1 Introduction

Writing bug-free multi-threaded programs is hard. Bugs in these programs have various types such as deadlock, livelock, race condition, starvation, and so on. These bugs are hard to be detected because they may only be excited by certain interleaving of instructions. Most of the time, we can not find the bugs by only observing the program outputs since the buggy interleaving of instructions will not always happen. Even though we had observed the problem occurred, it often takes us lots of time to debug.

Work-stealing is one kind of method to achieve workload balance on distributed system or shared-memory system. It is a dynamic and distributed algorithm and especially works well on shared-memory system. I believe that work-stealing scheduling would be the trend for future process/task scheduler on CMP system.

This project will create a simple work-stealing simulator. Intuitively, it is be a multi-threaded program and it is a good example for us to try on Inspect [7], a runtime model checking tool, to help up finding out bugs in the simulator. In this project, we will show our experience of debugging a multi-threaded program with tool.

2 Study of Task Migration and Work Stealing

For the load sharing problem, it has two solutions and is discussed under two environments. The two solutions are work-sharing and work-stealing. And the two environments are the distributed system and multiprocessors share-memory system. In work-sharing, when a processor generates a new task, or thread, a central scheduler responses to allocate it to an underutilized processor in order to distribute the workload. In work stealing, once a processor generates a new task, it will be kept in the local queue or put the current task into the queue and run the new one. For an idle processor, it will attempt to find a task to run from others local queue.

Task migrating implies the data affinity on the original processor would be broken. When the task migrated to another processor, it will run into the initial burst of cache misses again because the working set of that task may still resided in the original processors cache. It also causes the degradation in the whole system because of the increased of bus traffic due to cache misses. It may also companies with the increasing number of cache invalidations due to the copies of data also resided in the original processors cache. In [4], it shows that the cache reload time in existing systems can be significant. These penalties could be more significant in non-uniform memory access (NUMA) system. However, fixing tasks which they have affinity on one processor will causes the system suffered from load imbalance. That means tasks will waiting for one busy processor while other processors are idle. Therefore, there is a trade off between adhering to processor affinities and workload balance.

The load sharing problem discussed under distributed system and multiprocessors shared-memory system has several differences [5]. The first difference is that it causes additional network delay for a processor in the distributed system to migrate a task. In shared-memory system, it allows an idle processor to search other processors local queue and remove a waiting task without disturbing the victim processor. The second difference is that the usage of the communication medium distributed system, that is network, is relative lower than in shared-memory system, that is shared memory and inter connect network. The additional traffic on shared-memory system would be relatively higher than on distributed system. However,
migration of a task that had been started execution poses considerable difficulties in many distributed systems [3]. On the other hand, in [5], it illustrated the potential for improvements in system performance even with large migration cost under shared-memory system.

3 Implementation of the Simulator

3.1 Work Stealing Deque Implementation Baseline

The basic concept of operating on work stealing deque (double-end-queue) is that every core owns a deque to contain their pending tasks. Note that in this report, we use the term "task" as thread. The owner core of the deque would always accesses it from the bottom. On the other hand, the core which stealing thread always steals from the top of the deque. We suppose that one core would only execute one task a time. When the current executing task had done, the owner pops a pending task from the bottom and executes it. When the current task generates a child thread, the core will pushs the parent thread into the bottom of the deque and executes the new-created child thread. There may has other various ways of operations of work stealing deque. Yet, in this project, we will adopt the basic concept of operations described above.

The ABP, Arora, Blumofe, and Plaxton, algorithm [6] has been gaining popularity as the multiprocessor load-balancing technology of choice in both industry and academia. Non-blocking means that the popBottom and popTop function will not be stalled because of completing the top element in the queue. The key idea of ABP algorithm is using CAS, compare-and-swap, instruction to decide who win the last element in the queue. The victim and thief will record the index of top before extracting the top element. After gaining the top element, they will try to CAS the current top index of the queue with the recorded top value plus one. For example, the thief may have the recorded top index as t and try to compare the current top index with t and swap with t+1. Once the CAS instruction in popBottom/popTop success, it means the last element extracted successfully. If CAS instruction fail, the popBottom/popTop would be just aborted and the Empty returned by the function.

The main drawback of ABP algorithm is that it bases on the fixed array/deque size. It implies that this method is under the risk of memory overflow or memory inefficient usage. Instead of implement the deque in an fix-sized array, Dynamic-Sized Non-blocking Work Stealing Deque [1] implement the deque by using a doubly linked list of nodes, and each node is a short array which could dynamically allocated from and free to a shared pool. The basic idea of solving competition is the same as ABP algorithm, using CAS instruction.

Dynamic circular work stealing deque [2] is the improved version of [1]. It is simpler to implement and works more elegant. In [1], it uses a structure to record the top index and the tag of the work stealing queue. The tag is used to prevent the ABA problem since the stealer will increase the top index and the victim needs to reset it to zero. But it means that the tag should be put together with the top index and CAS with it. It may need additional encoding technique to make them fit into the size which CAS instruction accepts. Instead using tag, [2] uses a circular array to solve this problem. The top index would be always increased. But the index of the elements in the circular array would be computed as "index % arraysize".

3.2 Implementation for Pthread with Self-suspend and Resume

POSIX thread is one of the most popular threading API, pthread programs could also be verified by Inspect. However, it doesn't provide suspend and resume functions. There are lots of complex ways to implement these functions. In our simulator design, it only requires self-suspend and resume. Self-suspend means that only the thread itself could decide when to suspend. To implement them, it simply needs a conditional variable to wait or signal on it. Therefore, we designed a thread block structure called PSSThread which comes with each pthread. This structure for each pthread contains the function (task) which pthread will
typedef struct PSSThread {
    int psstid;
    void *(*func)(void *);
    void *func;
    int now_pid;
    pthread_t shell_thread;
    pthread_cond_t cond_suspend;
}pssthread;

void *pssthread_shell(void *psst) {
    // suspend
    pssthread_self_suspend((pssthread *)psst, NULL);
    // start the thread
    pthread_create(shell_thread....); // create the assigned pthread
    pthread_join(shell_thread....);
}

void pssthread_self_suspend(pssthread *psst, pthread_mutex_t *s_mutex) {
    pthread_cond_wait(&((psst->cond_suspend), s_mutex));
}

void pssthread_resume(pssthread *psst) {
    pthread_cond_signal(&((psst->cond_suspend));
}

Table 1: Definition of PSSThread structure, pssthread_shell, self-suspend function, and resume function

perform, the argument of it, the id of the core (simulated processor) current resided, and the condition variable used by self-suspending and resuming. Table 1 shows the PSSThread block structure, self-suspend function, resume function, and a pssthread_shell. If we want to create a pthread which can self-suspend and resume, we first create its’ PSSThread block, give the required parameters, then create a pthread which executes function pssthread_shell. Once the pthread which executes pssthread_shell created, it will first suspend itself, wait some other waking it up, then run the function which you assigned to the pthread. In Table 1 you can observe that the self-suspend function just waits a condition variable, and the resume function just signal the condition variable to wake up the thread which self-suspended.

3.3 Basic Mechanism of the Simulator

We will have one deque per simulated core. Each simulated core is just basically a task scheduler which dynamically pops a work from the deque and resumes it. The tasks which stay in deques are suspended and wait for task schedulers to resume them. Both the schedulers and tasks need to self-suspend and be resumed. Therefore, they are implemented by using pthread with PSSThread block. The work stealing algorithm in the simulator uses the most basic and common one: while one thread create a child thread, it push itself into the deque in which the core current located then push its child thread into the deque. Note that the terms push thread and pop thread actually means operating on the PSSThread blocks among deques. Items in deques are actually PSSThread blocks. After creating the child thread, the thread will first resume the scheduler responses for the core it located and suspend itself. Since the child thread would also be a pthread with PSSThread block and execute on pssthread_shell, it will suspend itself upon created. Table 2 shows the pseudo code for the scheduler and the thread function which create a child
void *TASK_SCHEDULER(void *s_a) {
    dcde_task_block *thisTask;
    dcde *de = ((scheduler_arg *)s_a)->de;
    pssthread *me_psst = ((scheduler_arg *)s_a)->psst;
    while (1) {
        if ((thisTask = dcde_popBottom(de)) != NULL) {
            pssthread_resume(thisTask->psst);
            pssthread_self_suspend(me_psst, NULL);
        } else { // thisTask == NULL - no more tasks
            if (thisTask = STEAL_FROM_OTHERS() ) {
                pssthread_resume(thisTask->psst);
                pssthread_self_suspend(me_psst, NULL);
            } else { // no task to steal
                return;
            }
        }
    }
}

void *Thread_Create_Child_Thread (void *my_arg) {
    (doing something....)
    pssthread_init(....); // init the pssthread info. Block
    pthread_create(....); // create child thread
    pushBottom( me ); // push itself into the deque
    pushBottom( child ); // push child into the deque
    resume( scheduler ); // resume the scheduler reponse for the deque
    self_suspend();
}

Table 2: task scheduler and (create a child) thread function

Beyond schedulers and tasks, there is a global tasks allocator which brings up the whole system. Before everything starting, it creates schedulers for simulated cores, create tasks initially for scheduler to wake up, push the task into a deque, and finally wake up all schedulers to start the simulation. After resuming all schedulers, it will join all schedulers. That means, it will not involve in any operations for the work stealing simulation. Figure 2 shows the works of global tasks allocator.

4 Inspect Help Finding Bugs

This section describes how we use this simulator to run on a merge sort program and fix bugs/faulty designs in the simulator. To find out the bugs easily by simplify the problem scalar without losing general, I create two schedulers to run the merge sort. One has a thread in its deque initially and another has a empty deque and will go to steal task from others’ deque. We also provide all of our codes for our experiment in this section. Please refer to the section: appendix A.

4.1 Deadlock in Original Self-suspend and Resume design

The original self-suspend and resume design (Table 1) causes deadlock. Even without tools like Inspect, we can still observe the deadlock occurrence. But using tool like Inspect help us figure out how the bug
popBottom from deque and wake up a task

and wake up a task

waked up by scheduler

create child task

pushBottom itself (thread info.) into deque

pushBottom child into deque

wake up scheduler

suspend

awaked by task

popBottom from deque and wake up a task

suspend

awaked by task

Figure 1: Interaction between scheduler and task

set up array pool, deque...

create scheduler 0

create scheduler 1

create the task (merge sort program)

put the task into scheduler 0's deque

resume scheduler 0

resume scheduler 1

join schedulers

Global Allocator

created by global allocator

created by global allocator

created by global allocator

awaked by global allocator

awaked by global allocator

resume a task (popBottom)

no task, steal a task

Figure 2: Illustration of Global Allocator
After T:0 putting the task into T:1's deque, it is now starting to resume scheduler 0 (T:1) and scheduler 1 (T:2).)

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event Id</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>T:0 event-type:</td>
<td>cond_signal</td>
<td>228</td>
</tr>
<tr>
<td>cond-id:119</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T:0 event-type:</td>
<td>cond_signal</td>
<td>228</td>
</tr>
<tr>
<td>cond-id:123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T:1 event-type:</td>
<td>mutex_unlock</td>
<td>214</td>
</tr>
<tr>
<td>mutex-id:118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T:2 event-type:</td>
<td>mutex_unlock</td>
<td>214</td>
</tr>
<tr>
<td>mutex-id:122</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T:3 event-type:</td>
<td>mutex_unlock</td>
<td>214</td>
</tr>
<tr>
<td>mutex-id:124</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Inspect error trace

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Scheduler 0</th>
<th>Scheduler 1</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create scheduler 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create scheduler 1</td>
<td>Created by allocator</td>
<td>Created by allocator</td>
<td></td>
</tr>
<tr>
<td>Create Task</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put task into scheduler 0's deque</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resume scheduler 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resume scheduler 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join schedulers</td>
<td>Self-suspend</td>
<td>Self-suspend</td>
<td>Self-suspend</td>
</tr>
</tbody>
</table>

Table 4: Inspect error trace

happened quickly. Table 3 show critical part of the message shown in the trace file created by Inspect. Refer to the Figure 2 after the global allocator creating scheduler 0 (T:1 in Table 3), scheduler 1 (T:2 in Table 3), and task (T:3 in Table 3), it puts the task into scheduler 0's deque and resume schedulers. In Table 3 Inspect shows that the global allocator resumes the schedulers before they suspend themselves. Note that the resume function in the original design is actually signaling a conditional variable. The line, "T:1 event-type: mutex_unlock mutex-id:118 line:214", in Table 3 means the T:1 suspend it self and give up a current hold mutex lock at the same time. By using error-tracing integrating with emacs, this line is actually the signaling of conditional variable. Line, "pthread_cond_wait(pthread_cond_suspend), s_mutex;" is the original design of self-suspend. The released lock is s_mutex. Since the global allocator try to resume schedulers before they suspend themselves, the resume function which global allocator called actually didn't wake up any thread. Also, there is no one to wake up schedulers after they suspending. Consequently, there is also no one to resume the task (T:3 in Table 3). Table 4 and Figure 3 give the snapshot for the deadlock situation.

4.2 Discussion About the Bug Inspect Point Out

By observing the bug shown up in the previous section, as shown in Figure 3, we can say that the the problem is global allocator fail to guarantee to resume the schedulers after they suspending themselves. Analogy, we can figure out that scheduler can not guarantee to resume a task after its' suspending, as shown in Figure 4(a). Also, task does not know whether the scheduler had suspended before it going to resume it, as shown in Figure 4(b).

There are two ways to solve this problem. One way is making some critical group of codes executed atomically. For example, we would like to make create pthread and self-suspend these two instructions executed like an atomic instruction. Another way to solve the bug is making block-resume. That is, the resume function will not return if it fail to wake up the target.
Global Allocator

create scheduler 0
create scheduler 1
create task
resume scheduler 0
resume scheduler 1
join

Scheduler 0

Scheduler 1

Task

resumed by scheduler

Figure 3: Bug illustration

Scheduler

Task A

Stealer

Scheduler

Task B

(a) scheduler (or stealer) fail to
resume a task

(b) task fail to resume the scheduler

Figure 4: Illustration for two similar bugs
4.3 Making Atomic Instruction

4.3.1 Using Shared Lock to Build Critical Sections

The idea of making atomic operations is shown as pseudo codes in Table 5, Table 6, and Table 7. We use a shared lock to build critical sections. The term atomic in these tables means the lock shared by all cores and all threads. That means instructions sequence protected by the shared lock cannot be interrupted by any core in the system. In Table 5, it shows how to make sure the create-suspend instructions hair atomic. It prevents the task being resumed before the self-suspend just after created. In Table 6, it makes sure the scheduler resume a task and suspend itself in atomic. By this way, we could prevent the resume instruction coming before the suspend. It is similar in Table 7, we put resume and suspend instructions into the critical section. We also put pushBottom instructions into critical section because that we have to prevent the stealer popTop, entering critical section, and resume the task before the task entering the critical section and suspending itself.

Intuitively, the task should also enter the critical section and resume the scheduler before it exiting. The code is quite trivial and not shown.

Table 7: Making Atomic Operations for parent thread

<table>
<thead>
<tr>
<th>Parent thread</th>
<th>Child thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_lock(atomic);</td>
<td>self-suspend()</td>
</tr>
<tr>
<td>&lt;Enter Critical Section&gt;</td>
<td>pthread_cond_wait(cond, atomic);</td>
</tr>
<tr>
<td></td>
<td>&lt;Exit Critical Section and Wait Conditional Variable at Same Time&gt;</td>
</tr>
<tr>
<td>pthread_create();</td>
<td></td>
</tr>
</tbody>
</table>
| | }

Table 5: Making Atomic Operations for "create-suspend"

<table>
<thead>
<tr>
<th>Pseudo Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_lock(atomic)&lt;Enter Critical Section &gt;</td>
</tr>
<tr>
<td>// resume a task</td>
</tr>
<tr>
<td>// self-suspend</td>
</tr>
<tr>
<td>pthread_mutex_unlock(atomic)&lt;Exit Critical Section &gt;</td>
</tr>
</tbody>
</table>

Table 6: Making Atomic Operations for scheduler "resume-suspend"
4.3.2 Potential Risk of Making Atomic Operations in our Simulator

The potential problem caused by this idea is improper usage of API. In our code for global allocator, we used the API improperly while creating schedulers as this way:

```c
for () pthread_mutex_lock(); pthread_create(scheduler);
```

The risk or hardship of making atomic operations is that we sometime need to enter the critical section in parent thread and exit in child thread. Under this risk, we need to guarantee that the parent thread will enter the critical section again after the child thread had left it. However, making this order is not easy in implementation level. Our solution is just making another shell function for schedulers that they will not suspend upon created. Of course, by doing so, we have to create schedulers after the initial task had been push into deque.

Current Inspect fail to verify codes written with atomic operations. We also fail to try on C#/CHESS because that it is almost impossible to enter the critical section in one thread and exit in another in C# program.

4.4 Synchronizing Self-suspend and Resume

4.4.1 Implement Block-resume by Using Spin Check

Previously we had known that it is resuming function which didnt match the suspending call cause problem. To solve this problem, we implemented block-resume by using spin check. Table[8] shows the newly design of PSSThread block information, self-suspend function, and block-resume function. The flag_suspend indicates the status of the thread, 1 for suspended and 0 for non-suspended. There is a while loop in function block-resume to check the flag and signal the condition variable once the flag been set.

Unfortunately, Inspect fail to check bugs in programs with such design. Inspect adopts run time model checker and uses a depth-first strategy to explore the state space. Therefore, Inspect can only handle programs that can terminate in a finite number of steps [7]. If we using spin check to implement the block-resume, it causes potential starvation problem and the depth-first search for state space results in endless searching. Table[9] shows the starvation situation in this design.

4.4.2 Suspend-Resume Synchronization by Using Barrier

Instead using spin check for block-resume function, another good way to synchronize two threads is using barrier. However, programs with barrier instructions are not supported by current version of Inspect. Fortunately, Inspect supports broadcast instruction. With broadcast instruction, implementing barrier is trivial. Table[10] shows the decide of the barrier and the redesigned code for self-suspend and block-resume.

We expect that this is the best solution for the bug described in section previous section. Unlike building atomic operations, we need to also considering about problem like improper usage of API. We also need to pick up instructions carefully into the critical section. However, with synchronizing resume and suspend by using barrier, we never worry about problem that the resume call does not match the correspond suspend call.

Current Version of Inspect fail to verify codes for spin check synchronization and barrier synchronization. For porting to C# and tested by CHESS, the CHESS just hanged and can not continue to verify the program.

4.5 Missing Resuming Scheduler in Certain Path in Thread

Sometime even we had observed the deadlock occurred in our program, we may took lots of time to figure out what happened. Sometime the programmers just make mistakes in careless. They may eventually find
typedef struct PSSThread {
    int psstid;
    void *(*func)(void *);
    void *func_arg;
    int now_pid;
    pthread_t shell_thread;
    pthread_mutex_t set_suspend;
    pthread_cond_t cond_suspend;
    char flag_suspend;
} pssthread;

void pssthread_self_suspend(pssthread *psst, pthread_mutex_t *s_mutex) {
    pthread_mutex_lock(&(psst->set_suspend));
    psst->flag_suspend = 1;
    if (s_mutex == NULL) {
        pthread_cond_wait(&(psst->cond_suspend), &psst->set_suspend);
    } else {
        pthread_mutex_unlock(s_mutex);
        pthread_cond_wait(&(psst->cond_suspend), &psst->set_suspend);
        pthread_mutex_lock(s_mutex);
    }
    pthread_mutex_unlock(&(psst->set_suspend));
}

void pssthread_block_resume(pssthread *psst) {
    while (1) {
        pthread_mutex_lock(&(psst->set_suspend));
        if (psst->flag_suspend == 1) { // could resume
            psst->flag_suspend = 0;
            pthread_cond_signal(&(psst->cond_suspend));
            pthread_mutex_unlock(&(psst->set_suspend));
            break;
        } else {
            pthread_mutex_unlock(&(psst->set_suspend));
        }
    }
}

Table 8: Block-resume by using spin check

<table>
<thead>
<tr>
<th>Choice of Inspect</th>
<th>Thread A: block-resume</th>
<th>Thread B: self-suspend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Competing: Thread A win</td>
<td>pthread_mutex_lock();</td>
<td>pthread_mutex_lock();</td>
</tr>
<tr>
<td>Thread A in critical section</td>
<td>flag_suspend == 0;</td>
<td>pthread_mutex_lock();</td>
</tr>
<tr>
<td>Thread A in critical section</td>
<td>pthread_mutex_lock();</td>
<td>pthread_mutex_lock();</td>
</tr>
<tr>
<td>Competing: Thread A win</td>
<td>pthread_mutex_lock();</td>
<td>pthread_mutex_lock();</td>
</tr>
<tr>
<td>Thread A in critical section</td>
<td>flag_suspend == 0;</td>
<td>pthread_mutex_lock();</td>
</tr>
<tr>
<td>Thread A in critical section</td>
<td>pthread_mutex_lock();</td>
<td>pthread_mutex_lock();</td>
</tr>
</tbody>
</table>

Table 9: Starvation situation
typedef struct M_Barrier {
    pthread_cond_t cond_barrier;
    pthread_mutex_t barrier_lock;
    int n_syn;
    int n_now;
} m_barrier;

void m_barrier_barrier(m_barrier *bar) {
    pthread_mutex_lock(&(bar->barrier_lock));
    bar->n_now++;
    if (bar->n_now != bar->n_syn) {
        pthread_cond_wait(&(bar->cond_barrier), &(bar->barrier_lock));
    } else { // wake up all
        pthread_cond_broadcast(&(bar->cond_barrier));
        m_barrier_reset(bar, bar->n_syn);
    }
    pthread_mutex_unlock(&(bar->barrier_lock));
}

void pssthread_self_suspend(pssthread *psst, pthread_mutex_t *s_mutex) {
    m_barrier_barrier(&(psst->suspend_bar));
}

void pssthread_block_resume(pssthread *psst) {
    m_barrier_barrier(&(psst->suspend_bar));
}

Table 10: Implementation of barrier instruction and suspend, resume functions redesign
Array Merge_Sort() {
    if (array_length < threshold) return Binary_Sort();
    else {
        // split the array into two sub-array
        Merge_Sort(array_part1); Merge_Sort(array_part2); // merge two sorted sub-array
    } // resume scheduler  // return sorted_array; }

Table 11: buggy merge sort

out the bugs. But they could save their time if they use tools. In this section, we propose a bug that caused by programmer out of mind.

For merge sort, we would use binary sort if the length of the sorted array is short. The pseudo code of merge sort is shown in Table 11. If the array length less than a threshold, we return the result generated by binary sorting function. Or else, we sort by general merge sort method. Finally, we resume the scheduler before the function exit. The bug is that we forget to resume the scheduler before the function exits with binary sorting result, the if path in the program. It is not the problem caused by faulty suspend-resume design but programmers may try to find bug from where they think the bug should be.

New version Inspect may point out this bug clearly and help programmers on the right path of finding bugs.

5 Appendix

A Codes for Our Experiment

Our original design code:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_original.c

Atomic instructions set with Multi-lock a mutex:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_multi_lock.c

Solve bug by using atomic instructions set:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_atomic_solution.c

Block-resume function by using spin check:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_spin_check.c

Solve bug by using barrier instruction:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_barrier_solution.c

Missing resuming function:
http://www.cs.utah.edu/~wfchiang/Inspect_test_code/dcde_miss_resume.c

References
