Dynamic Verification of Multicore Communication Applications in MCAPI

Subodh Sharma
School of Computing
University of Utah
Salt Lake City, UT 84112
svs@cs.utah.edu
www.cs.utah.edu/~svs

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112
ganesh@cs.utah.edu
www.cs.utah.edu/~ganesh

Eric Mercer
Computer Science Department
Brigham Young University
Provo, UT 84602.
eric.mercer@byu.edu
http://faculty.cs.byu.edu/~egm

***Abstract*—We present a dynamic direct code verification tool called MCC (MCAPI Checker) for applications written in the newly proposed Multicore Communications API (MCAPI). MCAPI provides both message passing and threading constructs, making the concurrent programming involved in MCAPI application development a non-trivial challenge. MCC intercepts MCAPI calls issued by user applications. Then, using a verification scheduler, MCC orchestrates a dependency directed replay of all relevant thread interleavings. This paper presents the technical challenges in handling MCC's non-blocking constructs. This is the first dynamic model checker for MCAPI applications, and as such our work provides designers the opportunity to use a formal design tool in verifying MCAPI applications and evaluating MCAPI itself in the formative stages of MCAPI.**

***Keywords*-Dynamic Verification, Software, Partial Order, Model Checking**

I. Introduction

It has been observed that the combined use of threading and message passing is necessary in order to create efficient multicore applications. This will require the standardization of an API for inter-core communication and synchronization. MCAPI [1] is one such effort which is under active development by a group of 25 leading companies in the embedded system's market. Unlike large existing APIs like MPI [2] which target high-end compute clusters, MCAPI is designed keeping in mind the very specific needs and goals of embedded software/hardware system developers. MCAPI is aimed at programmers writing applications for embedded distributed systems employing loosely coupled cores. In particular, MCAPI is well suited for systems that have much smaller memory footprints and are much more oriented towards reactive behaviors than computational. This paper describes the first direct code dynamic verification tool for MCAPI applications called MCC (MCAPI Checker). It takes as input a C code and verifies it directly as opposed to classical model checking tools which need a high level abstract model of the program (like SPIN [14] etc.). The process of constructing models is known to be cumbersome and bug prone. Therefore we resort to dynamic direct code verification methods that were originally pioneered in Verisoft [4]. Dynamic formal verification is witnessing ever growing presence in tools such as CHESS [9], Java Pathfinder [6], etc. In order to contain the thread interleaving explosion we use partial order methods that have been shown to be quite effective in software verification. Dynamic verification methods differ in the way in which the partial order reduction (POR) method operates within them. Dynamic verifiers compute more precise ample-sets [12] at runtime instead of when computed statically. MCC uses a customized version of dynamic partial order reduction (DPOR [5]) that is similar to the partial order with elusive interleavings (POE) algorithm explained in [11].

MCC builds on the strength of past projects namely ISP [8] and Inspect [7]. However, there are subtle differences between MCC, ISP, and Inspect. ISP is a purely MPI based verifier and Inspect is purely a shared memory thread program verifier. MCC, on the other hand, accommodates Pthread create and join calls as well as message passing based MCAPI calls. Furthermore, MCC differs from ISP in the manner in which non-determinism is handled in the input programs. ISP uses dynamic rewrite mechanism to force a deterministic match at runtime. MCAPI provides only non-deterministic receive calls, therefore, in the absence of specific receives the dynamic rewrite mechanism cannot work for MCC. We have extended our work presented in [13] by now supporting "get/create" endpoint calls, connection-less non-blocking constructs and the "wait" call. The novelty of the work presented in this paper lies in the way we enforce a deterministic match at the runtime. We insert an implicit wait call in the instruction stream after a send and non-blocking receive pair has been given a go-ahead by the MCC scheduler. More details can be found in Section III-D.

Contribution: The contributions of this paper are two fold. First, we have added support for non-blocking constructs to our verifier, and second, we have devised a novel way to enforce a deterministic match at the runtime by forcing a wait to acknowledge completion of issued calls to the runtime; thereby avoiding the possibility of a communication race.

The rest of the paper is organized as follows: Section II begins with an informal overview of MCAPI. Section III describes the need to have a systematic and exhaustive ex-

This work was funded collaboratively by NSF CCF 0903408/0903491 and SRC contracts 2009-TJ-1993/1994

ploration of MCAPI programs followed by MCC details in Section III-A and Section III-B. Section III-C explains the working of the MCC scheduler with the help of an example and Section III-D explains the scheduler pseudo-code followed by results and conclusions in Section IV.

II. Overview of MCAPI

The MCAPI effort traces its heritage to MPI and Socket communication libraries; however it differs from both with respect to the application domain it targets and the functionality it offers. MCAPI is less flexible than MPI (i.e., offers fewer functionalities as compared to MPI). It is an API specification for the inter-core communication in a loosely coupled distributed embedded SoC.

MCAPI defines three communication types viz., connection-less datagrams, connection-oriented FIFO packet streams and connection-oriented FIFO scalar streams. MCAPI communication is performed by nodes which are abstract entities that could either be a process, a thread, a hardware accelerator or a processor core. Furthermore, nodes communicate with each other via endpoints that are the communication termination points. Endpoints are defined as a tuple of ⟨node id, port id⟩ pair. Each node can support multiple endpoints and every endpoint in the system is assigned a globally unique identifier. Since MCC currently supports only connection-less MCAPI constructs, we will therefore restrict the discussion in this paper to only those API calls. The connection-less communication type of MCAPI is similar to MPI in that there is not static routing of messages. The API provides blocking and non-blocking variants of a send, receive, wait and test call to check the successful completion of non-blocking requests. An example code illustrating the usage of MCAPI calls in a C compilable code is shown in Figure 1.

```
1:#define NUM_THREADS 3
2:#define PORT_NUM 1

3:void* run_thread (void *t) {
4:    thread_start();
    ...
5:  mcapi_initialize(tid,&version,&status);
6:  if (tid == 2) {
7:  recv_endpt =
    pmcapi_create_endpoint (PORT_NUM,&status);
8:  pmcapi_msg_recv(recv_endpt,msg,
                  BUFF_SIZE,&recv_size,
                  &status);
9:  pmcapi_msg_recv(recv_endpt,msg
                  BUFF_SIZE, &recv_size,
                  &status);
10:   } else {
11:  send_endpt = mcapi_create_endpoint
                  (PORT_NUM,&status);
12:  recv_endpt = mcapi_get_endpoint
                  (2,PORT_NUM,&status);
13:  pmcapi_msg_send(send_endpt,recv_endpt,
                  msg,strlen(msg),
                  1,&status);
14:  }
15:  pmcapi_finalize(&status);
     ...
16:  thread_end();
     }

17:int main () {
     ...
18:  main_thread_start();
19:  for(t=0; t<NUM_THREADS; t++){
20:    rc = mcapi_thread_create(&threads[t],
       NULL, run_thread,
       (void *)&thread_data_array[t]);
21:  }
22:  for (t = 0; t < NUM_THREADS; t++) {
23:    mcapi_thread_join(threads[t],NULL);
24:  }
25: main_thread_end();
    ...
    }
```

Fig. 1. MCAPI example C program

III. Verification of MCAPI User Applications

Consider the example shown in Figure 2. Note that MCAPI connection-less receive calls are non-deterministic because each receive could potentially be matched at runtime with either send. Assume that the correctness of the program depends on whether T0's send or T1's send matches T2's first receive. While the runtime will always explore only one of the two possible execution scenarios, we must explore both the scenarios to guarantee program correctness. This simple example illustrates the futility of ad-hoc testing of concurrent programs in general. One popular approach for bug detection is to perturb schedules by inducing random sleep statements. Such a method does help manifest previously untried schedules, however, it certainly does not provide any guarantee of covering *all relevant* schedules. Such a guarantee is furnished by the MCC scheduler.

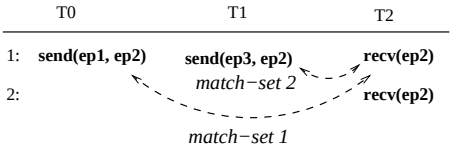

Fig. 2. MCAPI Receive Nondeterminism

A. MCC Overview

MCC is based on the current reference implementation of MCAPI provided by the MCA. The reference implementation uses Pthreads and a thread describes the notion of a node. Communication is performed only after a node has successfully issued MCAPI_INITIALIZE. It is an error to issue a communication call after a node has performed an MCAPI_FINALIZE. We have identified a list of safety properties that are important to ensure a correct and safe use of the API. For instance, invoking a communication call without creating valid endpoints or accessing the data buffer (passed to a non-blocking call) before the corresponding wait operation is issued are few of the conditions that violate the correctness of an MCAPI program. We have begun a list of default usage properties [10] which we hope to incorporate in MCC in the near future. Figure 3 describes an high level work-flow of the MCC tool. MCC has three components. The first component

instruments an input MCAPI C user program at compile time. As a part of the instrumentation process all the MCAPI calls along with the Pthread create/join calls are prefixed with an alphabet "p". These instrumented calls serve as wrappers to the actual MCAPI calls. Additionally, the thread function bodies are enveloped within the calls *thread_start* and *thread_end* and the main thread is instrumented with a *main_start* and *main_end* call. Figure 1 shows a snippet of instrumented C code that has the same communication pattern as depicted in Figure 4. Note that thread function body is instrumented with a *thread_start* (line 4) and a *thread_end* (line 16) call. The *thread_end* call notifies the scheduler that thread count, a piece of information noted by the scheduler before processing any instrumented call, should be decremented by one. The thread count helps the scheduler to determine when all threads have blocked. The *thread_start* call acts as a barrier (global fence) operation. In other words, all the threads (except the main thread) have to issue the *thread_start* call before any thread can proceed with its execution. The main thread is also instrumented with a *main_thread_start* and a *main_thread_end* call (lines 18, 25). These calls notify the scheduler of the start and end of the verification process. Additionally, the traditional Pthread create and join calls are also instrumented. The reason for create/join call instrumentation will become apparent in section III-D. All the MCAPI related calls are replaced with the wrapper calls that are defined in the profiler component of MCC.

The second component of MCC is the profiler that has function definitions of the instrumented calls. The profiler functions perform the necessary book-keeping and communicate the information collected to the scheduler. The functions block until they receive a signal to continue with the execution from the scheduler. The profiler wrapper functions eventually issue the actual MCAPI calls to the runtime.

The third component of MCC is the scheduler that ultimately decides which calls should be issued to the runtime and subsequently signals the blocked threads to unblock and execute those calls.

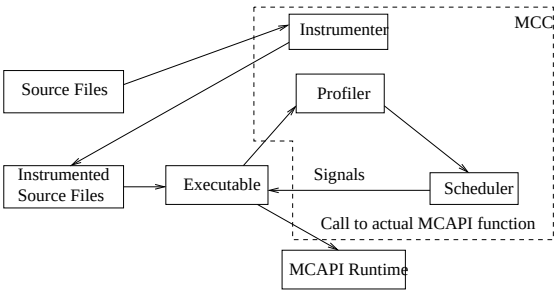

Fig. 3. MCC workflow

The scheduler explores all the independent thread steps in a single non-commutative canonical order while commuting all dependent co-enabled thread steps resulting in the exploration of a reduced state space that is a valid partial order reduction of the complete state space. However, MCC executes a dynamic POR that is considerably different from the one taken in classical shared memory in-order execution languages. As mentioned before, MCC adopts POE that is well suited for message passing based out-of-order execution semantics like that of MCAPI. The MCC scheduler also accommodates receive non-determinism by delaying (dynamically re-ordering) the processing of receive calls until all sends that can potentially match the receives are dynamically discovered. Each such send-receive match is explored in separate runs of the program (these matches form the *ample-sets [12]*).

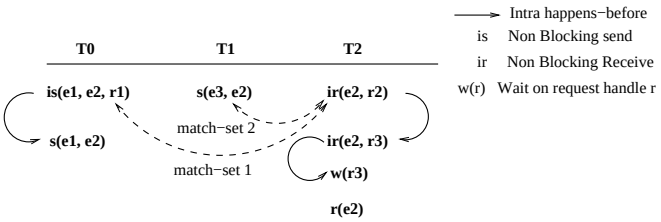

Fig. 4. Example with Non-blocking requests

B. MCC preliminaries

We describe the terminology that we use in forthcoming sections of the paper which will help in elucidating the internals of the MCC scheduler later in Section III-D.

match-sets: A match-set is a collection of operations that are signaled to the runtime together. For instance, a matching send and a receive call forms a match-set. These match-sets can also be singleton sets, for example, a wait call whose corresponding non-blocking call has drained/filled the intended message buffer. The match-sets for the example shown in Figure 2 are $\langle send(ep1, ep2), recv(ep2)\rangle$, $\langle send(ep3, ep2), recv(ep2)\rangle$.

ample-set: Ample set is the collection of match-set entries computed at each decision point. A decision point is where the status of all the threads is blocked and the scheduler has to decide on moving a match-set to the runtime thereby unblocking the participating threads.

replay: The MCC scheduler remembers all the ample-sets computed in the first run of the program. It also notes all the match-set entries that were explored at each decision point. MCC re-runs the program from the initial state and explores a different execution scenario by selecting the untried choice of match-set from the ample-set at every decision point.

Intra happens-before ordering: Intra happens-before (Intra-HB) ordering is a partial order that is established among the operations of a thread. All blocking operations of a thread are Intra-HB ordered. The Intra-HB rules for non-blocking send, non-blocking receive, and wait are the following:

- Two non-blocking sends are happens-before ordered in program order if they target the same destination endpoint and are issued from the same source endpoint.
- Two non-blocking receives are happens-before ordered in program order if they have the same receiving endpoint.
- A non-blocking send and its wait are happens-before ordered in program order. Similarly, a non-blocking receive and its wait are happens-before ordered in program order.

C. MCC Scheduler explanation through an example

The MCC scheduler unlike the ISP scheduler does not perform dynamic re-writing because MCAPI does not provide specific source point receives; meaning that one cannot designate where one would like to receive from. The scheduler is able to perform dynamic re-ordering of calls by first discovering all pending calls and then issuing matched calls sequentially to the run time and inserting waits when needed in non-blocking semantics. While an MCAPI node (i.e. a thread w.r.t. the reference implementation) would issue the calls in program order, the MCC scheduler can permute the order of these calls without introducing any new behaviors in the program.

Consider the example shown in Figure 5 where the MCC scheduler re-orders the calls. Threads T0, T1 and T2 are blocked at the *w(r2), s(e2, e1)* and *r(e3)* calls respectively. The enabled transitions are *ir(e1, r1), is(e1, e3, r2), s(e2, e1)* and *r(e3)*. The match-sets formed by the scheduler at this point are $\{\langle is(e1, e3, r2), r(e3)\rangle\}$ and $\{\langle ir(e1, r1), s(e2, e1)\rangle\}$. As the wait call for *ir(e1, r1)* is not yet seen, the *ir(e1,r1)* call is not obliged to finish before *is(e1, e3, r2)* call.

```
T0: ir(e1, r1); is(e1, e3, r2); w(r2);
T1: s(e2, e1);
T2: r(e3); s(e3, e1);
```

Fig. 5. Re-ordering Example

Note that signaling the match-set $\{\langle is(e1, e3, r2), r(e3)\rangle\}$ to runtime enables the *s(e3, e1)* call which is another potential sender to the call *ir(e1,r1)*. Hence, signaling the match-set $\{\langle ir(e1, r1), s(e2, e1)\rangle\}$ to the runtime before the match-set $\{\langle is(e1, e3, r2), r(e3)\rangle\}$ would lead to incorrect verification results. Noting this fact, the scheduler should signal a go-ahead to *is(e1, e3, r2)* call first thus permuting the issue order different from the program order.

Figure 4 illustrates a program with non-blocking requests. Intra-HB orderings are also shown that respect the rules mentioned in the section III-B. Some important observations regarding the example are:

- Thread T0's non-blocking and blocking sends have the same source and destination endpoints. Thus, they are Intra-HB ordered. Since the first non-blocking send is obliged to finish before the second send, it would not change the semantics of the program if we throw in a wait call corresponding to the first call. Hence, the operations can be alternatively viewed as *is(e1, e2, r1); w(r1); s(e1, e2);*
- Using a similar argument as before, the operations in thread T2 can be alternatively viewed as *ir(e2, r2); w(r2); ir(e2, r3); w(r3);*.

Figure 6 illustrates an interleaving scenario as a time-line based sequence of message interactions between the scheduler and the threads of an MCAPI user program (from Figure 4). The user program is branched off as a separate thread under the controlled environment of the scheduler. The main thread of the instrumented program issues thread create calls which when signaled to go-ahead by the scheduler, create threads T0, T1, and T2. Note that the main thread blocks at the first *thread_join* call. Threads T0, T1, and T2 are all blocked at their respective *thread_start* calls.

The reason to have a *thread_start* call is explained in Section III-D.

The scheduler unblocks the threads T0, T1, and T2 after ascertaining a count of the total number of threads alive in the system. The threads continue to run and issue calls until they have hit their fence operations (blocking calls). At this point the scheduler has seen the following operations: (i) *is(e1, e2, r1)* and *s(e1, e2)* from T0; (ii) *s(e3, e2)* from T1; and (iii) *ir(e2, r2), ir(e2, r3)* and *w(r3)* from T2. The list of enabled transitions is $\langle is(e1, e2, r1),\ s(e3, e2),\ ir(e2, r2)\rangle$. The scheduler has come across a decision point and subsequently forms match-sets from the list of enabled transitions. From Figure 6, it is apparent that the scheduler picks $\langle is(e1, e2, r1),\ ir(e2, r2)\rangle$ and signals the threads T0 and T2 to proceed. At the next decision point (when all threads have hit their fence operations), the enabled transitions are (i) *s(e1, e2)* from T0; (ii) *s(e3, e2)* from T1; and (iii) *ir(e2, r3)* and *w(r3)* from T2. The scheduler then forms the match-sets and decides to give the go-ahead to T1's *s(e3, e2)* call and T2's *ir(e2, r3)* call. Note that issuing the *s(e3, e2)* call into the runtime before *ir(e2, r2)* call has completed can cause the sends *s(e3, e2)* and *is(e1, e2, r1)* to race in the MCAPI runtime. This can potentially break the correctness of the verification process of the scheduler, wherein the scheduler decided one match-set to manifest at runtime however, the runtime instead picked another match-set. To make sure a deterministic match occurs at the runtime, the scheduler identifies a race and spin-loops until the *ir(e2, r3)* call completes before signaling a go-ahead to the next match-set. The box in the timing diagram of Figure 6 represents this spin-loop. The next match-set is signaled in to the runtime once the request *ir(e2, r3)* has completed

The main thread unblocks following the completion of the *thread_end* calls and the program runs to completion.

D. MCC Scheduler Algorithm

Figure 7 in Section III-D explains the working of the scheduler. The MCC scheduler works under certain assumptions. It assumes that all threads of the system are created at the outset of the program. The MCC scheduler must know the total thread count in the system to determine when all threads have blocked. As such, MCC count threads as they are created by the main thread, and blocks them on their thread-start calls until the main thread either invokes an MCAPI call or a thread join call. At that point, MCC assumes the total number of threads to be those already created and starts all the created threads running. After ascertaining the thread count, the scheduler liberates all the blocked threads (line 17) and starts receiving transitions from all runnable threads until the next decision point is hit. Note that if a thread issues a

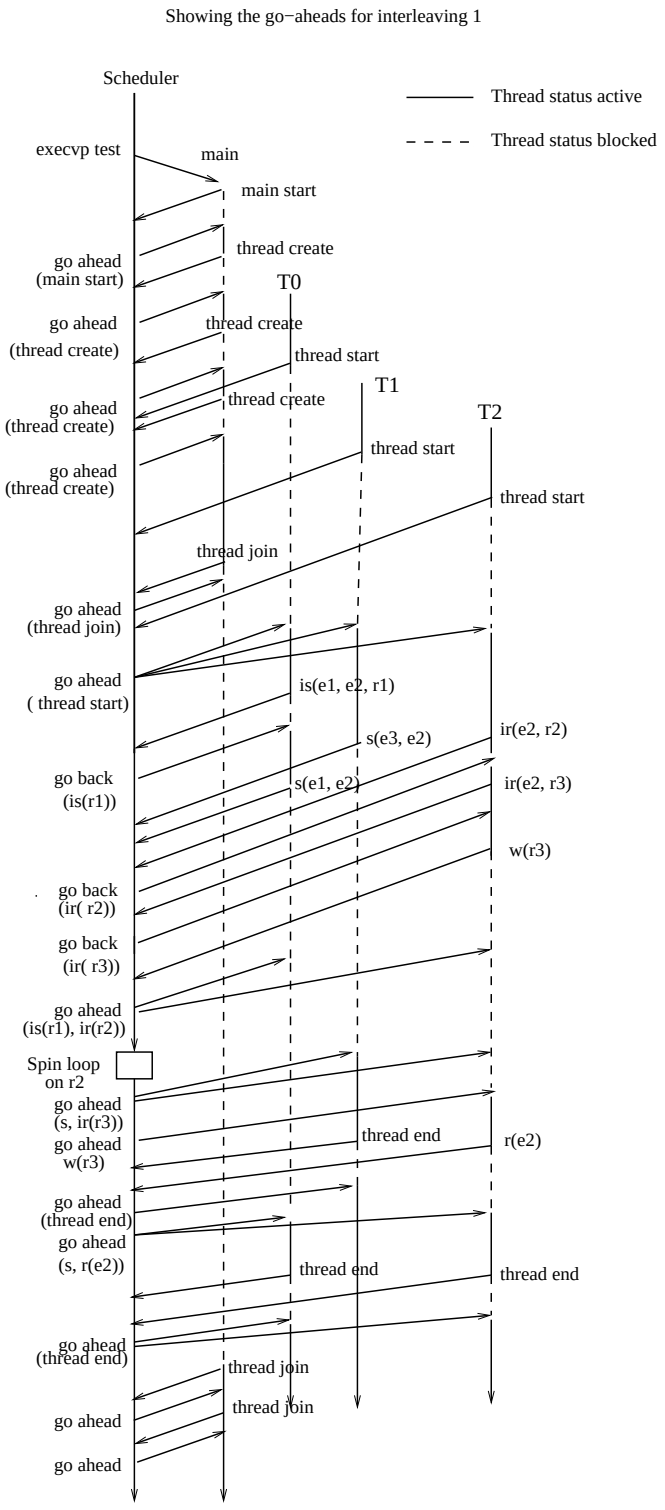

Fig. 6. Interactions of the scheduler with the example from Figure 4

```
1:  GenerateInterleaving( ) {
2:      while (1) { // Computes the total number of threads alive
3:          t_i = receive_transition ();
4:          if (t_i is thread_create) {
5:              num_threads++;
6:              signal go-ahead to thread_of(t_i);
7:          }
8:          if (t_i is thread_join || t_i is MCAPI communication call by thread
                "main") {
9:              signal go-ahead to thread i;
10:             break;
11:         }
12:         if (t_i is thread_start) {
13:             update the status of thread i to blocked;
14:         }
15:     }// while (1) ends here

16:     count = num_threads;
17:     signal go-ahead to all the blocked threads;

18:     while (count) { // till no more threads are alive
19:         for each (runnable thread i) {
20:             t_i = receive_transition from thread i;
21:             update transition_list of thread_of (t_i) in the S_curr;
22:             if (t_i is of blocking_type) {
23:                 update the status of thread i to blocked;
24:             }
25:             if (t_i is of type thread_end) {
26:                 count --;
27:             }
28:         }
            // All threads are blocked here
29:         while (no thread is runnable) {
30:             find_matchset ();
31:             unblock the threads owning transitions in the above match-
                    set;
32:         }
33:     }// while (count) ends here
34: }

35: check_for_runtime_race( ) {
36:     if (any t_i ∈ current_match-set races with non-blocking call from
            prev_match-set) {
37:         while (non-blocking call is completed); {
38:         }
39:     }
40: }

41: find_matchset( ) {
42:     Store the computed match-sets in ample_set of S_curr;
43:     if (ample_set is not empty) {
44:         for each (t_i in head_element of the ample_list) {
45:             check_for_runtime_race();
46:             give a go-ahead to thread_of (i);
47:         }
48:         remove head_element from ample_set;
49:         copy the ample_set in S_next;
50:         return;
51:     }
52:     flag that a deadlock found;
53: }
```

Fig. 7. MCC scheduler algorithm

thread_end call, the thread count of the system is decremented (lines 18-28).

Once a decision point is hit, the scheduler then computes the match-sets from a list of enabled transitions. It then selects one match-set and liberates the participating threads in that match-set (lines 29-32). A match-set consists of either a send-receive call pair, or a single entry comprising a wait call. The enabled transitions are computed with the help of the Intra-HB relationship that is maintained for each state of the scheduler. The priority order for evaluating these match-sets is the following: (i) enabled wait call (ii) and then the send-

receive match-set.

The MCC scheduler also handles *get_endpoint* and *create_endpoint* calls. When a thread issues a *create_endpoint* call, the scheduler looks to see if any blocked thread (on *get_endpoint* call) was waiting for it. If so, the *create_endpoint* call and the blocked *get_endpoint* call are both signaled to go-ahead. If that is not the case, then the scheduler stores the created endpoint in an auxiliary table. When the scheduler encounters a *get_endpoint* call then it first looks up the table of created endpoints. It blocks the thread if the sought endpoint is not created. Otherwise, *get_endpoint* call is immediately signaled to go-ahead.

Every decision point advances the state of the scheduler. The match-sets for a state under exploration are stored in a separate data structure (*ample-set*). Every state has an ample-set associated with it. One entry is selected from this ample-set for the go-ahead. Subsequently, the match-set entry that has been recently liberated is removed from the ample-set. The updated ample-set is then copied to the next state. Note that only the first interleaving builds the per-state ample-set. The scheduler declares a deadlock in the code if at a state the ample-set is found to be empty while there are still runnable threads in the system (lines 41-52).

A safety check is performed before the participating threads can be given a go-ahead. This safety check ensures that a deterministic match manifests at runtime and the transitions of the match-set in the current state (S_{curr}) do not race with the transitions from the match-set in the previous state (S_{prev}). In the case when a race is found then the scheduler spin-loops until the racing transition from S_{prev} is completed by repeatedly testing the request handle of the racing transition. Only after the completion of the racing transition is the current match-set processed (lines 35-40). Later if a wait call is observed by the scheduler for the completed racing transition, it is still issued to the runtime, however, it will return immediately.

The procedure *GenerateInterleaving* is called in a loop until there are no more replays to be performed. The decision whether to perform a replay is made by inspecting the ample-set of the visited states in the stack. If for each state the ample-set is found to be empty then the scheduler has explored all the relevant interleavings.

IV. Results and Concluding Remarks

We have developed the first dynamic verification engine for MCAPI user applications that currently handles blocking and non-blocking connection-less communication constructs of the MCAPI reference implementation. Since no publicly available benchmark using MCAPI is currently available, we tested MCC successfully on small test examples constructed by ourselves. For instance, the example program from Figure 4 was verified in 3 interleavings in a fraction of a second. We are currently working to extend MCC to support the full set of MCAPI calls. Future works involves exploring solutions to verify programs that have subtle bugs, for instance, data-races in unison with the MCAPI non-determinism. We acknowledge Jim Holt from Freescale for his help on this work.

References

[1] http://www.multicore-association.org.
[2] http://www.mcs.anl.gov/mpi/
[3] http://www.llnl.gov/computing/tutorials/pthreads/
[4] Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. CAV 1997, 476-479.
[5] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *POPL*, 110–121, 2005.
[6] Java Pathfinder. http://javapathfinder.sourceforge.net/.
[7] Yu Yang, Xiofang Chen, Ganesh Gopalakrishnan, R.M. Kirby. Distributed Dynamic Pratial Order Reduction Based Verification. SPIN 2007, 58-75.
[8] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. PPoPP 2008. 285-286.
[9] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI 2007, 446-455.
[10] www.cs.utah.edu/formal_verification/mediawiki/index.php/MCAPI/
[11] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. CAV 2008, 66-79.
[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
[13] Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. *MCC: A runtime verification tool for MCAPI applications*. accepted in FMCAD 2009.
[14] Gerard J. Holzmann The Model Checker Spin, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.