to the input format of BTV for assertion verification. We represent the time and the memory used by BTV in columns BTVtime and BTVmem, respectively. The last column, TotalTime, represents the total time that includes instrumentation, running time of instrumented multicore program, conversion of the partial order trace, and BTV analysis time. Note that during the execution of the instrumented program, we run our vector clock, deadlock, and race condition detection algorithms. We also check unmatched calls and generate the partial order trace of the execution.

We verified that msg11, scl1, pv2, and rc1 always satisfied the properties and pv1 never satisfied the property. Multicore program pkt5 satisfied the property for some observed executions and did not satisfy for other observed executions since messages are randomly sent and received. In other words, depending on the execution order of send/receive calls the property is satisfied or not. Two components, namely the trace converter and the Inspect instrumentor, result in the largest slowdown for our approach although the instrumented program and the BTV analyzer run fast. For example, we have the largest slowdowns for pkt5 and pv2 since the values of variables relevant to the property are updated many times in these programs, and for each update we dump the new value of the variable in the trace. That is, the sizes of the partial order traces to be converted are large and more time is spent on instrumenting the programs. We also observed that for programs with more complex assertions the BTV analysis time goes up, for example, the pkt5 example. Our predictive assertion verification tool found not only actual assertion violations but also potential ones. For instance, when we observed the execution of pv2, there was no state where both cs1 and cs2 are true. However, our tool found a state where the property is satisfied, and found the error by exploring the partial order trace generated during execution of pv2.

6.1.2. Predictive Deadlock and Race Condition Detection Experiments. For the second set of experiments, we used all multicore programs given in Table V. Table VIII shows our predictive deadlock, race condition, and unmatched call detection results. In the table,

Table VIII. Experimental Results of Predictive Deadlock and Race Condition Detection

Multicore program	#DL	#RC	Unmatched Calls	ORGtime	Itime	IRtime	TotalTime
msg2	–	–	No	0.011	0.15	0.012	0.162
msg11	–	–	Yes	0.022	0.17	0.023	0.193
pkt5	–	–	Yes	0.022	0.18	0.024	0.204
scl1	–	–	Yes	0.021	0.15	0.023	0.173
multiMessage	–	–	No	0.033	0.16	0.035	0.195
pv1	1	2	No	0.102	0.14	0.115	0.255
pv2	–	–	No	0.010	0.15	0.011	0.161
drc1	1	–	No	0.200	0.17	0.208	0.378
drc2	1	2	No	0.016	0.18	0.019	0.199
drc3	4	7	No	0.221	0.17	0.283	0.453
rc1	–	99	No	0.049	0.16	0.055	0.215

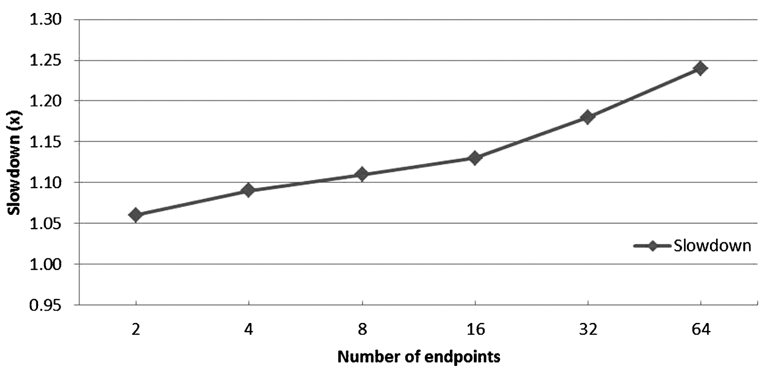

Fig. 13. Slowdown of pv1 deadlock, race condition detection.

the column denoted by #DL represents the number of deadlocks detected, and #RC represents the number of race conditions detected. The column denoted by Unmatched Calls represents whether unmatched calls were detected or not. We denote the running time of the multicore program in column ORGtime and the running time of the instrumented multicore program in column IRtime. The column Itime represents the instrumentation time by Inspect and the last column represents the total time used.

Our deadlock and race condition detection algorithms work online and do not use the entire partial order trace. However, predictive assertion verification works offline and needs all of the partial order trace to detect temporal assertions. Additionally, we need to monitor the variables relevant to the property in predictive assertion verification as well as shared variables. Hence, the times have gone down in Table V for the same examples compared with Table VII.

Experimental results in Table VIII show that multicore programs msg2, multiMessage, and pv2 are error-free programs. Six of the programs do not include deadlocks or race conditions but three of the programs have unmatched calls. We detected unmatched calls for multicore programs msg11, pkt5, and scl1. First, msg11 has 4 unmatched message receive calls. Second, pkt5 has 3 unmatched packet send calls, 1 unmatched packet send open call, and 1 unmatched packet receive open call. Last, scl1 has 4 unmatched scalar send calls.

We observe that multicore programs with a large number of deadlocks have larger slowdowns. In fact, the slowdown mostly depends on the number of the endpoints in the cycle. For instance, we have the largest slowdown for drc3 and where we detected 4 deadlocks and each detected deadlock has more than 20 endpoints that generated the cycle. Figure 13 shows the slowdown values for different numbers of endpoints for

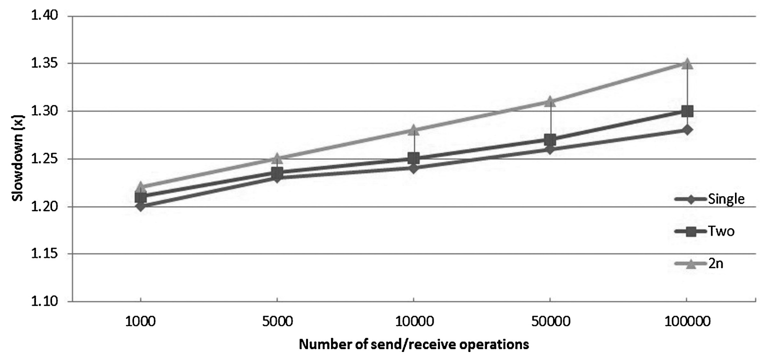

Fig. 14. Performance improvement due to reduced vector clock comparisons in race condition detection.

deadlock, race condition checking. We used pv1 for obtaining slowdown values where we incremented the number of the endpoints while the other parts of the multicore program were the same. For instance, the slowdown of checking errors in pv1 is 1.18× for 32 endpoints and 1.24× for 64 endpoints, which shows that we do not suffer from performance when the number of endpoints is increased.

For estimating the efficiency of our race condition detection technique, we use the multicore program rc1. This program consists of *mcapi_msg_send* and *mcapi_msg_recv* operations and users can change the number of those operations. In this program, there are three endpoints and only the second endpoint receives messages sent from the first and the third endpoints. We disabled deadlock and unmatched call detection mechanisms in order to see only the slowdown of race condition detection. Moreover, we extended our experiment in order to see the improvements due to Theorem 4.2. We know that if two endpoints concurrently send messages to the same endpoint, a message race occurs in the receive operation. We can check whether the sender events can be concurrent by two comparisons when using ECC and 2n comparisons when using CC, where n is the number of endpoints in the system. In CC, we need 2n comparisons since we compare each component of two vector clocks with n elements for two happened-before relations. We further improved the performance by the help of Theorem 4.2, where a single comparison to check the happened-before relation as shown in EHB is sufficient.

Figure 14 shows the slowdown of our technique and compares the results of a single comparison with two comparisons and 2n comparisons. For example, when we set the number of send/receive operations to 5000, the original program took 3.018 seconds. When we ran with our verification tool, it took 3.712 seconds with single comparison, 3.727 seconds with two comparisons, and 3.801 seconds with 2n comparisons. As we increase the number of send/receive operations, the enhancement that comes from doing a single comparison becomes more visible. Therefore our tool is efficient as an on-the-fly detection tool and can work on large-scale multicore programs.

In summary, our tool meets scalability while providing fast predictive verification.

6.2. Mutation Coverage Experiments

We have performed experiments on multicore programs using our mutation coverage tool. In the experiments, due to the lack of test sets, we used a single test that checks the exit code of the given multicore program. If a mutant multicore program exits with a code that is equal to the exit code of original multicore program, then we say that the mutant is alive, otherwise it is killed. We ran our mutation coverage tool and obtained the mutation coverage results in Table IX.

Table IX. Experimental Results for Mutation Testing

Multicore program	#Mutants	#Killed Mutants	MutCov (%)	Runtime
msg2	17	11	65	0.562
msg11	26	12	46	0.308
pkt5	34	13	38	0.294
scl1	91	57	63	0.905
multiMessage	20	10	50	1.112
pv1	11	6	55	0.297
pv2	25	15	60	0.612
drc1	17	14	82	1.115
drc2	23	12	48	0.070
drc3	11	7	64	0.411
rc1	27	16	59	0.685

In the table, we denote the number of generated mutants in column #Mutants and the number of killed mutants in the #Killed Mutants column. The column denoted by MutCov represents the mutation coverage percentage. Finally, the last column represents the total time that is consumed for mutant generation and executing all mutants. Experimental results show that mutation coverage is over 50% for many programs. The running time of our tool is nearly one second even for a high number of mutants. For instance, our tool generated 91 mutants for scl1 and 57 of them are killed by the test set in less than one second. The running time increases if an actual deadlock occurs when a mutant executes. In order to detect deadlocks in a mutant, we used a timeout approach, which declares a deadlock if a specified time period has elapsed. Multicore program drc1 has the maximum time (1.115 seconds) and we know that many of the generated mutants result in actual deadlocks. For instance, when we remove the matching send call of a blocking receive call in drc1, this causes an actual deadlock. We obtained low coverage for mutants where the injected mutation code does not execute in the observed execution.

The user can improve the mutation coverage by checking the exit status after every MCAPI function call. For example, in the multicore program pv2, although the coverage was 44%, we increased it to 68% after the addition of six status checks. It is clear that checking the status of MCAPI function calls is efficient in killing a mutant obtained by the *RCF* operator. For killing the mutants injected using other operators, the user can iteratively improve the test set by adding new tests.

Generated mutant multicore programs can potentially have different execution schedules than the original multicore program. For instance, the multicore program in Figure 1 can have a mutant that uses a blocking message receive call (*mcapi_msg_recv*) instead of a nonblocking one. The execution order of send/receive operations in the original multicore program depends on the thread schedule. The send operation in the mutant always completes before the receive operation since the receive operation blocks *Thread2* until a matching send is called. Mutants generated by our tool can help detect errors due to other possible executions of the same program.

We also performed experiments which show that mutants with race conditions, deadlocks, inference, etc., can all be detected by our predictive verification tool.

7. CONCLUSIONS AND FUTURE WORKS

We present verification and coverage techniques for multicore applications that use the message passing MCAPI standard. Our techniques are dynamic and predictive, which allows us to efficiently detect not only actual errors but also potential ones. Specifically, we developed predictive temporal assertion verification algorithms and specialized algorithms for predictive deadlock and race condition detection. We experimentally showed the effectiveness of our techniques on several applications, where we found bugs that were not found using traditional dynamic verification approaches. Performance was also an important factor in developing our algorithms. We observed that the specialized deadlock and race condition detection algorithms run much faster compared to the assertion verification algorithms. We further improved performance of our race condition detection algorithm by developing a faster comparison engine for concurrent events while exploiting the MCAPI standard. We believe that the performance of our tools can further be improved since the main slowdown comes from the partial order trace converter and the program instrumentor.

In order to develop and measure the quality of tests for message passing multicore programs, we developed mutation operators for the MCAPI standard. We observed that the mutant programs obtained by inserting mutations into the original programs can potentially explore execution schedules different than the original program. This is a useful tool for analyzing different behaviors of concurrent systems. Also, we showed that the coverage can be improved by writing new tests.

In summary, our solutions can improve the reliability of heterogeneous embedded multicore systems by pruning out actual and potential errors and determining the verification coverage, all with a small overhead.

A future research topic could be developing an AND⊕OR model for detecting deadlocks in multicore programs using MCAPI. Using the AND⊕OR model handles more deadlock situations and increases deadlock detection coverage.

MCAPI lacks larger benchmarks, hence we plan to work on developing benchmarks that will allow us to better measure the effectiveness of our techniques.

REFERENCES

BRADBURY, J. S., CORDY, J. R., AND DINGEL, J. 2006. Mutation operators for concurrent Java (J2SE 5.0). In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06).* 83–92.

CIL. 2011. C intermediate language (cil). http://cil.sourceforge.net/.

DESOUZA, J., KUHN, B., AND SUPINSKI, B. R. D. 2005. Automated, scalable debugging of mpi programs with intel message checker. In *Proceedings of the Workshop on Software Engineering for High Performance Computing System Applications (SE-HPCS'05).*

ELWAKIL, M. AND YANG, Z. 2010. Debugging support tool for mcapi applications. In *Proceedings of the Conference on Parallel and Distributed Systems: Testing, Analysis and Debugging'10*. 20–25.

ELWAKIL, M., YANG, Z., AND WANG, L. 2010. CRI: Symbolic debugger for mcapi applications. In *Proceedings of the Conference on Automated Technology for Verification and Analysis'10*. 353–358.

FIDGE, C. J. 1991. Logical time in distributed computing systems. *IEEE Comput. 24*, 8, 28–33.

FLANAGAN, C. AND GODEFROID, P. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the Conference on Principles of Programming Languages'05.* 110–121.

HILBRICH, T., DE SUPINSKI, B. R., SCHULZ, M., AND MILLER, M. S. 2009. A graph based approach for mpi deadlock detection. In *Proceedings of the International Conference on Supercomputing'09.* 296–305.

HOLZMANN, G. J. 1997. The model checker spin. *IEEE Trans. Softw. Engin. 23*, 5, 279–295.

KRAMMER, B., MILLER, M. S., AND RESCH, M. M. 2004. MPI application development using the analysis tool marmot. In *Proceedings of the International Conference on Computational Science'04.* 464–471.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7, 558–565.

MATTERN, F. 1989. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms,* North-Holland, 215–226.

MCA. 2011. Multicore association. http://www.multicore-association.org.

MPI. 2011. Message passing interface. http://www.mcs.anl.gov/mpi/.

OGALE, V. A. AND GARG, V. K. 2007. Detecting temporal logic predicates on distributed computations. In *Proceedings of the Conference on Distributed Computing'07*. 420–434.

OPEN MCAPI. 2011. Mentor graphics. http://www.mentor.com/embedded-software/.

PARK, M.-Y., SHIM, S. J., JUN, Y.-K., AND PARK, H. R. 2007. MPIRace-Check: Detection of message races in mpi programs. In *Proceedings of the Conference on Grid and Pervasive Computing'07.* 322–333.

Rosu, G. and Sen, K. 2007. An instrumentation technique for online analysis of multithreaded programs. *Concur. Comput. Pract. Exper. 19*, 3, 311–325.

Sen, A. 2011. Concurrency-Oriented verification and coverage of system level designs. *ACM Trans. Des. Autom. Electron. Syst. 16*, 4, 37:1–37:25.

Sen, A. and Abadir, M. S. 2010. Coverage metrics for verification of concurrent SystemC designs using mutation testing. In *Proceedings of the IEEE International High-Level Design, Validation and Test Workshop*. IEEE Computer Society, 75–81.

Sen, A. and Garg, V. K. 2007. Formal verification of simulation traces using computation slicing. *IEEE Trans. Comput. 56*, 4, 511–527.

Sen, A., Ogale, V., and Abadir, M. S. 2008. Predictive runtime verification of multi-processor SoCs in SystemC. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'08)*. 948–953.

Sen, K., Rosu, G., and Agha, G. 2005. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Proceedings of the Conference on Formal Methods for Open Object-Based Distributed Systems'05*. 211–226.

Sharma, S. and Gopalakrishnan, G. 2009. Formal verification of mcapi applications using the dynamic verification tool mcc. http://www.cs.utah.edu/formal_verification/pdf/subodh_techcon.pdf.

Sharma, S., Gopalakrishnan, G., and Kirby, R. M. 2007. A survey of mpi related debuggers and tools. Tech. rep. UUCS-07-015, School of Computing, University of Utah.

Sharma, S., Gopalakrishnan, G., and Mercer, E. 2009a. Dynamic verification of multicore communication applications in mcapi. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*.

Sharma, S., Gopalakrishnan, G., Mercer, E., and Holt, J. 2009b. MCC: A runtime verification tool for mcapi user applications. In *Proceedings of the Conference on Formal Methods in Computer Aided Design'09*. 41–44.

Vetter, J. S. and de Supinski, B. R. 2000. Dynamic software testing of mpi applications with umpire. In *Proceedings of the International Conference on Supercomputing'00*.

Yang, Y. 2009. Efficient dynamic verification of concurrent programs. Ph.D. thesis, School of Computing, University of Utah.

Received April 2011; revised October 2011; accepted January 2012