HEURISTICS FOR EFFICIENT DYNAMIC VERIFICATION OF MESSAGE PASSING INTERFACE AND THREAD PROGRAMS

by

Wei-Fan Chiang

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2011

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Wei-Fan Chiang**

has been approved by the following supervisory committee members:

Ganesh Gopalakrishnan	, Chair	**April. 07, 2011** Date Approved
Mary Hall	, Member	**April. 07, 2011** Date Approved
Rajeev Balasubramonian	, Member	**April. 07, 2011** Date Approved

and by **Al Davis** , Chair of

the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Concurrent programs are extremely important for efficiently programming future HPC systems. Large scientific programs may employ multiple processes or threads to run on HPC systems for days. Reliability is an essential requirement of existing concurrent programs. Therefore, verification of concurrent programs becomes increasingly important. Today we have two significant challenges in developing concurrent program verification tools: The first is scalability. Since new types of concurrent programs keep being created, verification tools need to scale to handle all these new types of programs. The second is providing formal coverage guarantee. Dynamic verification tools always face a huge schedule space. Both these capabilities must exist for testing programs that follow multiple concurrency models.

Most current dynamic verification tools can only explore either thread level or process level schedules. Consequently, they fail to verify hybrid programs. Exploring mixed process and thread level schedules is not an ideal solution because the state space will grow exponentially in both levels. It is hard to systematically traverse these mixed schedules. Therefore, our approach is to determinize all concurrent APIs except one API whose schedules will then be explored.

To improve search efficiency, we proposed a random-walk based heuristic algorithm. We observed many concurrent programs and concluded some common structures of them. Based on the existence of these structures, we can make dynamic verification tools focusing on specific regions and bypassing regions of less interest. We propose a random sampling of executions in the regions of less interest.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Almost all science and engineering research in the world is conducted with the aid of concurrent programs. Therefore, concurrent software reliability is increasingly important. Undetected bugs in software may cause data loss or even serious safety issues. The examples of the disaster caused by software bugs are Therac-25 tragedy [18], the dysfunction of the Mars Pathfinder [15], and the 2003 North American Blackout [10]. Ideally, software developers should have some efficient and systematic tools to help the verification of their software.

1.1 Specific Focus

In this thesis, we are interested in verifying threaded MPI hybrid programs, which are a combination of MPI programs and threaded programs. MPI programs are widely used at scale on large cluster machines. However, purely message passing programs create many redundant memory copies of messages. On the other hand, threaded programs use shared memory as their communication channel that optimizes the memory copies. However, threaded programs are difficult to scale. The combination of these two types of concurrent programs can achieve the benefits of these individual pure approaches. Threaded MPI programs can utilize hardware, eliminate redundant memory copies, and provide high throughput. They can effectively utilize hybrid concurrent hardware such as multicore processors used as super computing nodes. Recent supercomputers such as Tianhe-1A [12] and CRAY-XT5 [11] are comprised of thousands of multicore processors connected by efficient message passing networks. MPI programs distribute jobs on such supercomputers, and for each job on a multicore processor, the thread library parallelizes the job into concurrent tasks. Therefore, threaded MPI hybrid programs are a very good fit for utilizing supercomputers. In this thesis, for convenience, we use the term *hybrid programs* to refer to *threaded MPI hybrid programs*.

Verifying hybrid programs is important because of the growing usage. The whole computation could be distributed on many processors and take a long time to generate

results. Errors can waste days of supercomputing time.

We can catalog errors in concurrent programs into two categories: schedule independent and schedule dependent. Schedule independent errors are usually caused by the improper usage of concurrent such as incorrect arguments, buffer usage errors, and type matching errors. Schedule dependent errors are caused by certain interleavings of API calls. The buggy interleavings of calls are usually unexpected, and hard to reproduce. Deadlock is one such error.

Detecting and reproducing schedule dependent errors in concurrent programs is difficult. The traditional "printf" strategy is no longer suitable for debugging because in most cases, the bugs will only appear in a few schedules that rarely happen. In this thesis, we focus on detecting schedule dependent errors in hybrid programs.

1.2 Dynamic Verification on MPI Programs

Several formal verification approaches for concurrent programs have been proposed. One of these is dynamic verification. Dynamic verification methods have proven effective for detecting schedule dependent bugs in real-world concurrent programs [21]. These tools explore different concurrent schedules by repeatedly executing the real applications and enforcing various relevant schedules. Dynamic verification tools such as CHESS [23], Inspect [32], and JPF [8] have been proposed for testing concurrent programs. In this thesis, we will use our MPI program dynamic verification tool, ISP, to test hybrid programs.

Figure 1.1 shows a simple MPI program without the computational code typically used between message passing statements. The function $recv$ denotes the receiving operation. The function $send$ denotes the sending operation. The function $barrier$ denotes the barrier operation, which synchronizes all processes.

```
1: if id == 0 {
2:     recv(*)
3:     recv(*)
4:     recv(*)
5:     barrier()
6: else if 1 ≤ id ≤ 3 {
7:     send(0)
8:     barrier()
9: }
```

Figure 1.1. Example Program

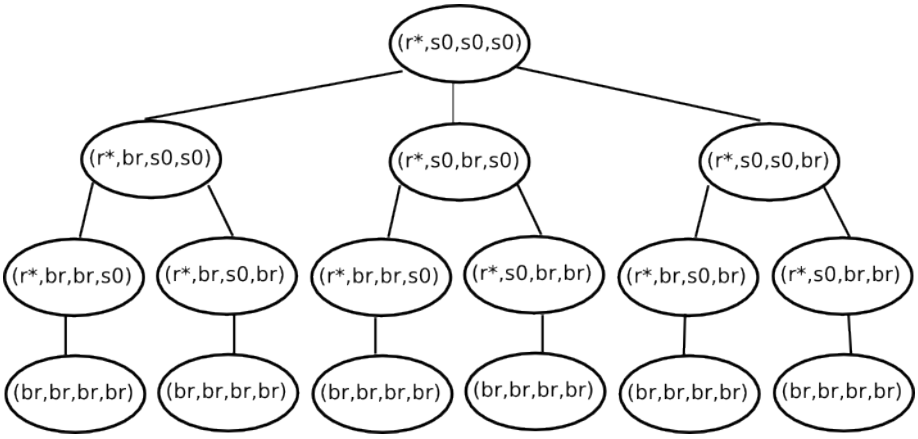

Figure 1.2. The State Space of the Example Program

Figure 1.2 shows the exploration tree of schedules of this program. The label ri denotes $recv(i)$ and si denotes $send(i)$. The state, (v_0, v_1, v_2, v_3), denotes the next operation of each process.

Figure 1.2 shows the six possible schedules in this MPI program. Each path from the root node to one of the leaf nodes denotes one possible schedule. Ideally, a dynamic verification tool must attempt to traverse the entire tree and check every possible schedule.

1.2.1 ISP

In this section, we will briefly introduce our dynamic formal MPI program verifier, ISP [29]. An ideal concurrent program verification tool should meet several goals:

1. Guarantee coverage.
2. Eliminate redundant tests.
3. Cover the input space.
4. Provide a intuitive user interface within popular frameworks.

ISP [29] is a successful MPI program formal verifier which meets these goals with the exception of the third. ISP employs a special verification scheduler with partial order reduction to reach the first two goals. ISP meets the last goal by integration within the Eclipse framework [2]. To meet the third goal, typically we need a symbolic analysis process.

ISP contains two main components: profiler and scheduler. The scheduler of ISP intercepts MPI calls and permutes the order among them to enforce interleavings. The profiler redefines the MPI functions and uses MPI's profiling mechanism (PMPI) [20] instead. For each redefined function, it sends a TCP socket message to the scheduler through which the scheduler can keep track of MPI calls. The scheduler collects MPI operations by receiving sockets message from the processes. This process continues until each process encounters a *fence operation* [30]. At each fence point, the scheduler carries out the POE algorithm [29] to explore the state space.

1.2.1.1 Stateless Exploration and Deterministic Replay

ISP employs stateless *Depth First Search* [7] possible for terminative programs. However, this DFS exploration is based on an important assumption: **the tested program must be deterministically replayable**. Specifically, nondeterministic behaviors other than in the message passing code are not allowed.

1.2.1.2 Fence Point

Before we introducing *fence points*, we need to explain what a *fence operation* is. A *fence operation* is defined as an MPI operation that cannot complete after any other MPI operation that follows it [30]. Examples are *MPI_Wait*, *MPI_Barrier*, etc. Once all processes reach their *fence*, ISP will fully know the senders, which may match a wildcard receive. The state that all processes reach their *fence* is called a *fence point*. In Figure 1.2, every state is a fence point.

1.2.1.3 Ample Set

When every process reaches its *fence point*, the ISP scheduler will form a set of send-receive matches for the POE algorithm to explore the state space. The set of send-receive matches is the *ample set* [29] at this *fence point*. The *ample set* denotes the minimal and sufficient choices for paths to explore the state space. For example, considering the root state in Figure 1.2, the *ample set* of this state is $\{(p_1, p_0), (p_2, p_0), (p_3, p_0)\}$. The pair (p_i, p_j) denotes a match that the sender is process i and the receiver is process j. **Ample sets define required explorations at each state**.

1.3 Thesis Statement

We have two goals for this thesis:

- Deterministically replay MPI hybrid programs.
- Reduce the number of schedules examined while preserving high coverage.

Deterministic replay of MPI hybrid programs can be achieved by developing a record/replay mechanism for thread programs and a scheduler for the message passing aspects. Furthermore, using heuristics we can cut down the number of schedules examined while achieving significant coverage. These approaches for the two goals are presented in two isolated parts in this thesis.

1.3.1 Deterministic Replay of Hybrid Programs

We investigate the challenges of deterministically replay hybrid programs and identify the problem of hybrid schedule nondeterminism. We propose a schedule determinizing solution for dynamic verification tools testing hybrid programs. For mixed thread level and process level schedules in hybrid programs, our solution is determinizing thread level schedules while dynamic verification tools are exploring process level schedules. Our approach hides the nondeterminism of thread level schedule for dynamic verification tools as what they expect to be.

1.3.2 Focus Plus Randomize Sampling Heuristic Search Space Reduction

Schedules of hybrid programs are mixed by thread level and process level schedules. A hundred-line hybrid program can generate a large number of schedules. However, to the best of our knowledge, even process level schedules of pure MPI programs cannot be gracefully traversed by any current dynamic verification tool. Therefore, we stay in pure MPI programs to deal with the schedule search space explosion problem. We observed several MPI programs and concluded some common program structures: loops, tested blocks, and symmetric code blocks. We present a focused plus randomized algorithm to reduce the schedule search space based on these structures in MPI programs.

1.3.3 Implementations and Evaluations of Our Solutions

In order to evaluate our ideas, we implement our solutions in our dynamic formal MPI program verification tool, ISP. We generate two experimental versions of ISPs.

Each of them realizes the solution of one part of our thesis. Our results show the effectiveness of our ideas.

1.4 Related Work

In this section, we briefly survey some successful dynamic verification tools for threaded programs. To our knowledge, ISP is the first and only dynamic verification tool for MPI programs.

1.4.1 Java PathFinder

The original Java PathFinder (JPF) described in [8] is a translator from a java program to Promela, a modeling language of the SPIN model checker [9][16]. In fact, Java PathFinder works like a Swiss army knife of java program verification. For example, in [24], the authors use JPF to exhaustively search for the possible paths in tested java programs. It dynamically enumerates all the path conditions of these paths and formulates these path conditions as formulas for SMT solvers. Consequently, they can verify the reliability of the java programs. The original Java PathFinder itself is not a "real" model checker or verifier. However, there are many extensions [14], such as a symbolic execution extension for the Java Path Finder, that make it feasible to perform different verification tasks. Our verification tools should find it easy to add external components to deal with novel programming styles or APIs.

1.4.2 CHESS

CHESS is a multithreaded program verification tool. It can test and debug on user-mode Win32 programs, .NET programs, and Singularity [17] applications. CHESS uses a dynamic verification method to verify the programs. Given a program, CHESS will repeatedly execute it and explore different schedules in each round. To capture and control the thread schedules, CHESS requires a wrapper layer between the tested program and the concurrency API. The main technique CHESS employs to reduce search space is controlling the number of preemptions in thread schedules. The thread schedules containing fewer preemptions will be assigned higher priority to explore and test. The intuition behind this strategy is that many bugs are exposed under a few preemptions occurring in the multithreaded programs.

1.5 Contributions

In this thesis, we propose a hybrid program verification helper for ISP and implement it to successfully verify MPI/Pthread programs. We also propose a focus plus random sampling heuristic to improve ISP's performance. While we demonstrate the efficiency of the focus plus random sampling (FPRS) testing on pure MPI programs, it can be also used on hybrid MPI programs without any alterations. We acknowledge the help of Grzegorz Szubzda in identifying the hybrid program verification issues in the context of Eddy Murphi.

CHAPTER 2

DYNAMIC VERIFICATION ON HYBRID PROGRAMS

In this chapter, we introduce a hybrid program dynamic verifier. We will demonstrate our idea on a nontrivial case study. The case study we choose is an MPI/Pthread hybrid program called Eddy Murphi [1] [19]. When we attempted to verify Eddy Murphi using the nonhybrid version of ISP, to our surprise, we found that ISP, which had previously verified a number of large applications, crashed. In the following section, we present the underlying reasons and our general solution applicable to a reasonably large class of hybrid MPI/thread programs.

2.1 Overview of the Nondeterministic Replay Problem

2.1.1 A Case Study on Eddy Murphi

Eddy Murphi [19] [1] is a parallel and distributed model checker. It essentially implements a BFS approach algorithm to explore the state space. Eddy Murphi distributes states to nodes and defines the *home node* of each state through a simple hash function over the state vector. Each node supports an MPI process to explore the state space comprised of a worker thread and a communicator thread. Each state has an *owner id* which is the home node of that state. The worker and the communicator of a node share FIFO queues (Q in Figure 2.1) of states. The worker dequeues a state from the FIFO queue Q and checks the ownership of all its successor states. For a successor state homed locally, the worker will enqueue it into the Q. Otherwise the worker inserts the state into an outbound communication queue (*CommQueue* in Figure 2.1). The communicator thread (*CT* in Figure 2.1) is responsible for sending out states. It scans through all queues in *CommQueue* and sees if the number of states in a certain queue is over a defined bound. The communicator thread also receives states by issuing MPI wildcard receive calls.

The pseudo codes in Figure 2.2 show how workers and communicators access shared structures. Note that the global variables, the shared structures, Q and *Com-*

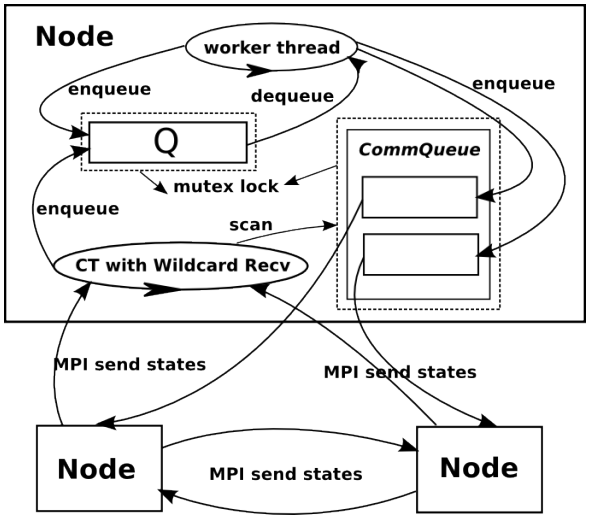

Figure 2.1. The Architecture of EddyMurphi

mQueue are protected by Pthread locks. In Figure 2.1, these structures are marked by dash-line frames.

2.1.2 Nondeterministic Replay Problem

We mentioned that ISP crashed while testing Eddy Murphi. The error message given by ISP shows that it failed to collect the same ample set for the same state in different rounds of execution. In other words, there were some MPI wildcard receive calls matched with different sets of senders.

In Eddy Murphi, the MPI wildcard receive and send calls are all issued the communicator thread. For issuing the MPI send calls, the communicator thread of a process visits the shared structure *CommQueue* periodically. If the number of the pending states in a certain queue is greater than a defined bound, the communicator sends these states out. For issuing the MPI receive calls, the communicator thread probes the incoming messages first and receives if there is any sender. The key is that the communicator thread sending the states depends on the loading of *CommQueue*. The loading of the queues in *CommQueue* depends on how many states the worker thread enqueued (the loading of *Q*). Therefore, we conclude that the race between the communicator thread and the worker thread decides what MPI calls are issued and their order. This underlies our inability to deterministically replay Eddy Murphi.

To make ISP capable of verifying hybrid programs like Eddy Murphi we must achieve these goals:

- ISP must explore the space of nondeterministic receives of the communicator

```
Worker Thread
1.  ParBFS () {  // worker thread
2.       pthread_create(CommThread);
3.       /* some initializing works */
4.       while (!ParTerminate()) { /* main loop */
5.           s = Dequeue(Q);
6.           foreach (s_next) in next(s) {
7.               if (!CheckState(s_next)) {
8.                   /* test if terminal */ }}
9.       } /* end of main loop */
10.      pthread_join(); }
11. CheckState (state s) {
12.     owner_rank = owner(s);
13.     if (owner_rank == my_rank) { /* my own state */
14.         Enqueue(Q, s); }
15.     else {
16.         Enqueue_line(CommQueue[owner_rank]); }}

Communicator thread
1.   CommThread () {  // communication thread
2.       while (true) {
3.       ProcMess(); /* processes the incoming messages */
4.       /* handles termination */
5.       DoSends();
6.       /* handles termination probing */ }}
7.   ProcMess() {
8.        /* may receive the terminating message */
9.        S = MPI_Recv(ANY_SOURCE, state);
10.      foreach state s in S
11.          Enqueue(Q, s); }
12.  DoSends () {
13.      foreach node n different from my_rank {
14.          while (lines_ready(CommQueue[n])) {
15.              S = Dequeue_line(CommQueue[n]);
16.              MPI_Isend(S, n, state); }}}
```

Figure 2.2. Worker and Communication Threads

threads (see Figure 2.1 where we show “CT with Wildcard Recv”).

- ISP must not be “confused” by the Pthread schedules that may vary from replay to replay with respect to lock access (see Figure 2.1 where we show “mutex lock”).

We can conclude we need to *deterministically replay the race between threads in the process* to solve this problem. To clarify, we record and replay the race condition. Since ISP is a stateless dynamic verifier, our record-replay mechanism is also stateless so that it only needs the record of the last round to control the race in the current execution.

2.2 Determinizing Solution

2.2.1 DR Event

A "DR event" denotes a decision made by ISP. ISP scheduler will collect MPI calls from all processes and wait for all processes to reach their *fence points*. Once ISP realizes that all processes have reached fences, it chooses a source from the set of potential matching senders to match a wildcard receive and rewrites the wildcard receive into a specific receive, thus determinizing the MPI schedule from that point. For our race replay daemon, we call this decision, meaning the matching of one eligible reader with a nondeterministic receive, a *DR event*. The *DR event* plays an important rule for our deterministic replay mechanism that will be explained in the following sections.

2.2.2 ISP with Record/Replay Daemon

Here we provide an overview to show how ISP works with the external Race Replay Daemon. We call it an "external" race replay daemon because our additions to ISP do not fundamentally change the algorithm of ISP for checking the MPI programs. Installing the daemon into ISP did not require significant modifications of the ISP source code. For ISP to cooperate with the daemon, it has to:

- Start up the daemon before the testing processes are created.
- Set a flag when there is a *DR event*.
- Restart the daemon before it restarts itself.
- Terminate the daemon before it terminates itself.

We believe that such a nonintrusive extension can make ISP a powerful MPI program verifier.

2.2.2.1 Our Solution Overview

We build Pthread call instrumentation facilities to record the order of accessing shared data structure and signaling conditional variables by creating our own versions of *pthread_create*, *pthread_mutex_lock*, *pthread_mutex_unlock*, *pthread_cond_wait*, and *pthread_cond_signal*. (We will introduce them in section 2.2.3.) These instrumentations can be easily automated. Our version of these functions will notify the race replay daemon and cooperate with the daemon to issue the original pthread calls at the "right"

timing. ISP collects MPI calls from all processes and waits for the processes to reach their *fence points*. Then ISP will raise a *DR event*. Our daemon sequentially probes the notifications (can also be seen as events) from threads and *DR events*. Then it records all these events into a log file. This is the *record mode* of the daemon.

With a log file of events for a round of execution, the daemon can also enforce the order of events according to the log. This is the *replay mode* of the daemon. For the deterministic replay of Pthread calls: (i) the daemon sends ACKs to threads calling Pthread routines in an order that matches the recorded order, or (ii) the daemon NACKs threads arrives in a different order.

DR events play an important role in our record/replay mechanism. It is critical for the daemon to switch from the replay mode to the record mode. In the initial run, since we do not have a log file, the daemon starts in record mode. During subsequent runs, the daemon starts in the replay mode and enforces the order of all events according to the log file generated by a previous execution. Note that the daemon still keeps recording the events even in replay mode. When it comes to replay a DR event, the daemon first sees if ISP has replayed the previous decision or not:

- If ISP chooses the same source as previously run for the wildcard receive, the daemon stays in the replay mode.

- When ISP pursues a new match at a fence point, the daemon switches to record mode. From this point to the end of this execution, the daemon stays in record mode and continues to generate a log file for the next run.

After ISP chooses a new match at a certain fence point, the previous run's log file will not be used subsequently. Therefore, the previous log file will be discarded. Such a record/replay mechanism is easily implemented by keeping only two log files: the previous one and the current one.

Figure 2.3 illustrates how we augment ISP with a daemon that helps record a previous Pthread schedule (recorded during first play to an MPI nondeterministic point) and enforce the recorded schedule (during a subsequent play to that MPI nondeterministic point). In Figure 2.3, each MPI process may create multiple threads. These threads notify the daemon when Pthread calls issued. The daemon controls the schedule of these Pthread calls by sending ACKs or NACKs. We will explain the protocol in later sections.

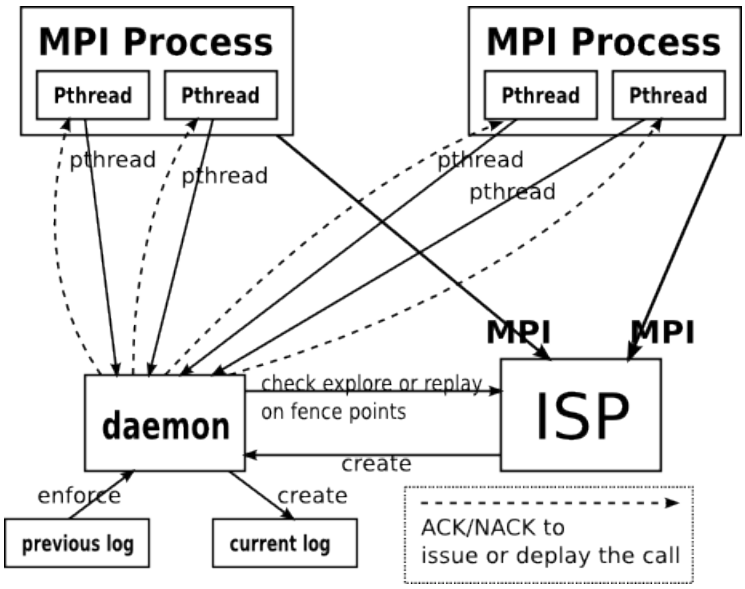

Figure 2.3. ISP-Daemon System

Note that every process in Figure 2.3 has only one thread called MPI. We make an assumption that the MPI threading level is MPI_THREAD_FUNNELED. That means there is only one thread in a process can issue MPI calls. A summary of assumptions we make is as follows:

- All read/writes are protected by mutual exclusion locks.
- The MPI threading level is MPI_THREAD_FUNNELED.
- Processes communicate only through MPI calls.
- Other API calls (besides MPI and Pthreads) are not allowed.
- The inputs provided by our test harness are deterministic.

While Eddy Murphi satisfies all these requirements, not all MPI-Pthread hybrid programs do. However, we believe that articulating such assumptions is a valuable contribution in that others may be tempted to relax (or strengthen) them to build better tools in an area where none exist.

2.2.3 The Instrumented Pthread Functions

In this section, we present how our version of Pthread functions works. We rewrite our version of functions: *create*, *mutex_lock*, *mutex_unlock*, *cond_wait*, and *cond_signal*. Before introducing them, we define a function *SendnRecv* which performs a socket *send*

followed by a socket *receive*. This function takes a message as the input and returns with the corresponding ACK or NACK.

2.2.3.1 pthread_create

Our version of *pthread_create* sends a *CREATE* message to the daemon. This message notifies the daemon that a new thread is going to be created. The daemon will always return ACK. Figure 2.4 shows the pseudo code of our *pthread_create*. It calls *SendnRecv* to send the notification to the daemon. Then it calls the original *pthread_create*. The arguments $args$ is given by the programmers for the original pthread creation used.

We also instrument the *MPI_Init* call that it will send a *BUILD* message to the daemon. This message is similar to the *CREATE*. It notifies the daemon that there will be a set of threads with the same process it created.

2.2.3.2 pthread_mutex_lock

Figure 2.5 shows our design of the new *pthread_mutex_lock* function. This function will first send the *TRY_LOCK* message to the daemon. This message informs the daemon which thread is waiting to enter a critical section. This information also helps the daemon to detect deadlock.

After the *TRY_LOCK*, the function will try to acquire the mutex lock by calling the original *pthread_mutex_lock*. Once it enters the critical section successfully (returns from the original *pthread_mutex_lock*), it will send another message, *ENTER_CS*, as a notification to the daemon. In the record mode, an *ACK* will always be replied. In the replay mode, the first *ENTER_CS* could receive either *ACK* or *NACK*. If the *NACK* is received, it means that "it is not my turn to enter the critical section." Then our *pthread_mutex_lock* will give up the lock and send a *TRY_AGAIN* message to the daemon. The thread calling our locking mutex function will be blocked by waiting for the reply of this message. Since we use the function *SendnRecv* to pass the message,

```
1: Our_pthread_ create(args) {
2:     SendnRecv(CREATE)
3:     return pthread_create(args)
4: }
```

Figure 2.4. Our pthread_create

```
1:  Our_pthread_mutex_lock(args) {
2:      SendnRecv(TRY_LOCK)
3:      returnv = pthread_mutex_lock(args)
4:      rcode = SendnRecv(ENTER_CS)
5:      if rcode = NACK {
6:          pthread_mutex_unlock(args)
7:          SendnRecv(TRY_AGAIN)
8:          returnv = pthread_mutex_lock(args)
9:          SendnRecv(ENTER_CS)
10:     }
11:     return returnv
12: }
```

Figure 2.5. Our pthread_mutex_lock

the thread will sends a message to the daemon and wait for the response. The daemon can block the thread by postponing the *ACK* or *NACK*.

The design of our "second-try" *pthread_mutex_lock* is nontrivial. We need to keep in mind that the thread should not care about the mode (record or replay) of the "ISP-daemon" system. We should not design two versions for these instrumented pthread calls. Therefore, the daemon needs to control the two different behaviors of these pthread calls by replying *ACK* or *NACK*. Meanwhile, we need to think about when the thread needs to send the message to the daemon. One option is sending the message before calling the original *pthread_mutex_lock* for entering the critical section. This option makes it very easy to replay thread schedules. We can just send *ACK* to *unblock* the threads with the recorded schedule. However, for recording the schedule, it becomes very hard to know the "real" order of accessing critical sections. Considering the case shown in Figure 2.6, we can see that the thread b could preempt the thread a after a calls the *SendnRecv* and enters the critical section. In this case, since the daemon receives the message from a first, it will think a is entering the critical section before b. However, it is not true. To solve this situation, we choose to send a message to the daemon after entering the critical section. The difficulty in this approach is to replay the thread schedule. A thread could acquire the mutex lock in the wrong order while replaying. To solve this case, the daemon will check the validity of it on the first *ENTER_CS*.

```
1: SendnRecv(ENTER_CS)
2: pthread_mutex_lock
```

Figure 2.6. Sending Message Before Entering Critical Section

```
1: Our_pthread_mutex_unlock(args) {
2:     SendnRecv(RELEASE)
3:     return pthread_mutex_unlock(args)
4: }
```

Figure 2.7. Our pthread_mutex_unlock

Once the daemon realizes that there is a thread acquiring the mutex lock out of order, it will send *NACK* to it. The thread will give up the lock after receiving a *NACK* and wait after sending *TRY_AGAIN*. The daemon will send *ACK* to the threads to grant the access to the critical section according to the log. Once the thread returns from the *THREA_TRY_AGAIN_INFORM*, it is safe to enter the critical section.

2.2.3.3 pthread_mutex_unlock

Figure 2.7 shows the code of our *pthread_mutex_unlock*. In our *pthread_mutex_unlock*, we send the notification to the daemon before we call the original unlock function. The reason why we do not send the message after calling the unlock function is that the daemon needs to record the "real order" of the accessing of the critical section. Considering the case shown in Figure 2.8, thread x releases the lock before sending notification. Suppose we are in the record mode and thread x is going to unlock the mutex lock m while thread y is going to lock m. If thread x releasing the lock m on line 1 and preempted by thread y before executing line 2, then thread y will acquire the lock m successfully and send the message to the daemon before thread x sends its message. Since the daemon will record the message with the order of their coming (will be explain in detail later), there will be a unreasonable sequence of schedule in the record: *ENTER_CS* $\rightarrow$ *RELEASE*. Such unreasonable sequences will cause problems while replaying the schedules. Instead of paying extra effort to fix this error in the log file, sending the message first in our *pthread_mutex_unlock* can easily solve the problem.

2.2.3.4 pthread_cond_wait

In our conditional wait or signal functions, we do not call the original waiting or signaling function. The original *pthread_cond_wait* releases an assigned mutex lock and then waits for another thread to signal an assigned conditional variable. However, it is hard to control the wait-signal relationship without modifying the underlying MPI implementation. Considering our *pthread_cond_wait* shown in Figure 2.9, line 2 and 3

```
Thread X:
1.  pthread_mutex_unlock(&m);
2.  SendnRecv(RELEASE);
Thread Y:
3.  pthread_mutex_lock(&m);
4.  SendnRecv(ENTER_CS);
```

Figure 2.8. Sending the Message After Calling the Unlock Function

```
1: Our_pthread_cond_wait(lock, cond) {
2:     SendnRecv(WAIT)
3:     pthread_mutex_unlock(lock)
4:     SendnRecv(WAKEN)
5:     return Our_pthread_mutex_lock(lock)
6: }
```

Figure 2.9. Our pthread_cond_wait

are used to release a mutex lock. We send the *WAIT* to the daemon instead of *RELEASE*. The message *WAIT* not only informs the daemon the lock released but also gives extra information to help the daemon maintain the wait-signal relationship. The daemon will track the waiting order and enforce the waken order. Line 4 sends the *WAKEN* message to the daemon. The daemon will send back the ACK on the time for this thread to wake up. Finally, after returning from line 4, the thread calls our version of *pthread_mutex_lock* to try to acquire the mutex lock back. An important issue in our *pthread_cond_wait* is that we need to make two actions, lock releasing and signal waiting, into one atomic action. We use our daemon to combine them. We will explain this in a later section.

2.2.3.5 pthread_cond_signal

The design of our *pthread_cond_signal* is similar to our unlock function for the mutex lock. Consider Figure 2.10, we send a *SIGNAL* to the daemon, the daemon will find a waiting thread to wake up in deterministic order. For the original *pthread_cond_signal* in line 3, it actually wakes nobody up! This is because that there is nobody calling the original conditional wait. It is just for returning value.

```
1: Our_pthread_cond_signal(args) {
2:     SendnRecv(SIGNAL)
3:     return pthread_cond_signal
4: }
```

Figure 2.10. Our pthread_cond_signal

2.2.4 ISP-Daemon Communication Protocol

ISP and the daemon associate with each other by producing and consuming the *DR events*. We define a shared variable *_DR_Event_Flag* to record the state of the *DR event*. ISP and the daemon access this shared variable in a "producer and consumer" approach. ISP raises a *DR event* by calling the function *SetDREvent*. Figure 2.11 shows the function *SetDREvent*. The shared variable *_DR_Event_Flag* is protected by a mutex lock *_EVENT_LOCK*. ISP can set *_DR_Event_Flag* to its desired value under the situation that the current value of *_DR_Event_Flag* is *NO_Event*. *NO_Event* implies the previous generated event had been consumed by the daemon.

Our daemon will *probe* the *DR Event* in the record mode and *wait* for it in the replay mode. As the probing function shown in Figure 2.12, the daemon copies the value of *_DR_Event_Flag*, sets it to "no event", and returns the previous *_DR_Event_Flag*. It is possible that the returned value is "no event." It means that ISP did not produce a *DR event*.

On the other hand, in the replay mode, when it is time to replay a *DR event* according to the log file, the daemon must wait and consume a *DR event* before continuing the replaying procedure. In the waiting function shown in Figure 2.13, the daemon will not leave the while loop until the value of *_DR_Event_Flag* is not *NO_Event*. However, the value of *_DR_Event_Flag* is not important. That the *DR event* denotes the exploration decisions made by ISP. In most of the cases, the *DR event* at a specific point in the trace

```
 1: SetDREvent(event) {
 2:     bool success = false
 3:     while !success {
 4:         pthread_mutex_lock(&_EVENT_LOCK)
 5:         if _DR_Event_Flag == NO_EVENT {
 6:             _DR_Event_Flag = event
 7:             success = true
 8:         }
 9:         pthread_mutex_unlock(&_EVENT_LOCK)
10:     }
11: }
```

Figure 2.11. Set a DR Event

```
1:  ProbeDREvent {
2:      pthread_mutex_lock
3:      int current_flag = _DR_Event_Flag
4:      _DR_Event_Flag = NO_EVENT
5:      pthread_mutex_unlock
6:      return  current_flag
7:  }
```

Figure 2.12. Probe the DR Event

```
1:  WaitDREvent {
2:      int get_event = NO_EVENT
3:      while get_event == NO_EVENT {
4:          pthread_mutex_lock(&_EVENT_LOCK)
5:          if _DR_Event_Flag != NO_EVENT {
6:              get_event = _DR_Event_Flag
7:              _DR_Event_Flag = NO_EVENT
8:          }
9:          pthread_mutex_unlock(&_EVENT_LOCK)
10:     }
11:     return  get_event
12: }
```

Figure 2.13. Wait for a DR Event

of the current round should be the same as in the previous round, which means ISP replays the previous choice at this point. Once the value of the *DR event* is different from the previous value at a certain point in the schedule, it means that ISP chooses a different match on a nondeterministic point. At this point, the daemon will switch from the replay mode to the record mode. Then we discard the previous round log file. Therefore, we just enforce the in order occurring of *DR events* but do not expect the deterministic values.

2.2.5 Daemon-Threads Communication Protocol

Our race replay daemon is implemented as a thread (Pthread) created by ISP. The daemon keeps doing similar tasks while helping ISP deterministically replay the thread schedules. While in record mode, the daemon keeps probing the *DR events* and the messages from the threads. The order in which these events and messages are successfully probed is the order they are recorded in the log file. On the other hand, while in replay mode, the daemon will "grant" the events or the messages in order. Remember that threads send messages in our version of Pthread functions. They will not continue in those functions until the daemon sends ACK or NACK to them. For our race replay daemon, sending an ACK to a thread is just like "authorizing" the thread to continue

the corresponding action. The same things happened on ISP. ISP will be blocked by *SetDREvent* if the current value of *_DR_Event_Flag* is not "no event." This means that the previous *DR event* had not been granted by the daemon. Therefore, ISP needs to wait for the daemon to consume the previous event before raising the new *DR event*.

Figure 2.14 shows the pseudo code of our race replay daemon's main function. For every round of ISP verification (executing the tested program), the daemon will execute its main function once. The daemon will only start from the record mode in the first round. In the consequent rounds, it will start from the replay mode. This is shown in line 2.

Before explaining the main routine of the daemon, we need to briefly introduce the function, *ProbeMessage*. The function *ProbeMessage* in line 16 and 21 is used to receive the incoming socket messages from all the threads. This function also has different behaviors in the replay and the record modes. For the usage in the record mode, it will issue the nonblocking receive calls for all the threads. For the different

```
 1: Daemon_Main_Body {
 2:     mode = is_first_round ? RECORD : REPLAY
 3:     while true {
 4:         if mode == REPLAY {                                  ▷ enter the replay mode
 5:             replay_event = PopLogFile()                  ▷ pop a record tuple from the log file
 6:             if replay_event == DR_Event {
 7:                 dr_event = WaitDREvent()                 ▷ waiting for an DR event
 8:                 WriteLog(dr_event)
 9:                 if dr_event == EXPLORE {
10:                     ACKAllPendingEvents()
11:                     mode = RECORD
12:                 }
13:             } else                                   ▷ a pthread action from a certain thread
14:                 if !AckInPendingList(replay_event) {          ▷ see if the event is pended
15:                     repeat
16:                         ProbeMessage(REPLAY, replay_event)
17:                     until ProbeMessage success
18:                 }
19:             }
20:         else if Mode == RECORD {                             ▷ enter the record mode
21:             ProbeMessage(RECORD, NULL)
22:             dr_event = ProbeDREvent()
23:             WriteLog(dr_event)
24:             if dr_event == TERMINATE or RESTART {
25:                 EndOrRestart()                ▷ routines for the daemon restart or terminate
26:                 break
27:             }
28:         }
29:     }
30: }
```

Figure 2.14. The Main Loop Body of Our Race Replay Daemon

types of incoming message, *ProbeMessage* will handle them with different actions. We will describe these actions later in detail. *ProbeMessage* requires a desired event (*event* in line 16) in replay mode. It still issues the nonblocking receive calls for all the threads. However, it will keep receiving the messages until the desired event (a specific message coming from a specific thread) happens. When those incoming messages do not match the desired event, the daemon will put them into a random access queue. Those pending messages denote that there are some threads blocked by our version of pthread calls and wait for the right timing to be unblocked.

Now we can look into the code in the daemon's main routine. In the record mode (line 20 to 28 in Figure 2.14), the daemon checks if there are any incoming messages from the threads by calling *ProbeMessage*. After that, it checks if there are any *DR events* raised by calling *ProbeDREvent*. In line 23, the daemon writes the returned value of *ProbeDREvent* into the log file. We do not record dr_event when its value is "no event." However, the function *WriteLog* in line 23 implicitly excludes this case. If the value of dr_event is "terminate" or "restart," the daemon performs some clean up tasks and leaves the main routine.

For the replay mode code in the daemon's main routine (line 4 to 19 in Figure 2.14), the daemon pops a tuple from the log file in each iteration. If the current event ($replay\_evet$ returned by *PopLogFile* in line 5) is a *DR event*, the daemon waits for ISP to set an event. Note that we still record that the event occurred even if in replay mode. This is because for the next round of the verification, ISP may still replay a section of the same schedule with the previous round. Therefore, we still need to record every message and event during the replay mode for the next round. The section from line 9 to line 12 handles the case that the daemon acquires an event whose value is "explore." This case means that ISP chose to explore a different schedule from this point. Therefore, from this point to the end of this round execution, the daemon will stay in record mode (line 11). Before entering record mode, the daemon has to ACK all the pending messages. The function *ACKAllPendingEvents* in line 10 sends ACK to all the pending messages in the order of their coming. Meanwhile, it will record all of them into the log file.

For the case that the current event $replay\_event$ in line 5 is not a *DR event*, it means that there should be a certain thread calling a certain pthread function. We do not show the detail data structure of the type of $replay\_event$. But it is a tuple that stores the information to indicate that it is a *DR event* for now along with the thread id of the calling function name. For a pthread function call event, the daemon will check if it had

been stored in the pending message list by calling the function *AckInPendingList*. If the daemon did not find the destination event in the pending list, it will call *ProbeMessage* to collect messages from the threads and ACK the desired one.

Figure 2.15 is the pseudo code for the function *ProbeMessage*. The for statement in line 3 issues a nonblocking receive call (in line 4) for a thread in each iteration. The function *valid* in line 5 tests if the nonblocking receive call acquired a valid message or not. The block from line 6 to line 19 decides whether the daemon should grant (send ACK) this incoming message or not. When the daemon is in record mode, it always grants the incoming message. For the *TRY_LOCK* and *FINALIZE* message, the daemon grants them anyway. This is because the *TRY_LOCK* message is just for the daemon to realize "who is waiting for which mutex lock." This is for the purpose of deadlock detection. For *FINALIZE* message, it is a note for the daemon that "a

```
 1: ProbeMessage(mode, replay_event) {
 2:     success = false
 3:     for every thread t {                          ▷ try to receive message from every thread
 4:         recv_event = non_blocking_recv(t)
 5:         if valid(recv_event) {                    ▷ if we readlly acquired a incoming message
 6:             if mode == RECORD ∨ recv_event == TRY_LOCK ∨ FINALIZE {
 7:                 grant = true
 8:             } else                                                    ▷ in record mode
 9:                 if relay_event.tid == t ∧ (replay_event.event == recv_event ∨ (replay_event.event
                        == ENTER_CS ∧ recv_event == TRY_AGAIN)) {
10:                     grant = true
11:                     if replay_event.event == recv_event {
12:                         success = true
13:                     } else
14:                         success = false
15:                     }
16:                 } else
17:                     grant = false
18:                 }
19:             }
20:             if grant {
21:                 SendACK(mode, t, recv_event)
22:             } else
23:                 if recv_event == ENTER_CS {
24:                     SendNACK(t)
25:                 } else
26:                     PushPendingList(recv_event)
27:                 }
28:             }
29:         }
30:     }
31:     return success
32: }
```

Figure 2.15. The Pseudo Code of ProbeMessage

thread is going down." It is always safe to grant this message. In the replay mode, the daemon can only grant the incoming message which matches the current desired event. Therefore, the daemon will not grant the incoming message from the thread which does not match the thread id of the desired event. Furthermore, the type of the event also needs to be the same. An exception is that the incoming message is *TRY_AGAIN* and the desired message is *ENTER_CS*. In this special case, the daemon should grant this message. However, we should not claim the replay is successful in this case because the daemon is waiting for the following *ENTER_CS* after this *TRY_AGAIN* message. Consider Figure 2.5, the desired *ENTER_CS* from the thread will be issued in line 9. The section from line 10 to line 14 in Figure 2.15 notifies the daemon's main routine that the replay is incomplete. For those undesired messages, the daemon will put them into a pending list except the *ENTER_CS* message. For the undesired *ENTER_CS* message, the daemon will send NACK instead to make the thread give up the newly acquiring mutex lock.

We use the function *SendACK* in Figure 2.15 to send back the ACK to the threads. There are some complex designs in this function. Figure 2.16 shows the pseudo code of the function *SendACK*. To send ACK for the *CREATE* event (from line 2 to 5), we need to write the event into the log, send ACK, and then wait for the connection from the newly created thread (done by the function *WaitForBUILD* in line 5). To send ACK for the *ENTER_CS* and the *RELEASE* event (from line 6 to 12), we need to write the event into the log, send ACK and update the deadlock detection table. For *ENTER_CS*, we also need to check if the deadlock occurred (by calling *DetectDeadlock* in line 11). We will introduce the deadlock detection built into our daemon later. Handling the *TRY_LOCK* case (from line 40 to 43) is similar to the *ENTER_CS* case. The only difference between them is that we do not record the *TRY_LOCK* event. We do not care about how many times a thread is trying to acquire a mutex lock (once or twice). We only record that "the daemon granted an access of mutex lock and sent back an ACK." Before introducing the *WAIT* case, we look back to our version of *pthread_cond_wait* shown in Figure 2.9. The thread calling the original *pthread_cond_wait* function gives up a mutex lock and waits for a signal. The two actions, "give up" and "wait," should be *atomic*. In other words, the instructions from line 2 to line 4 in Figure 2.9 should work like an *atomic instruction*. Our solution is to make the daemon "see these instructions as atomic."

```
1:  SendACK(mode, t, recv_event) {
2:      if recv_event == CREATE {
3:          WriteLog(t, recv_event)
4:          send(t, ACK)
5:          WaitForBUILD()
6:      else if recv_event == ENTER_CS ∨ recv_event == RELEASE {
7:          WriteLog(t, recv_event)
8:          send(t, ACK)
9:          UpdateDeadlockDetector()
10:         if recv_event == ENTER_CS {
11:             DetectDeadlock()
12:         }
13:     else if recv_event == WAIT {
14:         WriteLog(t, recv_event)
15:         send(t, ACK)
16:         waken_event = blocking_recv(t)          ▷ receive the consequent WAKEN message
17:         UpdateDeadlockDetector()
18:         DetectDeadlock()
19:         PushCondWait(t, waken_event)
20:         if mode == REPLAY {
21:             PushPendingList(waken_event)
22:         }
23:     else if recv_event == WAKEN {
24:         WriteLog(t, recv_event)
25:         send(t, ACK)
26:         RemoveCondWait(t, recv_event)
27:         UpdateDeadlockDetector()
28:     else if recv_event == SIGNAL {
29:         WriteLog(t, recv_event)
30:         send(t, ACK)
31:         if mode == RECORD {
32:             waken_event = PopCondWait()
33:             if valid(waken_event) {
34:                 WriteLog(t, waken_event)
35:                 send(waken_event.tid, ACK)
36:             }
37:         }
38:     else if recv_event == TRY_LOCK {
39:         send(t, ACK)
40:         UpdateDeadlockDetector()
41:         DetectDeadlock()
42:     else if recv_event == FINALIZE ∨ recv_event == TRY_AGAIN {
43:         send(t, ACK)
44:     }
45: }
```

Figure 2.16. The Pseudo Code of SendACK

Consider the code in Figure 2.16 from line 13 to 22, once the daemon is going to send ACK for the *WAIT* message to a thread, it must do nothing but wait for the consequent *WAKEN* message from that thread. After receiving the *WAKEN* message, the daemon pushes it into a random accessing queue by calling the function *PushCondWait* in line 19. If the daemon is in the replay mode, it also pushes the *WAKEN* message into the pending list. Now we see how the daemon sends ACK for the *SIGNAL* message. After writing the log file and sending ACK, the daemon finds the oldest *WAIT* message which matches the signaled conditional variable and sends ACK to the corresponding thread (from line 31 to 37). For the case of the daemon in replay mode, the daemon does nothing after acking the *SIGNAL* message. The log file should contain a record from which thread will be wakened by this signal. The daemon just enters the next iteration in its main routine and replays the next event.

2.2.6 Parallelizing the Race Replay Daemon

Now we have presented our design of our race replay daemon with the corresponding instrumented Pthread functions. One drawback of this design is the performance hazard. This is because we serialize all the events from the threads with the *DR events* from ISP. The daemon works like a "central scheduler" of all the threads. In fact, we need not serialize all the events. We must keep the order between the Pthread events and the *DR events*. However, we need not keep the order of the Pthread events from the different processes. In all the pseudo codes shown earlier, we hide the concept of *processes* in them. We only show how to deal with the message between the daemon and threads. But the hybrid programs usually consist of multiple processes and each process consists of multiple threads. Threads of different processes do not virtually share memory.

To explain how we parallelize the race replay daemon, here we use a pair, (x, y), to be an id of a thread. x denotes the process id and y denotes the thread id in the process x. We use $pe_{(x_i,y_j)k}$ to denote a Pthread event of the thread (x_i, y_j) in a log file. We use dr_k to denote a *DR event* in a log file. k denotes their index. We use $a \rightarrow b$ to denote that the event a is recorded before b in a log file and $a \Rightarrow b$ to denote that the event a is replayed before b based on this log file. The *DR events* is "universal" for all threads. Given a log file for a certain round of verification, the replay must hold the following rules:

- $\forall pe_{(x_i,y_j)m}, pe_{(x_i,y_j)n} \in L \wedge pe_{(x_i,y_j)m} \rightarrow pe_{(x_i,y_j)n} : pe_{(x_i,y_j)m} \Rightarrow pe_{(x_i,y_j)n}$.
- $\forall dr_m, dr_n \in L \wedge dr_m \rightarrow dr_n : dr_m \Rightarrow dr_n$.
- $\forall pe_{(x_i,y_j)m}, dr_k \in L \wedge pe_{(x_i,y_j)m} \rightarrow dr_k : pe_{(x_i,y_j)m} \Rightarrow dr_k$. $\forall dr_k \rightarrow pe_{(x_i,y_j)m} : dr_k \Rightarrow pe_{(x_i,y_j)m}$.

Rule 1 defines that for any two pthread events in a log file L from the same thread, (x_i, y_j), they must be replayed by their recorded order. Rule 2 defines that for any two *DR_events* in a log file, they must be replayed by their recorded order. Rule 3 defines that for any pair of a pthread event and a *DR_event*, they must be replayed by their recorded order. These three rules imply:

- $\forall pe_{(x_i,y_j)m}, pe_{(x'_i,y'_j)n}, dr_k \in L \wedge x_i! = x'_i \wedge pe_{(x_i,y_j)m} \rightarrow dr_k \rightarrow pe_{(x'_i,y'_j)n} : pe_{(x_i,y_j)m} \Rightarrow pe_{(x'_i,y'_j)n}$.

It means that for any two thread events from different processes, they must be replayed by their recorded order if there is any *DR_event* between them. Therefore, for any two thread events from different processes, they can be replayed in an arbitrary order if there is no *DR_event* between them.

It turns out with that we record the Pthread events from different processes separately into different log files. For the *DR events*, they need to be written into all the log files. While replaying the tested program, every process will be restricted to its thread schedule according to its log file. In addition, replays of these processes need to be "synchronized" on *DR events*. In this way, we parallelize our race replay daemon.

Figure 2.17 illustrates how we parallelize the daemon. Suppose (p_i, t_j) denotes a thread call from the j^{th} thread of the process i. Suppose $ISPDecision$ denotes a *DR event*. The upper rectangle denotes the original log file. The daemon serializes all events. The three bottom rectangles denote the three distributed log files for three processes. These log file can be recorded/replayed in parallel but need to be synchronized on *DR events*.

2.2.7 An Example

We show an example in this section to illustrate how the daemon determinizes the behaviors of an MPI-Pthread hybrid program. This example program contains a three-node system. Nodes exchange data via MPI. Figure 2.18 shows the code for node 0.

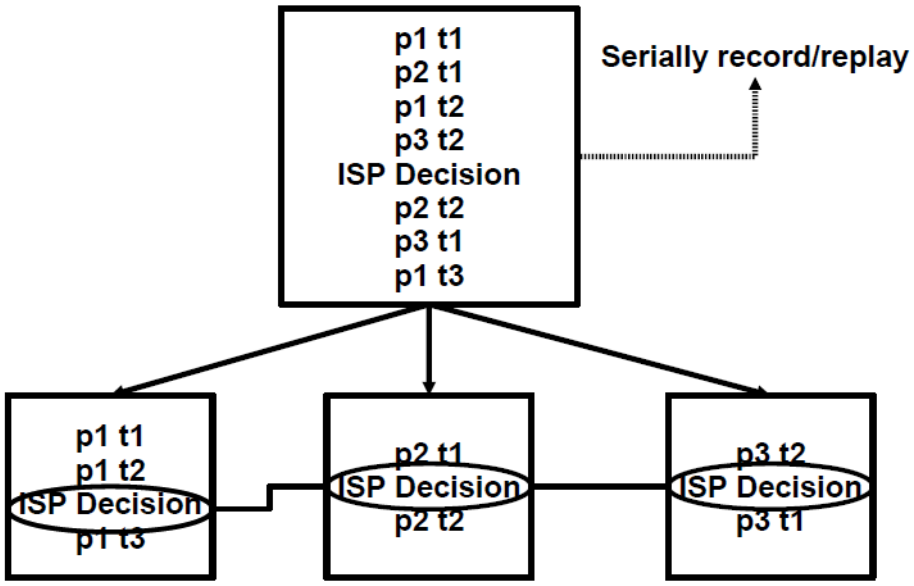

Figure 2.17. ISP-Daemon System

It keeps receiving messages from the other two nodes until the terminal messages are received from each of them (line 3 and 4). Figure 2.19 shows the code for node 1 and node 2. The code is mixed by MPI and Pthread library. Each of these two nodes contains two threads: a producer and a consumer. These two threads share a queue (not shown in our example code) and operate it by three functions: *Produce*, *ConsumeAll*, and *Terminate*. Function *Produce* is used by the producer putting an object to the queue. Function *Terminate* is used by the producer putting a terminal message to the queue. Function *ConsumeAll* is used by the consumer extracting *all* objects and messages from the queue. The producer (line 1 to 8 in Figure 2.19) tries to acquire the lock twice and puts one object each time entering critical section. After putting the second object, the producer puts a terminal signal into the shared queue. The consumer (line 9 to 17 in Figure 2.19) keeps extracting objects from the shared queue. Once acquiring something from the queue, it sends all contents from the queue to node 0 by *MPI_Send*. After that, the consumer will check if the terminal message has been acquired. If it has, the consumer stops testing the queue.

We will observe nondeterministic MPI behaviors while using ISP to test this program. In the first round of executing this program, suppose the producer of node 1 puts two objects with the terminal signal into the shared queue before the consumer acquiring anything. In other words, the order of entering the critical section is producer, producer, then consumer. To simplify our discussion, we assume the producer and the

```
1.  while (1) {
2.    MPI_Recv(buff, *);
3.    if (IsDone(buff)) {
4.      break; }
5.    else {
6.      DoSomething(buff); }}
```

Figure 2.18. The Code of Node 0

consumer of node 2 follow this same order. The log of node 1 (node 2) is shown in Figure 2.21. Changing "node 1" to "node 2" in Figure 2.21 will be the log of node 2. Figure 2.20 shows the log of node 0. Since node 1 and node 2 issue only one *MPI_Send* each, node 0 issues two *MPI_Recv* calls.

In the first replay of this program, we assume that the producer and the consumer of node 1 change their order of entering the critical section. For example, the consumer of node 1 enters the critical section once between the two accesses of the producer. That is, the order of entering the critical section is producer, consumer, producer, then consumer. Figure 2.22 shows the log of replaying node 1. To simplify our discussion, we assume that the producer and the consumer of node 2 luckily follow the same order with the previous round. In this case, node 0 needs to issue three *MPI_Recv* calls to match all *MPI_Send* calls (two from node 1 and one from node 2). However, ISP expects node 0 to issue only two *MPI_Recv* calls as what it did in the previous round.

```
1.  producer () {
2.    pthread_mutex_lock(&lock);
3.      Produce();
4.    pthread_mutex_unlock(&lock);
5.    pthread_mutex_lock(&lock);
6.      Produce();
7.      Terminate();
8.    pthread_mutex_unlokc(&lock); }

9.  consumer () {
10.   while (1) {
11.     pthread_mutex_lock(&lock);
12.       products = ConsumeAll();
13.       if (!Empty(products) {
14.         MPI_Send(products, 0); }
15.     pthread_mutex_unlock(&lock);
16.     if (IsTerminated(products)) {
17.       break; }}}
```

Figure 2.19. The Code of Node 1 and 2

```
1.  DR - (node 0, MPI_Recv)
2.  DR - (node 0, MPI_Recv)
```

Figure 2.20. The Original Log of Node 0

```
1.  ENTER_CS - (node1, producer)
2.  RELEASE - (node1, producer)
3.  ENTER_CS - (node1, producer)
4.  RELEASE - (node1, producer)
5.  ENTER_CS - (node1, consumer)
6.  DR - (node1, MPI_Send)
7.  RELEASE - (node1, consumer)
```

Figure 2.21. The Original Log of Node 1 (Node 2)

```
1.  ENTER_CS - (node1, producer)
2.  RELEASE - (node1, producer)
3.  ENTER_CS - (node1, consumer)
4.  DR - (node1, MPI_Send)
5.  RELEASE - (node1, consumer)
6.  ENTER_CS - (node1, producer)
7.  RELEASE - (node1, producer)
8.  ENTER_CS - (node1, consumer)
9.  DR - (node1, MPI_Send)
10. RELEASE - (node1, consumer)
```

Figure 2.22. The Log of Replaying Node 1

Our race replay daemon helps ISP to observe the deterministic MPI behaviors. Obviously, the reason the nondeterministic MPI behavior happened is race condition between threads. If using ISP with our daemon, in the first replay, when the consumer thread of node 1 acquires the lock (in line 3 in Figure 2.22), the daemon will send a *NACK* to the consumer thread. Then the consumer thread will give up the lock and wait. Our daemon enforces the consumer to yield the lock to the producer according to the log of the previous round (Figure 2.21). After the second time the producer leaves the critical section (line 4 in Figure 2.21), the daemon is aware that it is the consumer's turn to access the critical section. Meanwhile, the daemon knows that the consumer is blocked. Therefore, the daemon sends an *ACK* to the consumer. Under the control of our daemon, when the consumer calls *ConsumeAll* in the replay round, it will acquire two objects with a terminal signal. Then the consumer will call *MPI_Send* only one time to send all objects to node 0. Consequently, the MPI behavior of this program is determinized by our daemon.

2.2.8 Deadlock Detection

Since our daemon keeps track of the Pthread calls from all the threads, it can also detect deadlock by constructing a "wait for" graph [26]. We can map the whole threads system into a graph $G = (V, E)$ which V is a set collecting all the threads in the tested program. The *TRY_LOCK* event reveals which mutex lock the thread is waiting for. Supposing the thread i is waiting for a mutex lock m which is owned by the thread j, there is a directed edge $e \in E$ from v_i to v_j. A thread that received an ACK for the *WAKEN* event is also waiting for a certain mutex lock. Of course, we also need to keep track of the *ENTER_CS* and *RELEASE* events to know which thread is occupying which mutex lock. Finally, we can maintain a wait for graph on the fly and detect the deadlock by checking the cycle in the graph.

2.2.9 Informal Correctness Sketch of Our Scheduler

Our hybrid scheduler was tested and validated by designed test cases and experiments. It merits more rigorous testing/analysis. In future work, we will test around its Pthread record/replay mechanisms ensuring their correct operation for all record/replay sessions of up to a certain length. Then we will analyze how it interacts with a whole hybrid program. This project is important to have a reliable hybrid checking scheduler, and is estimated to take a year for a typical student, if carried out.

2.3 Related Work

2.3.1 Output Deterministic Replay

The idea of our "record-replay" mechanism is inspired by ODR, output deterministic replay [3]. ODR gathers a schedule trace, input trace, and read trace from the original execution and translates the program into logical formula. It then finds an execution with the same "output" as the original run. The "output" could be console outputs, program assertions, or even the value of a set of variables at a certain point in the program. In ODR, an inference method called SI-DRI records the lock-order as the schedule trace. This inspired us to design the exterior helper daemon for ISP to deterministic replay the threaded MPI programs.

2.4 Experimental Results

We have tested our "record/replay" mechanism with Eddy Murphi serving as the tested program for ISP to examine. Eddy Murphi itself needs an input protocol to

examine (it is a model checker after all) and we chose the *n_peterson* model (Peterson's mutual exclusion) as its test input. In effect, we are model checking a model checker – thus we have to scale down problem sizes at each level of model checking to limit state explosion.

In all our tests, ISP with our race deterministic replay daemon explored interleavings successfully. On the other hand, the original version of ISP can only explore interleavings in few small cases and unpredictably crashes when the Pthread schedules change. This demonstrates that we can make a reasonable set of assumptions and successfully determinize one API's behavior while probing the space of nondeterminism of another API.

In our experiments, we set different depth bounds on BFS to control the scale of exploration. We also added nondeterministic sleep durations before *pthread_mutex_lock* calls to jiggle the timing of Pthread calls. We showed that this can crash the unmodified (previous) ISP more readily while our new scheme can handle this extra nondeterminization quite robustly and enforce schedules reliably using "record/replay."

Table 2.1 presents our results. The *original isp* means the old version of ISP which cannot handle the nondeterministic replay problem. The *isp-daemon* means the new version of ISP which can enforce the Pthread call permutations. The column *ISP version & configuration* denotes the version of ISP we used, the number of processes, and the depth bound. For instance, "original isp / p3 / d5" means the result of running Eddy Murphi on the old version ISP with three processes created and with a 5-level depth bound BFS. The column *interleaving explored* denotes the number of interleavings

Table 2.1. Experiment on n_peterson Model

ISP version & configuration	interleaving explored	min./max./ave. DR events
original isp / p3 / d3	11	112 / 112 / 112
original isp / p3 / d5	fail on 2ed	133 / 133 / 133
original isp / p4 / d3	fail on 4th	145 / 149 / 146
isp-daemon / p3 / d3	11	112 / 112 / 112
isp-daemon / p3 / d5	61	133 / 133 / 133
isp-daemon / p3 / d10	over 1500	179 / 179 / 179
isp-daemon / p3 / d20	over 1600	723 / 765 / 727
isp-daemon / p4 / d3	6	141 / 145 / 143
isp-daemon / p4 / d5	over 1097	174 / 174 / 174
isp-daemon / p4 / d10	over 2000	300 / 304 / 303
isp-daemon / p4 / d20	over 2400	898 / 898 / 898

explored by ISP while verifying Eddy Murphi. The column *min./max./ave. DR events* denotes the minimum, maximum, and average number of DR events we encountered in one execution. The old version ISP can explore interleavings without encountering an error when the depth bound is small. However, increasing the depth bound caused the old version ISP to crash in the face of nondeterminism.

Based on these experiments, we are in a position to recommend our "record/replay" approach to the nondeterministic replay problem. We also can state the following claims:

- To the extent ISP explored the schedules of Eddy Murphi, it confirmed that there are no deadlocks or resource leaks in the MPI code-base of this tool. Before we devised our current record/replay scheme, we could not be as sure about this situation.

- We have not examined the schedule space of Pthread calls within Eddy Murphi sufficiently. However, the threading code of Eddy Murphi is quite simple. If these structures are implemented correctly, then we can bank on the fact that the order of enqueuing and dequeuing into the queue does not affect the correctness of Eddy Murphi.

CHAPTER 3

SEARCH SPACE REDUCTION

In this chapter, we focus on search space reduction. Instead of discussing in hybrid program scenarios, we focus our discussions in pure MPI programs. We believe that our approach can be easily migrated from pure MPI programs to hybrid programs.

3.1 Introduction

A common problem for all dynamic verification tools is dealing with a huge seach space. For example, given a 5-instruction MPI program executed by 5 processes, we get $25!/(5!)^5$ different interleavings among those instructions. This number of interleavings is over 10 billion. This is why conventional testing tools suffer from extremely poor coverage in that they get lost in this vast exponential space. Dynamic partial order reduction [6] methods reduce this space dramatically without losing any relevant behaviors. A DPOR algorithm for MPI was developed for the first time in [29] and subsequently improved in many ways [28][30]. Our dynamic verification tools, ISP, have been employed dynamic partial order reduction (DPOR) to reduce the search space. Even though DPOR for MPI is effective to eliminate the redundant interleavings of instructions, the performance of ISP is still not acceptable even verifying some simple hundred-line programs.

A good idea for the search space reduction is finding the symmetric executions in the tested programs. Consider the message passing code shown in Figure 3.1. The process 0 (the process with the rank equal to 0) issues 10 wildcard receive calls. Each process from the process 1 to 10 issues a specific send instruction to the process 0. Without any searching optimization, a dynamic verification tool will explore $10!$ different interleavings for this code. However, with the symmetry reduction, a dynamic verification tool will explore only one interleaving for this code. This is because the process 1 to process 10 are all *identical* in this code. They are semantically identical and also have the same behavior in run-time. Therefore, the order of scheduling these processes does not change the verification result.

```
1. if (rank == 0) {
2.     for (i = 1 to 10)
3.         Recv(*);
4. } else if (1 <= rank <= 10) {
5.     Send(0);
6. }
```

Figure 3.1. *Illustration of a Perfect Symmetric Scenario*

However, the *symmetric* code shown in Figure 3.1 is *perfectly symmetric*. Identifying the *perfect symmetric* property is hard. It is very difficult to check whether two processes have identical behavior run-time. For example, a process may decode the received buffer and decide the next sending target or the next receiving source. In this case, even though two processes execute the semantically equivalent code, their run time behaviors can still be different. The perfect symmetric execution rarely exists in practical concurrent programs. It is very possible to find a group of processes whose behaviors are almost identical. Their run-time behaviors are almost the same but have some slight differences. In this case, applying symmetry reduction make it possible to reduce the search space. However, it is also a challenge to create a robotic routine to check the "almost symmetric executions."

Several heuristic methods have been developed in some dynamic concurrent program verifier such as the preemption bound of CHESS [22] and the bounded mixing in ISP. The advantages of applying the heuristic methods are:

- The heuristic methods can dramatically reduce the search space.
- Most of the heuristic methods require nonsophisticated analysis or even no analysis required prior to the verification process.
- Most of the heuristic algorithms are easy to implement in the tools.

On the other hand, using the heuristic methods also bring us some drawbacks:

- The heuristic algorithms often sacrifice the reliability while eliminating the search space.
- Many heuristic methods require users to give some *heuristic information* as the input.

We believe that it is worthy for us to engage in developing the *better* heuristic al-

gorithms. Based on the advantages and the disadvantages from the above, we conclude that a *better* heuristic method means that

1. The method sacrifices less reliability but reduces more redundant search space.

2. The *heuristic information* should be easily generated (either by a user or by an automatic process).

The *heuristic information* often decides how the heuristic method performs. For example, suppose that we would like to set a depth bound while traversing the exploration tree. How much should we set for the bound? We could decide the bound by an analysis process, by learning from the previous experience, or even just picking randomly. If we give a small bound for our exploration, we can eliminate huge amounts of search space. However, we may also lose the buggy traces in the exploration tree. On the other hand, by giving a large bound, we may preserve the buggy traces but preserve many bug-free traces also. Therefore, picking an *optimal* input for the heuristic methods is important. Many heuristic methods require the users to choose a good heuristic input for the algorithm. But it is not always a trivial task to choose optimal input information. Sometimes the users may need to understand the underlying verification process in order to give an optimal input. For example, to set the traversing bound, the users need to understand the exploration tree of a concurrent program. They need to understand how deep the tree of the tested program would be and decide the search bound. They even need to know how the tool traverses the tree. Therefore, we are in a position to say that a good heuristic method must easily specify the optimal input.

In this section, we introduce a new focus plus random sampling heuristic algorithm (FPRS) and implement it in ISP. The key idea of our algorithm is that we focus on exploring some specific regions and do random-walk on the rest. Before the presentation of our algorithm, we show three motivating scenarios in the message passing programs.

3.1.1 Three Motivating Scenarios of Search Space Reduction

In this section, we show three scenarios of dynamic verifying concurrent programs. If we use ISP to verify programs containing these scenarios, we may encounter the schedule explosion problem. We will discuss how the state space explosion problem happens and how to ideally solve these problems.

```
1.  for (i = 1 to K) {
2.      A_Routine_Calls_MPI();
3.      Another_Routine_Calls_MPI();
4.      // Barrier();
5.  }
```

Figure 3.2. *Scenario 1 - Loop*

3.1.1.1 Scenario 1 - Loop

Considering the case shown in Figure 3.2, the program issues many MPI send or receive calls in a loop. (There are many MPI calls in the function *A_Routine_Calls_MPI* and the function *Another_Routine_Calls_MPI*.) The number of potential schedules will grow exponentially by the loop bound K. A heuristic approach to verify this program is sampling some iterations among this loop. It is based on the fact that this program repeatedly does similar tasks. Therefore, verifying some iterations to capture the representative behaviors is sufficient. There may or may not be a *Barrier* instruction (in line 4) issued at the end of the loop body. If there is a *Barrier* in this loop, one iteration could be representative. Therefore, ideally, we can set a *depth bound* or a *sampling bound* for the loop statements while verifying MPI programs.

3.1.1.2 Scenario 2 - Calling Subroutines

A good software engineering strategy is to define layers and develop subroutines in each layer. High level routines call low level subroutines as services. Therefore, low level subroutines should be carefully constructed and verified to guarantee reliability. The program shown in Figure 3.3 calls three functions. The function *Bug Free Subroutine* denotes a verified subroutine. The function *Unverified Subroutine* is newly developed and has not been verified. In this case, ISP will explore the interleavings in *Bug Free Subroutine* and mix with the schedules in *Unverified Subroutine*. However, in most of the cases, it is needless to test *Bug Free Subroutine* again. Ideally, we would like to skip the verified subroutines and focus on the regions we have not tested yet. We need *compositional testing*.

```
1.  Bug_Free_Subroutine();
2.  Unverified_Subroutine();
3.  Bug_Free_Subroutine();
```

Figure 3.3. *Scenario 2 - Calling Subroutines*

```
1. if (rank == 0) {
2.     for (i = 1 to 10) {
3.         Recv(*, &status);
4.         swtich (status.MPI_TAG) {
5.             case 1: .... ;
6.             case 2: .... ;
7.             default: .... ; }}
8. } else if (1 <= rank <= 10) {
9.     Send(0);
10.    Recv(0, &status);
11.    switch (status.MPI_TAG) {
12.         case 1: .... ;
13.         case 2: .... ;
14.         default: .... ; }}
```

Figure 3.4. *Scenario 3 - Nonperfect Symmetric Execution*

3.1.1.3 Scenario 3 - Nonperfect Symmetric Execution

The third scenario we observed is *symmetric* property which we introduced in section 3.1. The programs with the *symmetric* property usually use the *master-slaves* structure. The master and the slaves in such programs may decode the incoming message and perform the corresponding actions. In this case, it is very hard for us to decide whether the *symmetric* property still holds in run time. However, we still can apply the symmetry reduction if *strong symmetric* property exists. For example, if decoding the incoming message is just for probing the task completed or not, applying symmetry reduction can still reduce the search space, but make it highly possible to preserve the bugs.

3.1.2 Focus Plus Random Sampling Heuristic Algorithm

We studied three scenarios in section 3.1.1 and observed the ideal solutions. To solve these cases, we need to set *depth bound*, analyze the *symmetric* property, and do *compositional testing*. We can simplify them by answering two questions:

- Which process should be involved in the verification process?
- When (at which position in the program) will these interesting processes be involved in the verification process?

Finding the answers to these two questions automatically may not be a trivial task. It requires extra sophisticated static or dynamic analysis programs. However, people who use ISP to test their programs should understand the tested programs well. We

believe that the *who and when questions* (the heuristic input of our algorithm) can be easily answered by the program developers.

3.2 Algorithm

In this section, we present our FPRS algorithm. We need two inputs for this algorithm. The first one is a deterministic concurrent program. The second parameter is a set of program counter pairs. Suppose we have a program P creates n processes as our first parameter, the second parameter PC is a set of pairs which enclose a set of regions in the program P. We denote the PC as $PC = \{(pc_{sy}, pc_{ty}) \mid 1 <= y <= n\}$. Each pair in PC marks a specific region in the program P. For a pair (pc_{sy}, pc_{ty}), the pc_{sy} denotes the starting point of a specific region and the pc_{ty} denotes the terminating point. Every two enclosed regions should not overlap each other. Specifically, two processes that execute the same program can have the same enclosed regions. However, the enclosed regions in the same process should not overlap each other.

While executing the input program P, we define two states of all processes: *explore* and *nonexplore*. A process p_i of the program P is in the *explore* state at a certain point if: Given the history of the program counter $H = \{h_1, h_2, ..., h_k\}$ is before the current point,

- $\exists h_x \in H \wedge (pc_{sx}, pc_{tx}) \in PC \ : \ h_x == pc_{sx}$
- $\nexists h'_x \in H \wedge (pc_{sx}, pc_{tx}) \in PC \wedge x' > x \ s.t. \ h'_x == pc_{ts}$

A process p_i is in the *nonexplore* state it is not in the *explore* state. At every nondeterministic decision point in the schedule, like a *fence point* of ISP, we define two sets of processes, E and NE. $E = \{p_i \mid p_i \, in \, the \, explore \, state\}$ and $NE = P - E$. We denote the ample set at this nondeterministic point as A. The sender-receiver pair candidates in A are labeled as $c_j = (s_j, r_j)$. Therefore, $A = \{c_j \mid s_j \in P \wedge r_j \in P \wedge s_j \neq r_j\}$.

With the setting described above, we can now present the algorithm. Our algorithm redefines the ample set of each nondeterministic point. Given an ample set A, our algorithm return a new ample set NA. We define two sets, EM and NEM, which was initially empty. For each sender-receiver pair $c_i = (s_i, e_i) \in A$, we append c_i to EM if $s_i \in E \vee e_i \in E$. Otherwise, we append c_i to NEM. Finally, we get:

$$NA = EM \cup random(NEM)$$

The function *random()* chooses an element randomly from the input set. One problem derived from using the random number is the nondeterministic replayability. To deterministically generate the same new ample set NA on the same nondeterministic point, we can record the seed of the random number generator at the beginning of the verification. Then we can apply the same seed on each subsequent round to deterministically generate NA. In fact, the algorithm still works with other selection functions. For example, we can apply a function to choose the sender-receiver pair with smallest sender process id (or rank). However, the random selection gives us some extra benefits. The main advantage of the random selection is the existence of *nonbiased* choices on a given number of chances to pick up interleaving [31]. We can expect to detect the bug more quickly with it.

Figure 3.5 illustrates how our FPRS algorithm works. The left most rectangle denotes the original ample set of a certain fence point. (x, y) denotes a sender-receiver match which x is the sender and y is the receiver. In this example, we assume that process 1, 2, and 3 are in their specific exploration regions. Therefore, we divide the original ample set into two groups (the two rectangles in the middle). One group (EM) the matches that the sender or the receiver is in its exploration regions. One group (NEM) for the rest. Then the final ample set (the right rectangle) unions EM and one of the elements in NEM.

3.2.1 Interface for the Search Space Reduction

We have implemented this algorithm in ISP. We provide a simple interface to specify *whose* and *where* the exploration regions are. The *who* means "which process" in an MPI program. The *where* means "which region" in an MPI program. We provide two

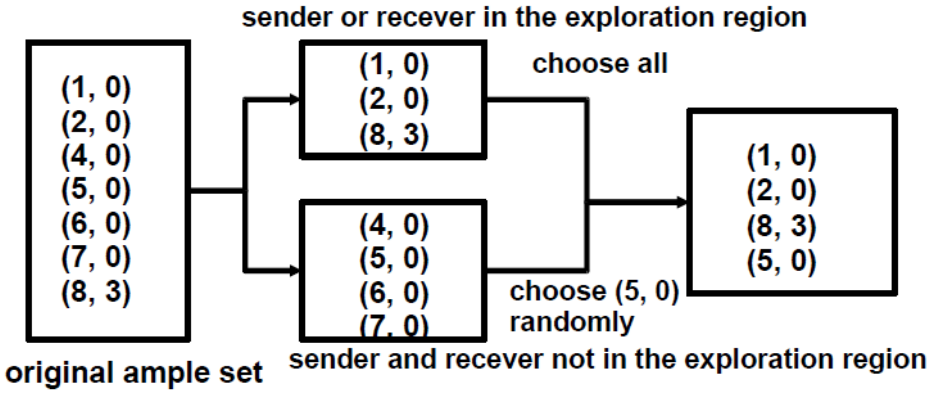

Figure 3.5. Illustration of reducing ample set

functions for the programmers to specify these two key points: *StartExp* and *EndExp*. The MPI program developers need to insert these two functions into their programs in order to mark the specific regions in their code. For those uninteresting regions, ISP will randomly choose one sender-receiver match for each ample set in them and quickly go through those sections.

Here we review the three scenarios shown in section 3.1.1 and demonstrate how to focus the verification process.

3.2.1.1 Loop Sampling

Considering Figure 3.6, the *StartExp* in line 1 and the *EndExp* in line 3 sample the loop with the bound b. ISP will only explore the potential schedules in the first b iterations. After the b iteration to the end of the loop, ISP will randomly choose a match on each fence point. The benefit of applying our FPRS heuristic method is that we set a sampling bound for a loop without changing the original loop bound.

3.2.1.2 Compositional Testing

Considering Figure 3.7, the *StartExp* in line 1 and the *EndExp* in line 3 restricts ISP to explore the schedules only in the function *Unverified_Function1*. It makes ISP able to test only a specific function or subroutine in the whole program. In this way, we can test the underlying subroutines of a MPI program first and then avoid scheduling in these subroutines while testing on higher layer functions.

3.2.1.3 Symmetry Reduction

In Figure 3.8, we restrict the process (*who*) involved in the schedules exploration. ISP will only explore send-receive matches whose sender is the process 1. In this way, we introduce the *symmetric* technique to ISP. More precisely, the original ample set of the first wildcard receive is $\{(1,0),(2,0),...,(10,0)\}$. Since we only care about the

```
1.  StartExp();
2.  for (i = 1 to K) {
3.      if (i == b) EndExp();
4.      A_Routine_Calls_MPI();
5.      Another_Routine_Calls_MPI();
6.      // Barrier();
7.  }
```

Figure 3.6. *Solution for Scenario 1: Loop Sampling*

process 1, at this point, we will try *scheduling the send from the process one at this point and not scheduling it*. Redefining the ample set by our algorithm, the new ample set will be $\{(1,0), random((2,0),...,(10,0))\}$. Therefore, the search space of the schedules is just like the program creating only two processes.

3.3 Related Work

3.3.1 Bounded Mixing

Bounded mixing is a heuristic method which has been implemented in ISP for the state space reduction. The ISP users set a bound b as a heuristic input information. Given an ample set, ISP will explore only those sender-receiver matches chosen in the consequent b ample sets. Consequently, bounded mixing is good to find out the buggy interleavings which happen in a bounded window. However, compared to FPRS, it is program structure unaware. Also, generating an optimal input for bounded mixing is nontrivial. A small window (the bound) may miss the bug. On the other hand, a big window may preserve many bug-free schedules.

3.3.2 Preemption Bound and Preemption Sealing

The preemption bound idea [22] is to restrict the number of preemptions while the multithreaded programs verification tools exploring schedules. This idea is based on two observations: (i) most of the bugs require at least one preemption to reproduce them, (ii) most of the bugs can be reproduced within a finite bound of preemptions. Under these facts, setting the preemption bound can reduce the schedule searching space. Reference [22] shows that CHESS with preemption bound scheduler can efficiently verify several benchmark programs without losing accuracy.

Preemption Sealing is an application of preemption bound idea [27]. Preemption sealing scheduler will disable the preemption in some particular scopes of program execution. Disabling preemptions will not introduce any additional behaviors or safety violations (i.e., deadlock and race condition) in programs. Therefore, the preemption

```
1.  Bug_Free_Subroutine();
2.  StartExp();
3.  Unverified_Subroutine();
4.  EndExp();
5.  Bug_Free_Subroutine();
```

Figure 3.7. *Solution for Scenario 2: Compositional Testing*

sealing scheduler works as skipping the exploration on these scopes. Then we can compositionally test the programs or even detect multiple bugs.

The main difference between preemption sealing and our algorithm is that the preemption sealing (or preemption bound) idea can only apply to multithreaded programs. Multiprocess programs are hard to control by restricting the preemptions. Our algorithm is suitable for both multithreaded and multiprocess concurrent programs.

3.4 Experimental Results

In this section, we present some experimental results for our search space reduction algorithm. We use both industrial size programs and our designed benchmarks. We seed bugs in some of our benchmarks and keep the rest unchanged. We compare the verification results between the original ISP and the ISP with our FPRS heuristic. For those bug seeded benchmarks, we want to see how fast the bug can be caught. We set the time bound for this type of benchmark as 4 hours. For those unchanged benchmarks, we want to see how fast the ISP returns the verification result. We do not prove the unchanged benchmarks are buggy or bug-free. However, it is important for a verification tool to answer this *yes or no* question within an acceptable time. We set the time bound for the unchanged benchmarks as 60 hours. We expect ISP (original or FPRS) to reply: "this program is buggy" or "this program is bug-free," instead of "I don't know the answer right now."

Before introducing the benchmarks and showing the experiment results, we define *the depth of a bug* in an MPI program to label how hard the bug we seeded is to detect.

3.4.1 Bug Depth

In [4], the depth of a bug in the multithreaded concurrent programs. The depth of a bug directly relates to the number of order constraints to meet this bug. In multithreaded

```
1. if (rank == 0) {
2.     for (i = 1 to 10)
3.         Recv(*);
4. } else if (1 <= rank <= 10) {
5.     if (rank == 1) StartExp();
6.     Send(0);
7.     if (rank == 1) EndExp();
8. }
```

Figure 3.8. *Solution for Scenario 3: Symmetry Reduction*

concurrent programs, we can see the number of preemptions as the number of order constraints. The philosophy behind this definition is that the higher the depth of a bug, the fewer schedules contain this bug and the harder for dynamic verification tools to detect it.

We define the depth of a bug in an MPI program with the same philosophy. The definition of the depth is

$$\frac{number\ of\ the\ buggy\ schedules}{total\ number\ of\ the\ schedules}$$

Considering the example shown in Figure 3.9, the process 0 issues wildcard receives for other processes. An assertion will be raised if the first wildcard receive matches the send from the process k. We suppose that the raising of an assertion denotes the happening of a bug. Let n be the number of the processes in this example program, the total number of schedules is $(n-1)!$. The number of buggy schedules (the schedules raise the assertion) is $1 * (n-2)!$. Therefore, the depth of this bug is

$$\frac{(n-2)!}{(n-1)!} = \frac{1}{n-1}$$

Our definition of the bug depth shows the chance of dynamic verification tools hitting the bug in MPI programs. The smaller the bug depth, the harder it is for the tools to find the bug.

3.4.2 Buggy Benchmarks

3.4.2.1 mpiBlast

The mpiBlast is the parallel implementation of NCBI BLAST. It uses MPI to distribute the genome databases queries and modifications. It has been widely used in the research of biology relative fields. We planted a bug into mpiBlast. The depth of this bug is $3/13$. To apply our FPRS heuristic to detect the bug, we set a depth bound to the

```
1.  if (rank == 0) {
2.      for (i = 0 ; i < (nprocs - 1) ; i++) {
3.          Recv(*, &status);
4.          assert (!(i == 0 && status.MPI_SOURCE == k)); }}
5.  else {
6.      Send(0);
7.  }
```

Figure 3.9. The Depth of a Bug

primary loop body of it. We start up mpiBlast with 4 processes. Intuitively, we set the depth bound to 5 which is greater than the number of the processes.

3.4.2.2 π computation

This benchmark uses MPI to implement a well known Monte Carlos method for computing π. This program adopts the *master-slaves* structure. One of the processes in this MPI program responds to distribute the computation tasks. Each of the slaves handles the assigned computation task, sends back the result to the master, and handles a new task if there are any pending in the master's queue. We can see the slaves processes are *symmetric* and apply our FPRS heuristic. We start up this π computation with four processes, one master and three slaves. We did the experiments on π computation benchmark twice and seeded two different bugs in the two experiments. The depth of the first bug we seeded is $1/56$. The depth of the second bug we seeded is $3/56$.

3.4.2.3 matrix multiply

This buggy program (shown in Figure 3.10) does not simply do the matrix multiply, it calls the multiply computation a verified subroutine. The function *MM* is a subroutine which does the matrix multiply. We assume that it has been verified as a bug-free subroutine. In fact, as another experiment result shows later, it costs a lots of time to verify the function *MM*. In our program shown in Figure 3.10, we call *MM* twice and call another unverified routine *UnverifiedRoutine* between them. We apply our FPRS heuristic as the *compositional testing*. Since we are only concerned about verification on *UnverifiedRoutine*, we restricted the exploration region to this function. Again, we start up this program with 4 processes. The depth of the bug we seed in *UnverifiedRoutine* is just $1/6$.

```
1.  MatrixMultiply () {
2.       MM();
3.       Barrier();
4.       UnverifiedRoutine();
5.       Barrier();
6.       MM();
7.  }
```

Figure 3.10. Our Matrix Multiply Benchmark

3.4.3 Unchanged Benchmarks

3.4.3.1 matrix multiply

This version of matrix multiply is the function *MM* in Figure 3.10. (In previous buggy matrix multiply benchmark, we assume that *MM* is bug-free.) This benchmark uses a typical *master-slaves* structure to multiply two matrices. We observed that the slaves processes are symmetric. Therefore, we restrict the exploration region to one of the slaves.

3.4.3.2 diffusion2d

This is one of the benchmarks in *Functional Equivalence Verification Suite* [13]. We add an *MPI_Barrier* at the end of a loop in *diffusion2d* benchmark to fix the functional equivalence problem (to be equivalent to the sequential version). Beside that, we do modify this benchmark. Since all processes are synchronized by *MPI_Barrier* each iteration, we can restrict the depth bound of the loop to be one.

3.4.4 Results

We show our experiment results for buggy benchmarks in Table 3.1. We show our experiment results for unchanged benchmarks in Table 3.2. Each column in these tables shows the results of certain version of ISP (the original or FPRS) testing different benchmarks. Each row in these tables shows the results of certain benchmarks tested under different versions of ISP. The benchmark π *computation 1* is the pi computation benchmark which seeded the $1/56$ depth bug. The benchmark π *computation 2* is seeded the $3/56$ depth bug. The notation yes/n denotes that ISP detects a bug by n^{th} replays. The notation no/n denotes that ISP detects no bug and reports *bug-free* after n replays. The notation $timeout/n+$ denotes that ISP does not reply *yes or no* before our defined time limit. It replays n times within time bound.

Table 3.1. Experiment Results of Buggy Benchmarks

benchmark	original isp	FPRS
mpiBlast	timeout / 14747+	yes / 81
π **computation 1**	yes / 145	yes / 6
π **computation 2**	yes / 181	no / 27
matrix multiply	timeout / 4824+	yes / 6

Table 3.2. Experiment Results of Unchanged Benchmarks

benchmark	original isp	FPRS
matrix multiply	timeout / 83833+	no / 1849
diffusion	timeout / 114659+	no / 1680

3.4.5 Discussion

Our FPRS algorithm provides a heuristic search space reduction. The nature of the heuristic reduction is the risk of losing reliability. Specifically, we are under the risk of losing bugs. For example, in Table 3.1, ISP with FPRS reduction cannot deterministically detect the second version bug we seeded in the π computation benchmark. The reason why we lost bugs is that our algorithm makes ISP focusing the search on some specific regions. For those noninteresting regions, ISP will quickly bypass them. Therefore, if the MPI calls in the noninteresting regions involved in the buggy schedules, we may lose these bugs.

Random-walk is our compensation of the potential bugs losing. Randomization was empirically shown to improve the discovery in dynamic model checking [25]. We tested the second version bug in the π computation by FPRS ISP again and found the bug in 10 rounds of execution. This idea is very similar to our hybrid programs deterministic replay solution. We completely search the MPI interleaving by fixing one thread schedule. If we want to explore the both thread level and process level schedules, we can randomly permute the thread level schedules and verify the MPI schedules under this permutation. This approach prevents the mixed thread and process schedules from exponentially expanding. In the case of search reduction, for the noninteresting regions, we fix the interleavings. But the random-walk preserves the chance to find bugs in those noninteresting regions and exponentially reduces the search space.

Sampling the noninteresting by random-walk is a *nonbios* approach. In our experiment results, we can observe that the depth of the bug does not imply the performance of both versions ISP. This is because that ISP exploys the *DFS approach traversal* to exhaustively traverse the interleaving three. The *DFS approach traversal* is a *bios search* of tree structures. That is, the schedules we explore in a sequence of rounds are close to each other. Bugs could also be *bios existed* in the interleaving three. That is, the buggy schedules could be close in the interleaving tree. Consequently, the bug depth may not match the performance of ISP. However, our random-walk method

performs a *nonbios search*. It gives us a higher chance to capture the lost bugs in a few reverifications of ISP.

CHAPTER 4

CONCLUSION AND FUTURE WORK

Hybrid message passing programs are being used in cluster machines and multicore processors. Consequently, we need to pay more respect to the reliability and robustness of hybrid message passing programs. We need to improve the scalability and coverage of our dynamic verification tools. In this thesis, we contribute the following ideas to solve the challenges and show they are successful by experiments:

- Determinize the thread level schedules while exploring process level schedules by an external race replaying helper for ISP. We suggest this approach for future hybrid program verification.

- Dramatically reduce the search space of schedules by focusing exploration. Our FPRS heuristic can reduce the search space while highly preserving the bugs.

One limitation of our current hybrid scheduler is that we do not explore thread schedules; doing so will easily result in a cross-product effect between MPI schedules and thread schedules. There are two promising heuristics that can help avoid computing the product: (i) incorporate preemption bounded searching as in [5], or (ii) employ random walk in conjunction with schedule enforcement. These will be further studied in our research.

Another future work of our hybrid scheduler will examine how to make ISP capable of dynamically verifying more types of hybrid programs such as MPI-OpenMP, MPI-CUDA, etc. Dealing with mixed MPI/OpenMP opens up a number of challenges. Since OpenMP implementations differ significantly from each other, it would be extremely helpful (from the dynamic verification perspective) if OpenMP implementors were to expose some scheduling related entry-points into the OpenMP runtime so that hybrid dynamic verifiers can resort to a single standardized solution in determinizing the OpenMP behavior while exploring the MPI behavior.

Our immediate future work for the FPRS heuristic is automatic generation of heuristic input. For now, users need to manually identify the focusing regions. We believe that

it is trivial for the developers who construct the tested programs. However, it could be nontrivial for those users who are not familiar with the programs they tested. Therefore, our automatic heuristic input generator should achieve the following goals:

- Dynamically detect the scale of loops and decide their sampling bounds.
- Identify the subroutines in the tested programs.
- Detect the semantic symmetric execution.

With our input generator, users still need to give heuristic input such as the name of verified subroutines. However, we should minimize the information required from users.

REFERENCES

[1] Eddy murphi. http://www.cs.utah.edu/fv/mediawiki/index.php/Eddy_Murphi.

[2] AANANTHAKRISHNAN, S., DELISI, M., VAKKALANKA, S., VO, A., GOPALAKRISHNAN, G., KIRBY, R. M., AND THAKUR, R. How formal dynamic verification tools facilitate novel concurrency visualizations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2009), vol. 5759 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 261–270.

[3] ALTEKAR, G., AND STOICA, I. Odr: output-deterministic replay for multicore debugging. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), pp. 193–206.

[4] BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not. 45*, 3 (2010), 167–178.

[5] COONS, K. E., BURCKHARDT, S., AND MUSUVATHI, M. Gambit: effective unit testing for concurrency libraries. In *PPoPP '10* (2010), pp. 15–24.

[6] FLANAGAN, C., AND GODEFROID, P. Dynamic partial-order reduction for model checking software. *SIGPLAN Not. 40* (January 2005), 110–121.

[7] GODEFROID, P. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1997), POPL '97, ACM, pp. 174–186.

[8] HAVELUND, K., AND PRESSBURGER, T. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT) 2*, 4 (2000), 366–381.

[9] HOLZMANN, G. J. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[10] HTTP://EN.WIKIPEDIA.ORG/WIKI/2003_NORTH_AMERICA_BLACKOUT.

[11] HTTP://EN.WIKIPEDIA.ORG/WIKI/CRAY_XT5.

[12] HTTP://EN.WIKIPEDIA.ORG/WIKI/TIANHE I.

[13] HTTP:HTTPS://VSL.CIS.UDEL.EDU/TRAC/FEVS/.

[14] HTTP://JAVAPATHFINDER.SOURCEFORGE.NET.

[15] HTTP://RESEARCH.MICROSOFT.COM/EN US/UM/PEOPLE/MBJ/MARSPATHFINDER/.

[16] HTTP://SPINROOT.COM/SPIN/WHATISPIN.HTML.

[17] HUNT, G., AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HODSON, O.,

LARUS, J., LEVI, S., STEENSGAARD, B., TARDITI, D., AND WOBBER, T. Sealing os processes to improve dependability and safety. *SIGOPS Oper. Syst. Rev. 41*, 3 (2007), 341–354.

[18] LEVESON, N., AND TURNER, C. Investigation of the therac-25 accidents. *IEEE Computer 26*, 7 (1993), 18–41.

[19] MELATTI, I., PALMER, R., SAWAYA, G., YANG, Y., KIRBY, R. M., AND GOPALAKRISHNAN, G. Parallel and distributed model checking in eddy. *Int. J. Softw. Tools Technol. Transf. 11*, 1 (2009), 13–25.

[20] MINTCHEV, S., AND GETOV, V. Pmpi: High-level message passing in fortran77 and c. In *High-Performance Computing and Networking*, B. Hertzberger and P. Sloot, Eds., vol. 1225 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997, pp. 601–614. 10.1007/BFb0031632.

[21] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 446–455.

[22] MUSUVATHI, M., AND QADEER, S. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not. 42*, 6 (2007), 446–455.

[23] MUSUVATHI, M., QADEER, S., AND BALL, T. Chess: Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation* (2007), vol. 4407/2007.

[24] PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 226–237.

[25] RUNGTA, N., AND MERCER, E. G. Clash of the titans: tools and techniques for hunting bugs in concurrent programs. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems* (New York, NY, USA, 2009), ACM, pp. 1–10.

[26] SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. *Operating System Concepts*. Wiley Publishing, 2008.

[27] THOMAS BALL, K. E. C., AND MUSUVATHI, M. Preemption sealing for efficient concurrency testing. Tech. rep., Microsoft Research, 2009.

[28] VAKKALANKA, S., DELISI, M., GOPALAKRISHNAN, G., KIRBY, R. M., THAKUR, R., AND GROPP, W. Implementing efficient dynamic formal verification methods for mpi programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)* (2008), pp. 248–256. LNCS 5205.

[29] VAKKALANKA, S., GOPALAKRISHNAN, G., AND KIRBY, R. M. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification (CAV 2008)* (2008), pp. 66–79.

[30] VO, A., VAKKALANKA, S., DELISI, M., GOPALAKRISHNAN, G., KIRBY, R. M., , AND THAKUR, R. Formal verification of practical mpi programs. In

Principles and Practices of Parallel Programming (PPoPP) (2009), pp. 261–269.

[31] VUDUC, R., SCHULZ, M., QUINLAN, D., AND DE SUPINSKI, B. Improving distributed memory applications testing by message perturbation. In *PADTAD* (July 2006).

[32] YANG, Y., CHEN, X., GOPALAKRISHNAN, G., AND KIRBY, R. M. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN* (2007), D. Bosnacki and S. Edelkamp, Eds., vol. 4595 of *Lecture Notes in Computer Science*, Springer, pp. 58–75. Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings.