Portable Data Race and Deadlock Detector

Kranthi K Gade, Prashant Puniya
Department of Computer Science
New York University
New York, NY-10012
{kranthi, prashant}@cs.nyu.edu

Abstract

Multithreaded programming requires lot of care on the part of the programmer to synchronize access to and preserve integrity of shared data. It is easy to make a mistake in synchronization which will in turn lead to problems such as data races and deadlocks, which are hard to detect when debugging. This article describes a tool which dynamically detects data races and deadlocks in multithreaded programs. This tool has been embedded in the C++ `pthread` and `libasync` libraries which renders it portable.

1 Introduction

With the demands increasing on present day applications, such as web servers etc., multithreaded programming has provided a very useful paradigm allowing programmers to introduce parallelism in software. This has had several advantages, for example it helps reduce response times and increase throughput of applications. However, multithreaded programming also introduces some completely new problems. Mistakes in synchronization between threads can lead to data races or deadlocks which make the task of debugging very difficult. This has made many programmers to refrain from using threads.

In this paper, we describe a tool that dynamically reports possibilities of data races and deadlocks to the programmer at run time. We have embedded this tool within the C++ `pthread` and `libasync` libraries. This allows the programmer to compile his program just as he would normally. However, for making this implementation possible we place a restriction on the shared variable in the multithreaded program. We require these variables to be `ref` or `ptr` variables.

Previously, Stephen Savage et al have described the same data race detection algorithm in their paper [1]. However, they use link time code modification to embed their data race detection code within the program being debugged. Specifically, they insert checks whenever a shared memory location is accessed by some thread. Thus, their tool is not portable. We have moved these checks within the C++ libraries to make them invisible to the programmer and also avoid the problem of detecting shared memory accesses in the link time code.

In the remainder of this article we first describe what data races and deadlocks are. Section 2 and 3 describe our data race and deadlock detection algorithms respectively. In section 4 we look at specific implementation details of our tool. Section 5 presents the results of the test cases we carried out. Section 6 gives our experiences during the course of this implementation.

1.1 Data Races

A data race occurs when two threads try to simultaneously access the same shared memory location and

- at least one of these accesses is a write.
- the threads don't use any explicit mechanism to synchronize their accesses.

Typically, locks or mutexes are used to achieve synchronized access to a shared variable. It is always advisable to protect every shared variable by some lock in all its accesses.

For the detection of data races our approach is similar to [1]. We enforce a simple locking discipline on shared memory accesses. Our tool requires each shared variable to be protected by at least one lock in all the accesses made to it. If at any point during the execution, this locking discipline is violated, we report a "potential data race" and warn the user. Although violation of this requirement might not result in an actual data race during program execution, it does indicate the possibility of one.

1.2 Deadlocks

Consider the simple situation below:

- Thread *T1* tries to acquire a lock *l2* when it already owns lock *l1*.
- Thread *T2* tries to acquire lock *l1* and it owns *l2*.

What happens in such a situation? Thread *T1* will wait for *T2* to release *l2* so that it can proceed with its execution, and vice versa. Thus, neither *T1* nor *T2* can make any progress and keep waiting. This is a simple example of a deadlock.

More generally, if the threads in a program do not consistently order their lock acquisitions, it may result in a deadlock. This is exactly what our tool checks to detect "potential deadlocks". Our tool requires all the threads to follow some consistent order when acquiring multiple locks. Again failure in doing this may not result in an actual deadlock, but our tool warns the user of such a situation.

2 Data Race Detection

In this section we describe in detail, how our tool detects potential data races. We use the algorithm that was proposed by Savage et al [1]. As we mentioned above, this algorithm enforces the simple locking discipline that every access to a shared variable must be protected by some lock.

For each shared variable v, the algorithm maintains the set $C(v)$ of candidate locks for v. This set contains all the locks which were held by each thread whenever it accessed v. $C(v)$ is initialized to the set of locks held by a thread t which performs the first write to v. This write can be interpreted to be the initialization of v. Upon each subsequent access by a thread t, we refine $C(v)$ by performing a set intersection of the current value of $C(v)$ with the locks held by t. If at any point during the execution, $C(v)$ becomes empty, we warn the user that there is a potential data race. The following pseudo-code describes how $C(v)$ is modified when a thread t accesses variable v.

```
Let locks_held(t) be the set of locks held by thread t.
If v has already been written to, then
      C(v) = C(v) ∩ locks_held(t);
else
      C(v) = locks_held(t);
If C(v) == ∅, then
      WARN ''Potential data race detected".
```

2.1 Extensions to the algorithm

The above simple version of data race detection algorithm has some shortcomings. We describe them below and provide extensions to handle such cases.

- *Single-thread access:* Suppose a variable v has been declared as shared but only one thread accesses it. For this case, there is no need for explicit protection of v by a lock. So even if the candidate lock set for v becomes empty, there is actually no cause for alarm.

- *Read-only variables:* In this situation, a variable v is initialized by some thread t_{init}. After this initialization, all threads (including t_{init}) have read-only access to it. Clearly there is no need to protect this variable by a mutex but the above described algorithm will warn the user of a potential data race if the candidate lock set for v becomes empty.

To account for these cases, we modify our algorithm by slackening the protection requirements in such situations. We maintain a state diagram wherein different states require varying levels of protection. Figure 1 illustrates how a variable v transitions between the different states and how the candidate lock set is modified in each state.

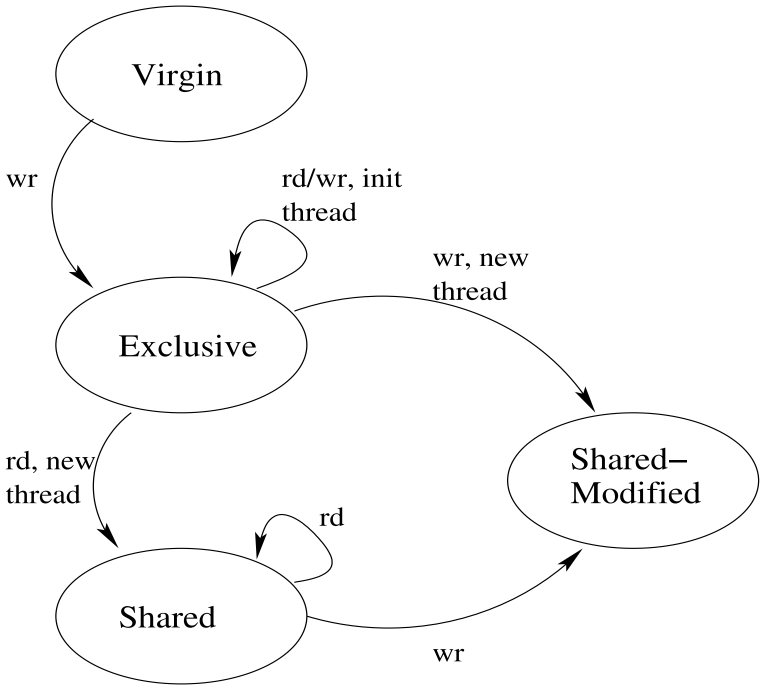

Figure 1: State Transition Diagram

- *Virgin:* This is the state in which v resides in when it has just been declared but no value has yet been written into it. v enters *Exclusive* upon first write by a thread t_{init}. This write can be interpreted as the initialization of v.

- *Exclusive:* All subsequent accesses by t_{init} keeps v in the *Exclusive* state and $C(v)$ is not refined upon such transitions. The variable v enters *Shared* state upon read by a thread other than t_{init} and *Shared Modified* state upon write by such a thread. The candidate lock set $C(v)$ is first initialized upon such transitions. This is to take care of the *Single Thread* access case that we described above so we initialize $C(v)$ only when it is first accessed by a thread other than t_{init}.

- *Shared:* When in this state, v has already been initialized by t_{init} but only been read by other threads. Any read access to v keeps it in this state and $C(v)$ is updated on each such access but data races are not reported even if $C(v)$ becomes empty because v could be a read-only variable. Any write access to v in this state takes it into *Shared Modified* state.

- *Shared Modified:* This state demands for full mutex protection on v since a non-initialization write has been performed to it. $C(v)$ is updated on any access to v in this state and data races are reported if $C(v)$ becomes empty.

3 Deadlock detection

Now we move on to describe the algorithm that we use for detecting "potential deadlocks" in a multithreaded program.

The order in which threads acquire multiple locks imposes a partial order on these locks. For instance, if a thread t_1 tries to acquire a lock l_2, when it already holds lock l_1, it imposes the partial order $l_1 > l_2$ on these two locks. In a more elaborate situation with many locks, the locks will form a directed dependency graph amongst themselves. If ever any of these partial orders is violated by any thread, then there is a potential for deadlock. In our simple situation above, if thread t_2 tries to acquire l_1 while holding lock l_2, it will violate the partial order $l_1 > l_2$ that was imposed by thread t_1. Next we describe how our algorithm checks for such "potential deadlocks".

We maintain a mutex dependency graph to hold this partial order information. Specifically, the vertices in this graph are the mutexes and there is a directed edge from vertex corresponding to l_2 to the vertex corresponding to l_1 if l_1 and l_2 satisfy the partial order $l_1 > l_2$. Whenever a thread tries to lock a mutex l_1, a vertex corresponding to l_1 is inserted in the graph if its not there already. Also for any lock l that t held at the time it tried to lock l_1, an edge is inserted from l_1 to l. As described above, this edge defines the partial order imposed between l_1 and l by t's attempt to lock l_1. Thus our mutex dependency graph grows dynamically with new vertices and edges being inserted in it at run-time.

Theorem 1 *If there is a deadlock then there is a cycle in the mutex dependency graph.*

Proof: If there is a deadlock, then there are threads $t_1, t_2 \ldots t_n$ and locks $l_1, l_2 \ldots l_n$ such that t_i owns l_i (for $i = 1, \ldots n$) and is waiting for $l_{(i+1) mod\ n}$. Assume that all the threads are following a consistent partial order, that is if t owns lock l_{k_1} and is trying to acquire l_{k_2} then the partial order $l_{k_1} > l_{k_2}$ is followed by all threads. Thus in our situation, threads $t_1, \ldots, t_{n-1}$ will impose $l_1 > l_2 > \ldots > l_n$. But t_n tries to acquire l_1 when it owns l_n which is a contradiction, since $l_1 > l_n$ by the partial order already imposed by the other threads.

Now in the mutex dependency graph there is an edge $l_i \rightarrow l_j$ if $l_j > l_i$ in the partial order. The above violation of partial order will induce the edge $l_1 \rightarrow l_n$ in the graph, when it already has the edges $l_1 \leftarrow l_2 \ldots \leftarrow l_n$. This defines a cycle in the mutex dependency graph. q.e.d.

Theorem 2 *If there is a cycle in the mutex dependency graph, then there is a potential deadlock.*

Proof: Suppose the cycle is $l_1 \rightarrow l_2 \rightarrow \ldots \rightarrow l_n$. By the partial order that we defined above, there is an edge $l_1 \rightarrow l_2$ iff l_1, l_2 satisfy $l_2 > l_1$. The existence of the cycle implies that $l_1 > l_n > l_{n-1} > \ldots > l_2 > l_1$. This violates the partial order and hence there is a potential deadlock. q.e.d

The above theorems imply that the task of detecting "potential deadlocks" is equivalent to detection of cycles in a directed mutex dependency graph.

4 Implementation

4.1 Data race detection

We embedded our data race and deadlock detection code into the `pthread` and `libasync` libraries. We impose the restriction that all the shared variables in the input program be of `ref` or `ptr` type. `libasync` provides `ref` and `ptr` as a mechanism for reference-counted deallocation of dynamically allocated data structures. `ref<foo> p` and `ptr<foo> p` behave essentially like `foo *p` but are reference counted. If the reference count goes to zero at any point, then the objects are garbage-collected. For more information about these, please refer to [2]. In the exposition that follows, we assume that the global variable is of type `ref`. Exactly the same things hold true for `ptr` type variables too.

There are standard operators defined in `ref` like $*$, $\rightarrow$ which are normally defined for pointers. So we can know when memory is being accessed within the code for these operators and insert appropriate checks for data race detection in there. The candidate lock set for a particular global variable is also maintained in its `ref` implementation. Whenever the variable is accessed using the operators $*$,$\rightarrow$, the lock set is appropriately refined (as we will explain later).

The set of locks that a particular thread holds is maintained in its `pthread` structure.

```
struct pthread {
    ...
    set my_locks;
    ...
};
```

Whenever a thread t successfully acquires a lock l, l is inserted into the lock set of t. When t releases l, it is removed from the lock set (as shown below).

```
pthread_mutex_lock(pthread_mutex_t *l) {
    ...
    if (lock is successful)
       set_insert(my_locks, l);
}

pthread_mutex_unlock(pthread_mutex_t *l) {
    ...
    if (unlock is successful)
       set_remove(my_locks, l);
}
```

To insert data race detection code, we maintain the following variables in the `ref` implementation : `lock_set` which gives the candidate lock set for the variable, `state` which gives the state in which the variable resides in the state diagram currently and `init_thread` is the thread which initialized the variable. The following gives the pseudo-code for the implementation of operator $\rightarrow$ inside `ref<T>`. The implementation of other operators like $*$ and $=$ is similar.

```
T *operator-> () {
    if (init_thread != get_current_thread()) {
       state = SHARED_MODIFIED;
    }
```

```
    refine(lock_set);
    return p;
 }
```

Here, `p` represents the underlying pointer to `T` that `ref<T>` maintains.

The function `refine` refines the lock set only if the variable is in *Shared* or *Shared Modified* states. If this is the first access by a thread t other than `init_thread`, then `lock_set` is initialized to the lock set of t, otherwise it performs a set intersection of `lock_set` with the lock set of t. If `lock_set` becomes empty, then `refine` flags an error only if the variable is in the *Shared Modified* state. The pseudo-code for `refine` is given below:

```
refine(lock_set) {
    if (state != EXCLUSIVE) {
        if (first access by a thread other than init_thread)
            lock_set = get_locks(get_current_thread());
        else
            lock_set = lock_set ∩ get_locks(get_current_thread());
        if ((lock_set == ∅) && (state == SHARED_MODIFIED))
            WARN "Potential data race detected."
    }
 }
```

Each thread maintains the set of locks it owns. The main operations performed on this set are `insertion/deletion` of locks acquired and locks released respectively. Also, each shared variable maintains a candidate lock set of the locks protecting it. The main operation here is `set_intersection`. To minimize the cost of all these operations, we decided to maintain our sets as *sorted doubly-linked lists.*

With this choice `insertions` or `deletions` in a set S of size n, can both be done in time $O(n)$. Also a set intersection between two such sets of sizes n and m, can be done in time $O(m+n)$. Thus we have made the most frequent operations, pretty cheap in terms of overhead.

We assumed in the code for operator $\rightarrow$ that we described previously that the user could be performing a write operation to the variable of type `ref<T>` (and hence the `state` changes to `SHARED_MODIFIED`). But what if the user wants to use operator $\rightarrow$ for only a read access? There is really no way to differentiate a read from a write access within the code for handling operator $\rightarrow$.

If the user is not going to use operators $\rightarrow, *$ for a write operation, we impose the restriction then he declare the variable to be of type `ref<const T>`. In this case, we can be sure that operators $\rightarrow$, $*$ can only be used for a read access to the variable. If a write needs to be issued on a variable v of type `ref<const T>`, it can be done only by assigning v to another `ref <const T>` variable. The code for handling operator $\rightarrow$ in `ref<const T>` gets modified as follows:

```
const T *operator-> () {
    if (init_thread != get_current_thread()) {
        state = SHARED;
    }
    refine(lock_set);
    return p;
 }
```

Note that the variable enters the `SHARED` state as opposed to the `SHARED_MODIFIED` state in the previous case. However there will be no change to assignment operator = code in ref<const T> from that in ref <T>. So even if a user defines a variable to be of type ref<const T>, it can still be written to using the assignment operator.

Finally to report a potential data race to the user, we use the __BACKTRACE__ utility provided by libasync. This returns the sequence of base pointers in the program execution stack leading upto the point where the data race was detected. The user can then use gdb to detect at what line and in which source file, the potential data race was detected. We modified the __BACKTRACE__ utility slightly to take care of multithreaded programs.

4.2 Deadlock detection

For deadlock detection, we need to maintain a mutex dependency graph G_M and dynamically update it whenever a thread tries to acquire a lock. To achieve this, we maintain a global mutex dependency graph in the pthread library which is visible to all the threads. When a thread t tries to acquire a lock l, we insert a new vertex corresponding to l (if it doesn't already exist) and add directed edges from l to all the locks currently owned by t. This directed edge represents the partial order $l_1 > l$ for any lock l_1 already owned by t.

As described in Section 3, detecting a potential deadlock is equivalent to finding a cycle in the mutex dependency graph G_M. To find a cycle, we do a *modified DFS* in G_M starting from the vertex corresponding to lock l which the current thread is trying to acquire, after all the updates to G_M (as described above) have been performed. The cycle detection algorithm in directed graphs using *modified DFS* can be found in [3]. The complexity of this algorithm is $O(|V| + |E|)$.

Typically in a multithreaded program, there are locks which are completely independent of each other *i.e.,* they don't have any partial order among themselves. Hence the mutex dependency graph G_M consists of disconnected components and it is inefficient to search for cycles in the whole graph because any new cycle created will certainly pass through the vertex corresponding to the lock currently being acquired. It makes sense to only search for cycles in the connected component containing this vertex. Performing the *DFS* starting from this vertex will take care of this for us.

```
pthread_mutex_lock(pthread_mutex_t *l) {
    if (l is not already in G_M)
        set_insert(V(G_M), l);
    for each lock l' in my_locks
        set_insert(E(G_M), l → l');
    detect_cycle(G_M, l);
    /* Normal mutex locking code */
        ⋮
}
```

Note that all the updates to G_M are done before the actual mutex locking code and hence are irrespective of whether the thread is able to successfully acquire the lock or not. Also note that all vertices and edges remain persistent in the graph throughout the execution of the program. This is because a potential deadlock detected here could result in an actual deadlock under different scheduling decisions.

5 Results

In this section, we present the results of some test cases that we carried out on our data race and deadlock detection tool. We performed exhaustive testing to highlight the correctness and efficiency of the tool that we built.

5.1 Evaluation

We could not test our tool on existing programs because of the restriction that we imposed on the shared variables - that they have to be of type `ref` or `ptr`. Hence we developed our own test cases to check for correctness and soundness.

5.1.1 Data race detection

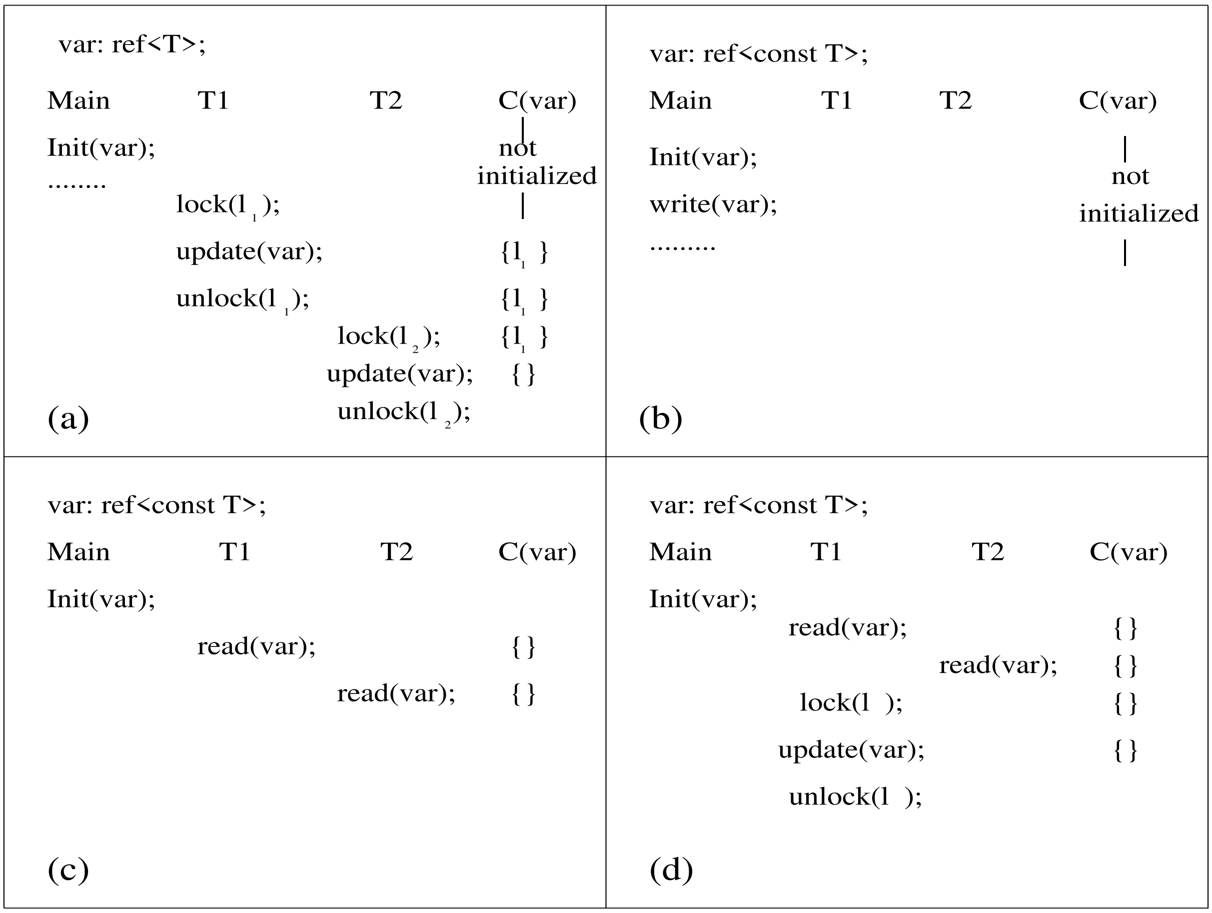

Figure 2: Some data race examples

In Figure 2, we illustrate some special test cases that our data race detection tool is designed to handle. In Figure 2(a), a global variable `var` of type `ref<T>` is first initialized by `main` thread and then thread T_1 updates it holding lock l_1 while thread T_2 updates it holding lock l_2. Note that `var` is not protected by any single lock consistently. While in an ideal situation, the updates by T_1 and T_2 should be done sequentially, the lack of mutex protection on `var` fails to impose such a restriction. Our tool detects this situation and warns the user that there is a potential data race.

Figure 2(b) illustrates the case of “exclusive” access by a single thread. Even though the variable `var` is accessible to all the threads, only the main thread actually accesses it. Here there is no need for any mutex protection on `var` so even though the `main` thread doesn't hold any lock while updating `var`, our tool does not report any data race.

Figure 2(c) depicts the situation wherein the variable `var` is of type `ref<const T>` and operators $\rightarrow, *$ are used only for read access. Initialization is done by `main` thread and all accesses to `var` are “read-only” thereafter. `var` is now in *Shared* state, so even though candidate lock set for `var` is updated at each access, no warning is thrown when the set becomes empty.

Figure 2(d) is similar to 2(c) except that thread T_1 writes to `var` using assignment operator. Then `var` enters *Shared Modified* state and since the lock set is empty at that point, our tool reports to the user that there is a “potential data race”.

5.1.2 Deadlock detection

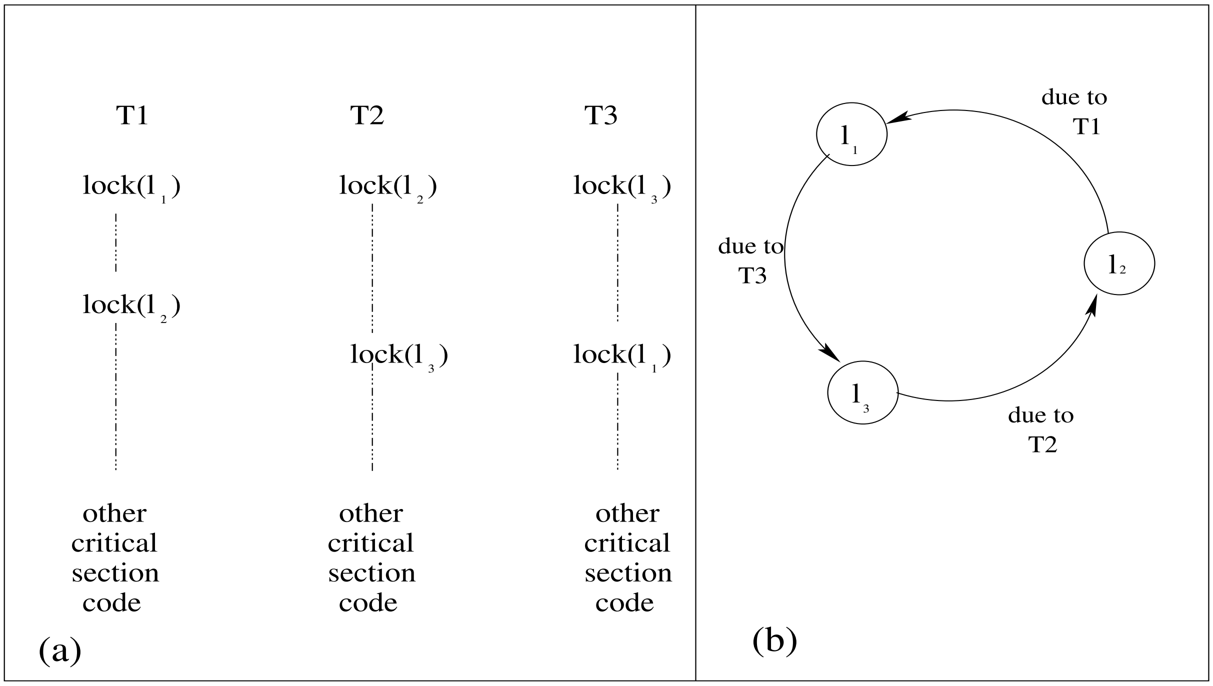

Figure 3: Simple Deadlock situation

Figure 3(a) illustrates 3 threads involved in a deadlock and the corresponding mutex dependency graph is shown in 3(b). Our tool detects the cycle $l_1 \rightarrow l_2 \rightarrow l_3 \rightarrow l_1$ and warns the user of a “potential deadlock” just before the actual deadlock occurs.

5.2 Performance

Our `pthread` and `libasync` libraries perform the extra task of dynamically detecting potential data races and deadlocks. So it is bound to impose some overhead on the multithreaded programs using these libraries. In this section, we aim to illustrate that benefits of these new features outweigh the overhead of using them.

We started by testing our tool on a program which tries to evaluate the worst case performance of both the data race detection as well as the deadlock detection utility. In this test program, there are N mutex locks and one global variable `var`. Initialization of `var` is done by the `main` thread as

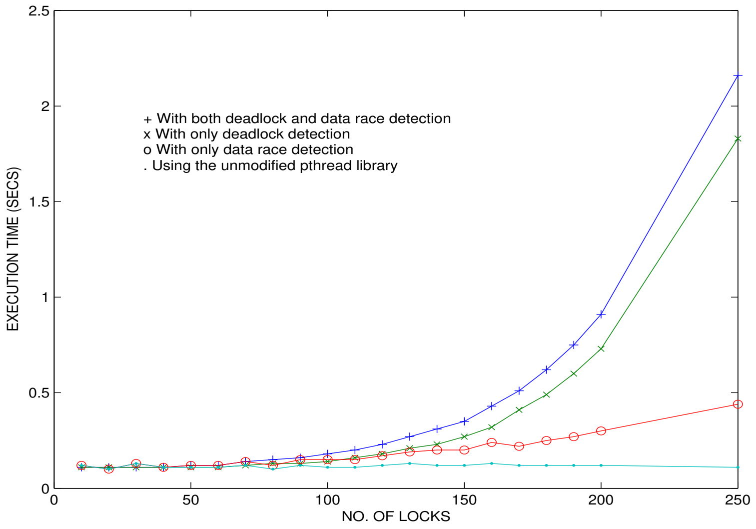

Figure 4: Worst Case Performance

before. Now a thread T acquires all the locks from N to 0, updates **var** N times and then releases all the locks. In the next step it acquires locks $N-10$ to 0 and updates **var** $N-10$ times. It goes on in this fashion, decreasing the number of locks it acquires and the number of updates to **var** by 10 at each step. At the end it accesses **var** without holding any locks.

Lets see why this approximates the worst case overhead of our program. For deadlock detection, there is a graph of N vertices with each vertex having a directed edge to each vertex above it, ie. vertex for the lock 0 has an edge to every vertex 1 to N and so on. Thus the number of edges in the mutex dependency graph is $O(N^2)$. Moreover the graph is fully connected, so that for each of the $N + (N-10) + \ldots + 10$ mutex acquisitions a `DFS` is done on the entire graph. Thus the total overhead due to deadlock detection is almost $O(N^4)$ (since one cycle detection run takes time $O(\mid V \mid + \mid E \mid)$ and there are $O(N^2)$ such runs).

For data race detection again, there will be $O(N^2)$ accesses to the shared variable. So the total overhead over all the lock set intersections will be $O(N^3)$ ($O(N)$ time for set intersections of the lock sets which are of size N, and $O(N^2)$ such intersections). The overhead also includes the overhead for all the set insertions/deletions, which are done each time a thread acquires and releases the lock respectively.

The overhead for this unrealistic case is high as can be seen in the figure 4. The figure also shows the overheads for the deadlock detection and data race detection parts separately. As can be clearly seen, the results for simple data race detection are quite encouraging even for this unrealistic test case. We next show some realistic performance statistics of our tool to show that even the deadlock detection part of our tool is practical for all real purposes.

In the next test case, we slightly modified our mutex dependency graph to induce dense disconnected components in it, instead of having one huge dense graph which is uncommon in actual situations. Here we divided the mutexes into $\frac{N}{50}$chunks of 50 nodes each. Each of the chunks is densely connected with $O(50^2)$ edges. This represents a more realistic case since multithreaded programs usually divide work amongst the threads, with each of the sets of threads performing independent functions, and thus vying for independent sets of mutex locks. The results for this case are depicted in figure 5. As can be seen, the results for this case are quite encouraging.

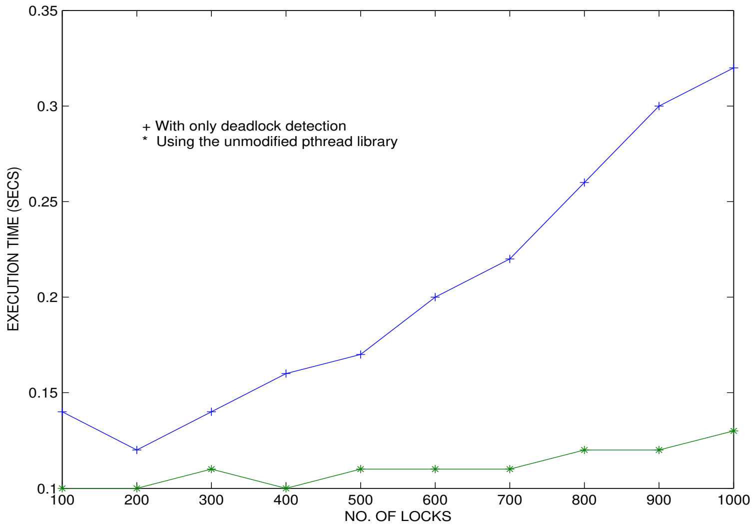

Figure 5: Realistic Deadlock Performance

6 Experiences and Conclusion

We started this project with the aim of improving upon the implementation of the data race detection tool designed by Savage et al [1]. We used the `pthread` and `libasync` for this implementation. This choice is natural since `pthread` is a widely used standard for multithreaded programming, while `libasync` provided us with an interface between shared variable accesses made in a program and the actual physical memory accesses where we could insert the checks of our tool.

Embedding this tool inside the libraries has made it easily usable by any programmer. However, in doing this we weren't able to extend our implementation to "read-write" locks, where multiple threads may be simultaneously reading a shared variable but only a single thread can have write access to it.

The basic problem with making this extension is that we couldn't figure out a way to differentiate a read access from a write access within the `ref/ptr` structure. But we decided to go on to extending our tool for deadlock detection instead of toiling on this problem. This is partly because "read-write" locks are not so common in practice.

Though we were not able to test our tool on existing multithreaded programs because of the restriction on the shared variables, we still believe that our tool will work well for all realistic multithreaded programs that use our libraries. We have tested our tool on all the cases that we designed it to detect and to overlook and are well convinced of its correctness and soundness.

The performance of both the data race and deadlock detection parts is pretty good for realistic cases as described in the previous section. Keeping all this in mind, we feel that our tool is ready for use by real multithreaded programs.

7 Acknowledgment

We did this project as part of the "Honors Operating Systems" course instructed by Prof. David Mazières. We are very grateful to him for helping us out whenever we got stuck during the course of this project and also for promptly responding to the zillion doubt mails we sent him.

References

[1] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, *Eraser: A Dynamic Data Race Detector for multithreaded Programs*, ACM Transactions on Computer Systems, Vol.15, No.4, 1997, pp 391-411.

[2] David Mazières, Revised by Frank Dabek and Eric Petererson, *Using TCP Through Sockets*, Manuscript, http://www.scs.cs.nyu.edu/G22.3250/lab/net2.pdf.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction To Algorithms*, The MIT Press, 2001, ISBN 0-262-03293-7.