A Source-to-Source Transformer for QD-locking

Robert Markovski
Abstract

A Source-to-Source Transformer for QD-locking

Robert Markovski

Locking is a common method of synchronization in parallel programming. Delegation locking is a form of locking wherein threads may, under the right circumstances, delegate critical sections to be executed by another thread instead of waiting for the lock. Queue Delegation Locking (QD-locking) is a novel method of delegation locking which drastically improves performance in situations where one or more locks are moderately or heavily contended, and essentially lets a thread delegate a critical section to the thread which currently holds the lock. QD-locking requires critical sections to be written differently than standard pthreads mutex locking.

In this thesis, we discuss the feasibility of taking a program which uses pthreads mutex locks and automatically transforming it into a program using QD-locking. We describe an implementation of this transformation, called QDTrans, using the qd_lock_lib implementation of QD-locking in the C programming language. We also discuss the quality and performance of code transformed by QDTrans. QDTrans is currently capable of successfully converting seven out of the nine SPLASH2 benchmarks.
1 Introduction

1.1 Overview

Traditionally, the most common way to increase the speed of a given computer was to increase the clock speed [10]. However, a point was reached where it is no longer feasible to do so — the increase in performance no longer outweighed the increased power and cooling requirements. At that point, CPU designers instead began adding multiple cores on CPUs [10], and concurrency and parallelism went from only being relevant when programming big supercomputers to being relevant when programming virtually anything.

However, when programming with parallelism, it is important to deal with shared memory correctly. Multiple threads accessing the same shared memory space can lead to serious problems if the threads are not somehow in agreement about who is allowed to access what memory at what moment in time — multiple threads attempting to modify the same data structure simultaneously can lead to many kinds of data race and race conditions.

Ensuring threads are in agreement over who is allowed to do what during parallel execution is called synchronization. Various methods of synchronization exist, such as semaphores, barriers, and various types of locks [12].

The purpose of this thesis is to present a source-to-source transformation tool called QDTrans. The necessary transformation is outlined, described and discussed. QDTrans and its design are also discussed and evaluated. Finally, QDTrans was tested on the SPLASH2 benchmarks and the results of this are also evaluated and discussed.

1.2 Locks

Mutually exclusive locks, commonly referred to as mutex locks, mutexes (mutex being short for “mutual exclusion (locks)”) or simply locks, are a common subset of methods of synchronization. The basic idea of a lock is as follows: A lock may only be held by a single thread at a time. In order to access resource X, a thread must first acquire the corresponding lock. If another thread is already holding the lock in question, threads who wish to access resource X must wait until the thread that currently holds the lock is done accessing resource X and releases the lock [8].

1.2.1 Contention

Contention occurs when more than one thread want to access the same locked resource at the same time, leaving threads waiting for others to finish. Slight contention is usually a sign that a program is working in parallel as intended, but heavy contention can cause severe performance degradation — threads get

---

1A data race is a type of error which is defined as a situation where two or more threads access the same location simultaneously and at least one of those accesses is a write. This can potentially lead to data corruption and race conditions [11].

2A race condition is a type of error which can arise when a program relies on a particular sequence of operations across multiple threads or CPUs — if this sequence is not enforced properly via some means of synchronization, operations may be carried out in violation of this required sequence, breaking the program, potentially leading to problems such as data corruption, infinite loops, crashes, etc. (Paraphrasing [21]).
stuck for significant periods of time waiting to acquire contented locks when they could be doing useful work \[20\].

1.2.2 Avoiding contention

There are solutions to this problem; for example, it might be possible to rewrite the affected program to avoid contention, for example, by using finer grained locking. However, other solutions also exist, such as various delegation locking algorithms. The idea of delegation locking is to allow critical sections to be delegated to the thread which holds the relevant lock. Delegation locking algorithms include Remote Core Locking \[17\], Flat Combining \[13\], and a novel and interesting solution called Queue Delegation Locking \[14\].

1.3 Queue Delegation Locking

Queue Delegation Locking, or QD-locking for short, is a scalable synchronization mechanism devised by David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. The idea behind QD-locking is to allow critical sections to be delegated to the thread currently holding the lock, which may allow the delegating thread to move on and do other useful work instead of waiting for the lock, as well as enabling cache reuse by keeping consecutive operations on the same shared piece of data on the same core \[16\].

Klaftenegger, Sagonas and Winblad have developed a pair of libraries implementing QD-locking for C and C++. However, for an existing program which uses mutexes to benefit from this library some rewriting is necessary due to the difference in how critical sections need to be written.

1.3.1 My contribution

Rewriting a program to utilize QD-locking properly can be a tedious, manual procedure. However, there is no reason why someone could not write a program which automatically transforms source code to utilize QD-locking rather than conventional mutex locks.

There are essentially two ways to automatically rewrite a program, source-to-binary transformation and source-to-source transformation. Source-to-binary transformation means the program is transformed as part of the compilation process, similar to the various performance optimizations many compilers apply.

This thesis presents QDTrans, a source-to-source transformation tool which automatically rewrites critical sections so use QD-locking. Source-to-source transformation, however, is a transformation technique which consists of parsing valid source code into an abstract syntax tree\(3\) for a particular programming language, possibly performing modifications, and finally outputting valid source code.

\[3\] An Abstract Syntax Tree, or AST, is a tree containing an abstract representation of the structure of programming source code.\(4\)
2 QDTrans

First, a source-to-binary transformation tool would by nature be locked to the compiler infrastructure in which it was implemented, in this case Clang, while a source-to-source transformation tool could in theory be used with any C compiler capable of compiling the transformed program. It was decided that the flexibility of being able to use any compiler would be preferable, so the decision was made to attempt source-to-source transformation. Finally, knowing that the C programming language is quite flexible in what it lets you do, it did not take long to realize that supporting transformation of “every possible valid C program which uses pthreads mutex locking” would have been an enormous undertaking, so a few limitations were defined:

- Only the simple “lock-access-unlock” structure will be fully supported. Supporting multiple reader locks would probably be possible but more complicated locking structures could be complicated to support.
- Due to the fact that mutex locks in C are not explicitly tied to a resource (as in some other languages, e.g. Rust), optimizing critical sections perfectly (e.g. by factoring out pieces of code which do not actually need to reside inside the critical section) would not be feasible since there is really no consistent way to programmatically determine exactly which variable(s) the lock is actually supposed to synchronize more accurately than “one or more of the global variables accessed inside the critical section”.

2.1 How QDTrans works

This section will explain the required changes, and thus the transformation procedure, between mutex locking and QD locking, by means of a couple of examples, one example written from scratch and a slightly more complex example inspired by shared_int_example.c, which is included with the qd_lock_lib library.

The first example is shown in listing 1.
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t lock;
int counter;

int main() {
    counter = 0;
    pthread_mutex_init(&lock, NULL);
    pthread_mutex_lock(&lock);
    printf("Line 1\n");
    counter++;
    printf("Line 3\n");
    pthread_mutex_unlock(&lock);
    pthread_mutex_destroy(&lock);
}

Listing 1: A simple example using Mutex locking

The things that need to happen in order to fully transform this example are:

- A message struct would need to be defined. This struct will contain the values of whatever local variables the critical section accesses. However, this example does not access any local variables and thus does not need a message struct.
- The critical section needs to be factored out into a new delegatable function.
- Any and all unlock statements belonging to the critical section need to be deleted.
- The declaration of the lock needs to have its type changed from pthread_mutex_t to QDLock.
- The calls to pthread_mutex_init() and pthread_mutex_destroy() need to be changed into calls to LL_initialize() and LL_destroy().
- The lock statement needs to be replaced with a call to LL_delegate_wait().

The end result is shown in listing 2.
Listing 2: A simple example using QD locking

The next example is shown in listing 3.
```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    pthread_mutex_t lock;
    int value;
} SharedInt;

sem_t sem;
SharedInt* sip;

int foo = 0;

void *funcWCritSec(int* v2) {
    // Do some work
    pthread_mutex_lock(&(sip->lock));
    sip->value = sip->value + *v2;
    int currvalue = sip->value;
    pthread_mutex_unlock(&(sip->lock));
    printf("Current value was %i.\n", currvalue);
    foo = currvalue;
    // Do some more work
    sem_post(&sem);
}

int main() {
    sem_init(&sem, 0, 0);
    SharedInt si;
    sip = &si;
    sip->value = 0;
    int v2 = 1;
    pthread_mutex_init(&(sip->lock), NULL);
    pthread_t thread1;
    pthread_t thread2;
    pthread_create (&thread1,NULL,funcWCritSec,&v2);
    pthread_create (&thread2,NULL,funcWCritSec,&v2);
    sem_wait(&sem);
    sem_wait(&sem);
    pthread_mutex_destroy(&(sip->lock));
    sem_destroy(&sem);
    printf("%d\n", sip->value); // Should print "2".
    if(sip->value+foo == 3 || sip->value+foo == 4) {
        return 0;
    } else {
        return 1;
    }
}
```

Listing 3: A more complex example using Mutex locking
This example requires a few more operations than the last example:

- A message struct needs to be defined. This struct will contain the values of whatever local variables the critical section accesses (the critical section in the last example does not access any local variables and thus needs no message struct).

- The critical section needs to be factored out into a new delegatable function containing the following:
  1. Code for copying values out of the message struct into local variables.
  2. The entire critical section (without lock and unlock statements).
  3. Code for copying values from variables local to the critical section back to the memory locations of the corresponding variables outside the critical section, which are stored as pointers in the message struct.
  4. A return handler if necessary (necessary to handle return statements inside the critical section).

- Any and all unlock statements belonging to the critical section need to be deleted.

- The declaration of the lock needs to have its type changed from `pthread_mutex_t` to `QDLock`.

- The calls to `pthread_mutex_init()` and `pthread_mutex_destroy()` need to be changed into calls to `LL_initialize()` and `LL_destroy()`.

- The lock statement needs to be replaced with the following:
  1. A declaration of a message struct.
  2. A declaration of the variable `currvalue`, which was originally declared inside the critical section but also used after the critical section. Since the critical section has been factored out, `int currvalue` no longer exists after the critical section and thus needs to be declared before the call to `LL_delegate_wait`.
  3. Code for copying values from local variables into the message struct.
  4. A call to `LL_delegate_wait`.
  5. A return handler if necessary (necessary to handle returning from inside the critical section rather than at the end of it).

The end result is shown in listing 4.
#include "locks/locks.h"

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    QDLock lock;
    int value;
} SharedInt;

sem_t sem;
SharedInt* sip;
int foo = 0;

struct funcWCritSec_critSec0_msg {
    int * v2;
    int * currvalue;
};

void funcWCritSec_critSec0(unsigned int sz, void* msgP) {
    struct funcWCritSec_critSec0_msg* funcWCritSec_cs0msg = (struct
            funcWCritSec_critSec0_msg*)msgP;
    int * v2 = funcWCritSec_cs0msg->v2;
    sip->value = sip->value + *v2;
    int currvalue = sip->value;
    *(funcWCritSec_cs0msg->currvalue) = currvalue;
}

void *funcWCritSec(int* v2) {
    // Do some work
    struct funcWCritSec_critSec0_msg funcWCritSec_cs0msg;
    funcWCritSec_cs0msg.v2 = v2;
    int currvalue;
    funcWCritSec_cs0msg.currvalue = &currvalue;
    LL_delegate_wait(&(sip->lock), funcWCritSec_critSec0,
            sizeof(funcWCritSec_cs0msg), &funcWCritSec_cs0msg);
    printf("Current value was %i.\n", currvalue);
    foo = currvalue;
    // Do some more work
    sem_post(&sem);
}

int main() {
    sem_init(&sem, 0, 0);
    SharedInt si;
    sip = &si;
    sip->value = 0;
    int v2 = 1;
    LL_initialize(&(sip->lock));

Listing 4: A more complex example using QD locking
2.1.1 How QDTrans works

QDTrans is written using the libTooling interface to access the LLVM/Clang framework. QDTrans uses a series of RecursiveASTVisitors to identify critical sections and variables accessed therein, and performs the actual modification by applying Replacements to a Rewriter. We explain these notions below.

A RecursiveASTVisitor is a template from which classes can be derived which recursively visit every node in the AST [2], a Replacement is an object which contains a range of source code and a string with which the range is to be replaced [3], and a Rewriter is an object which gives access to functions which manipulate an underlying source code buffer and which Replacements can be applied to [3].

The Replacements are stored inside a map (std::map) from filenames (std::string) to Replacement vectors (std::vector<Replacement>), and are applied at the very end.

2.1.2 Step by step

This section will show what the AST for one of the two examples above looks like and further explain the inner workings of QDTrans.
Figure 1: The Abstract Syntax Tree for the example shown in listing 1.
Step by step:

1. Each critical section is identified by finding its lock statement (call to `pthread_mutex_lock(lock)`) and iterating through the nodes inside it to find its last unlock statement (call to `pthread_mutex_unlock(lock)` where `lock` is the same lock as in the lock statement).
   - In this case, there is one critical section, its lock and unlock statements have been coloured orange, and its lock has been coloured red in the above graph.

2. Identify any variables which are accessed inside each critical section.
   - In this case, the critical section accesses the variable `counter`.

3. For each critical section, if there are any variables which are accessed inside the critical section and are either local to the function that originally contained the critical section or declared inside the critical section, generate a message struct which will be used to pass these variables into the critical section.
   - In this case, `counter` is a global variable and thus does not need to be passed in, thus no variables which need to be passed in exist, and thus no message struct is generated.

4. Transform and factor out each critical section, removing lock and unlock statements and adding code for copying variables from the message struct before the critical section and code for writing values back after the critical section as necessary.

5. At the original program point of each critical section, add a call to `LL_delegate` or `LL_delegate_wait` as applicable (`LL_delegate` is used iff the critical section contains no variables which need returning, otherwise `LL_delegate_wait` is used in order to wait for the result), preceding it with code for setting up and populating the message struct if one was generated.

6. For each critical section, change the type of the declaration of its lock from `pthread_mutex_t` to `QDLock` and change all calls referring to this lock to `pthread_mutex_init` and `pthread_mutex_destroy` into calls to `LL_initialize` and `LL_destroy`.

The manipulations cannot really be shown individually at the tree level because they are not happening at the tree level; QDTrans never actually modifies the tree but instead uses a Clang class called `Replacement`, which is essentially an interface to automated text editing.

### 2.2 How QDTrans is used

QDTrans can be obtained from its GitHub repository. QDTrans is built by following the build instructions in the file `README.md` in the repository, and is run by typing `./bin/qdtrans [filename]`, where `[filename]` is the name of the file containing the C source code to transform.

---

[QDTrans' GitHub repository is located at](https://github.com/Redhotsmasher/QDTrans)
3 Obstacles

This section will detail several obstacles I had to overcome in making QDTrans work, and will thus also serve as an explanation of many of the inner workings of QDTrans.

3.1 The C implementation

The first attempted implementation was written in C, using libClang, but libClang did not allow modification of the AST, so the proposed solution at the time was to simply recreate the AST in a struct based tree. This coupled with the general verbosity of libClang led to a large codebase which proved to be difficult to understand and debug, so the approach was ultimately abandoned in favour of a new implementation written in C++, against libTooling.

3.2 Types of critical sections

To facilitate the implementation, three types of critical sections were defined:

(A) Critical sections for which the lock and unlock are both located below the same node in the Clang AST.

(B) Critical sections for which the lock and unlock are both contained in the same function but not both located immediately below the same node in the Clang AST.

(C) Critical sections for which the lock and unlock are not both contained in the same function.

Critical sections with several unlock statements are explained in Section 3.3. The transformation works as follows.

3.2.1 Type A

The transformation of type A critical sections is relatively straightforward:

1. Move everything between lock and unlock to a separate function which takes a message struct as input.

2. Add code to the beginning of new function for copying values from the struct to local variables.

3. Replace lock and unlock with code for setting up the message struct and its contents and a call to LL.delegate or LL.delegate_and_wait, followed by code for handling early returns (see Section 3.3) if necessary.

4. Add the declaration of the message struct.

An example of a type A critical section is shown in listing 1.
3.2.2 Type B

Transformation of type B critical sections is not currently implemented. The transformation of type B critical sections would be similar to that of type A sections, although special attention must be paid to recreating control statements when factoring out the critical section, and leaving closing braces intact in some cases.

As an example,

```c
void *functionWithCriticalSection(int* v2) {
  // Do some work
  pthread_mutex_lock(&(sip->lock));
  sip->value = sip->value + *v2;
  if(sip->value = 1) {
    printf("Value is currently: \%i\n", sip->value);
    pthread_mutex_unlock(&(sip->lock));
  } else {
    pthread_mutex_unlock(&(sip->lock));
  }
  // Do some work
  sem_post(&sem);
}
```

Listing 5: Type B example pre transformation

has the unlock statements inside an if else clause, and becomes

```c
void critSec0(unsigned int sz, void* msgP) {
  sip->value = sip->value + *v2;
  if(sip->value = 1) {
    printf("Value is currently: \%i\n", sip->value);
  } else {
  }
}
```

```c
void *functionWithCriticalSection(int* v2) {
  // Do some work
  LL_delegate_wait(&(sip->lock), critSec0, 0, NULL);
  // Do some more work
  sem_post(&sem);
}
```

Listing 6: Type B example post transformation

3.2.3 Type C

Transformation of type C critical sections is also not implemented but would be similar to performing type B transformation after inlining the inner function.
3.3 Return handling

Variables which need to be “returned” (written back) because they are referenced again after a critical section has accessed (and thus potentially modified) them have a corresponding pointer in the message struct, and are copied from this pointer at the beginning of the new function and copied back at the end of it.

Return statements inside critical sections are a different story. First off, the reason this is not as uncommon as it may sound is due to QDTrans’ handling of critical sections with multiple locks: A critical section, as far as QDTrans is concerned, has exactly one lock statement, exactly one “main” unlock statement, which is the one which appears last in the code, and zero or more additional inner return statements. Thus, one may end up with a critical section which, in QDTrans terms, contains one or more unlock statements followed by returns, located before the “main” unlock statement. Consider the following example, shown in listing 7, containing a critical section with a return statement.
Listing 7: An example of a critical section with returns

In QDTrans terms, the critical section spans from the lock statement on line
16 to the unlock statement on line 24, and there is an unlock and a return on lines 20 and 21. A field in the message struct is needed to carry the return value back to the delegating function within which the return statement was originally located and which the return statement is supposed to return from.

However, this is not quite enough, since line 23 modifies the value of \texttt{sip->value}, something which only happens if the return on line 23 \textit{does not} happen. The solution to this is to add another field in the message struct which indicates if we returned before the end of the critical section or not, which is checked immediately after the \texttt{LL_delegate_wait} to know if we should return immediately or not.

The resulting conversion:

```
8  typedef struct {
9     QDLock lock;
10    int value;
11  }SharedInt;
12  sem_t sem;
13  SharedInt* sip;
14
16  struct funcWCritSec_critSec0_msg {
17      int * v2;
18      void * * __retval__;
19      int * __earlyReturn__;
20  };

22  void funcWCritSec_critSec0(unsigned int sz, void* msgP) {
23      struct funcWCritSec_critSec0_msg* funcWCritSec_cs0msg = (struct
24      funcWCritSec_critSec0_msg*)msgP;
25      int * v2 = funcWCritSec_cs0msg->v2;
26      sip->value = sip->value + *v2;
27      if (sip->value < 3) {
28          sip->value = sip->value + *v2;
29          *(funcWCritSec_cs0msg->__retval__) = ((void*)0);
30          *(funcWCritSec_cs0msg->__earlyReturn__) = 1;
31          return;
32      } else {
33          sip->value = sip->value - *v2;
34      }
36  }
37
39  void *funcWCritSec(int* v2) {
40      // Do some work
41      struct funcCritSec_critSec0_msg funcCritSec_cs0msg;
42      funcCritSec_cs0msg.v2 = v2;
43      int __earlyReturn__ = 0;
44      void * __retval__ = NULL;
45      funcCritSec_cs0msg.__earlyReturn__ = &__earlyReturn__;
46      funcCritSec_cs0msg.__retval__ = &__retval__;
47      LL_delegate_wait(&sip->lock), funcCritSec_critSec0,
48      sizeof(funcCritSec_cs0msg), &funcCritSec_cs0msg);
49      if(__earlyReturn__ != 0) {
50          return __retval__;
51      }
52      // Do some more work
53  }

55  void* funcWFunc(int* v2) {

Listing 8: A transformed example of a critical section with returns
As stated, we have a field in the struct `__retval__` for the return value and another field `__earlyReturn__` to indicate whether we have returned early or not. We also have lines 46-48 to return the return value immediately if the critical section returned early, i.e., if the original non-transformed code would have returned before the “main” unlock statement.

4 Related work

There are some existing works similar to QDTrans. Delegation locking is a family of synchronization algorithms where the idea is to increase the scalability of programs with lock contention by allowing critical sections to be delegated to the thread holding the relevant lock. Queue Delegation locking is a form of delegation locking.

4.1 Flat Combining

Flat Combining [13] is a form of delegation locking where a data structure protected by a lock has a publication list to which threads write their access/-modification requests. When a thread needs to access the resource protected by the lock, it will attempt to acquire the lock, and, upon successful acquisition of the lock, become a combiner, scanning the publication list and applying combined requests until the list is empty and releases the lock. If the lock was not acquired, meaning it is currently owned by some other lock, the lock seeking thread will write its requests to the publication list and wait for the lock owning thread to indicate that the request has been fulfilled. The disadvantage compared to QD-locking is that in Flat Combining, a lock seeking thread has to wait until its request is fulfilled, while in QD-locking, a thread may, under the right circumstances, delegate a critical section and continue executing without having to wait for the critical section to finish executing.

4.2 Remote Core Locking

Remote Core Locking [17] (RC-Locking) is another form of delegation locking wherein critical sections are delegated to one or more cores dedicated solely to executing critical sections. The main difference between QD-locking and RC-locking is that, with RC-locking, one or more cores are completely dedicated to executing critical sections, whereas with QD-locking, critical sections are executed by whichever core happens to be holding the relevant lock when the critical section is delegated. Another difference worth mentioning is that, much like Flat Combining, RCL requires the lock seeking thread to wait until its critical section to finish executing, while QD-locking, as previously mentioned, under the right circumstances, allows a thread to delegate a critical section and continue executing without waiting.

4.3 Coccinelle

Coccinelle [18] is a tool which was mainly developed as a tool for performing collateral evolutions in Linux, but can be used to perform many different kinds of transformations of C code. It is essentially an advanced patching tool, which can
patch C code based on various semantic properties of the code. An automatic source-to-source transformation tool for enabling code to utilize RC-Locking was implemented using Coccinelle, but after Coccinelle was looked into, we concluded that it would not be possible to implement QDTrans in Coccinelle because we did not think Coccinelle supported the kind of advanced function manipulation QDTrans requires and thus Coccinelle would not be a suitable way forward.

5 Results

QDTrans is a source-to-source transformation tool developed with the goal of being able to fully automatically transform a C program using conventional mutex locks into a C program using QD-locking (qd_lock_lib). This is what has been accomplished thus far:

5.1 Implementation status

The implementation is currently unfinished — it will currently find critical sections as long as the lock and unlock statements are inside the same function (and possibly sometimes even if they are not) and transform type A critical sections only.

5.2 Known limitations

5.2.1 QDTrans currently only processes one file at a time

QDTrans currently only processes a single file at a time, which can be problematic in certain cases. An example is shown in listings 9 and 10.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    pthread_mutex_t lock;
    int value;
}SharedInt;
```

Listing 9: common.c

#include "common.c"

sem_t sem;
SharedInt* sip;

void *functionWithCriticalSection(int* v2) {
    // Do some work
    pthread_mutex_lock(&(sip->lock));
    sip->value = sip->value + *v2;
    pthread_mutex_unlock(&(sip->lock));
    // Do some more work
    sem_post(&sem);
}

int main() {
    sem_init(&sem, 0, 0);
    SharedInt si;
    sip = &si;
    sip->value = 0;
    int v2 = 1;
    pthread_mutex_init(&(sip->lock), NULL);
    pthread_t thread1;
    pthread_t thread2;
    pthread_create (&thread1, NULL, functionWithCriticalSection, &v2);
    pthread_create (&thread2, NULL, functionWithCriticalSection, &v2);
    sem_wait(&sem);
    sem_wait(&sem);
    pthread_mutex_destroy(&(sip->lock));
    sem_destroy(&sem);
    printf("%dn, sip->value); // Should print 2.
    return 0;
}

Listing 10: main.c

When QDTrans is invoked on common.c it will not detect any critical sections and thus return the original code verbatim, and when it is invoked on main.c, it will detect the critical section but fail to find the lock itself and thus again returns the original code verbatim.

A possible solution could be to simply assume in the first case that critical sections exist elsewhere and will be QD-transformed and to assume in the second case that the lock itself exists elsewhere and will be converted into a QD lock, but this effectively makes the assumption that the user actually runs QDTrans on all relevant files. One possible solution could be to print a warning of the form Warning: lock and unlock calls found but no definition for lock 'lock' was found!, but this is currently not implemented.

5.2.2 QDTrans does not support all possible kinds of type A critical section

While QDTrans is able to handle critical sections with inner unlocks followed by returns (see section 3.3), it is currently only able to handle such critical sections which do not have any code between the inner unlock and the following return.
A program containing such a critical section with code between the inner unlock and the following return is shown in listing 11.

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    pthread_mutex_t lock;
    int value;
} SharedInt;

sem_t sem;
SharedInt* sip;

void *funcWCritSec(int* v2) {
    // Do some work
    pthread_mutex_lock(&(sip->lock));
    sip->value = sip->value + *v2;
    if(sip->value < 3) {
        sip->value = sip->value + *v2;
        pthread_mutex_unlock(&(sip->lock));
        sem_post(&sem);
        return NULL;
    }
    sip->value = sip->value - *v2;
    pthread_mutex_unlock(&(sip->lock));
    // Do some more work
    sem_post(&sem);
}

int main() {
    sem_init(&sem, 0, 0);
    SharedInt si;
    sip = &si;
    sip->value = 0;
    int v2 = 1;
    pthread_mutex_init(&(sip->lock), NULL);
    pthread_t thread1;
    pthread_t thread2;
    pthread_create (&thread1,NULL,funcWCritSec,&v2);
    pthread_create (&thread2,NULL,funcWCritSec,&v2);
    sem_wait(&sem);
    sem_wait(&sem);
    pthread_mutex_destroy(&(sip->lock));
    sem_destroy(&sem);
    printf("%d\n", sip->value); // Should print "2".
    return sip->value-2;
}
```

Listing 11: A program with a critical section QDTrans will not transform correctly

QDTrans attempt at transforming this program is shown in listing 12.
Listing 12: An incorrectly transformed program with a critical section with returns

QDTrans is not aware of this extra line of code, which is on line 21 in listing 11 and line 28 in listing 12, so the line is simply left inside the critical section.
The correct transformation would be to move this line so that it sits between lines 46 and 47 in listing 12. The correct transformation of the relevant part of the program is shown in listing 13.

```c
void funcWCritSec_critSec0(unsigned int sz, void* msgP) {
    struct funcWCritSec_critSec0_msg* funcWCritSec_cs0msg = (struct
    funcWCritSec_critSec0_msg*)msgP;
    int * v2 = funcWCritSec_cs0msg->v2;
    sip->value = sip->value + *v2;
    if (sip->value < 3) {
        sip->value = sip->value + *v2;
        sem_post(&sem);
        *(funcWCritSec_cs0msg->__retval__) = ((void *)0);
        *(funcWCritSec_cs0msg->__earlyReturn__) = 1;
        return;
    }
    sip->value = sip->value - *v2;
}
```

```c
void *funcWCritSec(int* v2) {
    // Do some work
    struct funcWCritSec_critSec0_msg funcWCritSec_cs0msg;
    funcWCritSec_cs0msg.v2 = v2;
    int __earlyReturn__ = 0;
    void * __retval__ = NULL;
    funcWCritSec_cs0msg.__earlyReturn__ = &__earlyReturn__;
    funcWCritSec_cs0msg.__retval__ = &__retval__;
    LL_delegate_wait(&(sip->lock), funcWCritSec_critSec0,
        sizeof(funcWCritSec_cs0msg), &funcWCritSec_cs0msg);
    if(__earlyReturn__ != 0) {
        return __retval__;
    }
    // Do some more work
    sem_post(&sem);
}
```

Listing 13: A correctly (manually corrected) transformed version of the same program

5.2.3 Other known limitations

- QDTrans does not currently perform any analysis of execution paths.
- QDTrans therefore currently only performs very limited optimization of the end result.
5.3 Test results

The Splash2 benchmarks were downloaded and a patch for modern computers from UDEL was applied. QDTrans was successfully able to convert seven of the nine SPLASH2 benchmarks. QDTrans was able to transform nearly all critical sections.

5.3.1 Setup

The benchmarks were transformed using a modified makefile and modified macro files. A patch file containing these changes is available here. You will need to supply your own qd_lock_lib headers and static library file. Seven out of nine SPLASH2 benchmarks were successfully transformed, compiled and run, both non-transformed (using the c.m4.null.POSIX_BARRIER_RFIX macros) and transformed (using the c.m4.null.POSIX_BARRIER_RFIX_QD macros).

5.3.2 Transformation rate

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Critical sections</th>
<th>Transformation rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type A</td>
<td>Type B</td>
</tr>
<tr>
<td>ocean/contiguous_partitions</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>ocean/non_contiguous_partitions</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>radiosity</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>raytrace</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>water-nsqared</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>water-spatial</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>volrend</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>76</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1: Transformation rates

---

5This patch file technically contains modified parts of SPLASH2 files. To avoid any potential legal issues in the future (the SPLASH2 license is not entirely clear on the matter), if you are one of the authors of SPLASH2 and take issue with this, contact me via email and I will take the file down if requested.
On average, QDTrans was able to transform approximately 98.6% of all critical sections and 100% of all type A critical sections.

5.3.3 Performance

All benchmarks were run, both non-transformed and transformed, on a machine with a Haswell Intel i7 4770K CPU, running Ubuntu 16.04 LTS, after a fresh boot with no other programs running. The benchmarks were automatically run 5 times each, under 3 different threading configurations, for a total of 15 times. They were run on 1 thread, to get an idea of how much overhead QDTrans adds, 4 threads, to gauge performance when using all available physical cores, and 8 threads, to gauge performance when using all available hardware threads. All numbers were measured by the benchmarks themselves except the volrend numbers, which was measured by the shell script running the benchmarks because that benchmark does not print any timing information.

\footnote{Excludes the first timestep of the simulation.}
### Table 2: ocean/contiguous_partitions

<table>
<thead>
<tr>
<th></th>
<th>With initialization</th>
<th>Without initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mutex locking</td>
<td>QD locking</td>
</tr>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>932259 µs</td>
<td>817823 µs</td>
</tr>
<tr>
<td></td>
<td>811970 µs</td>
<td>814044 µs</td>
</tr>
<tr>
<td></td>
<td>800895 µs</td>
<td>814872 µs</td>
</tr>
<tr>
<td></td>
<td>801914 µs</td>
<td>820820 µs</td>
</tr>
<tr>
<td></td>
<td>812894 µs</td>
<td>821756 µs</td>
</tr>
<tr>
<td>Average</td>
<td>831986.4 µs</td>
<td>817863 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-1.70%</td>
<td>+0.78%</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>540352 µs</td>
<td>491067 µs</td>
</tr>
<tr>
<td></td>
<td>486601 µs</td>
<td>544220 µs</td>
</tr>
<tr>
<td></td>
<td>513940 µs</td>
<td>508720 µs</td>
</tr>
<tr>
<td></td>
<td>535845 µs</td>
<td>537862 µs</td>
</tr>
<tr>
<td></td>
<td>513402 µs</td>
<td>523951.2 µs</td>
</tr>
<tr>
<td>Average</td>
<td>518028 µs</td>
<td>523951.2 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+1.14%</td>
<td>+0.98%</td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>482960 µs</td>
<td>475069 µs</td>
</tr>
<tr>
<td></td>
<td>491996 µs</td>
<td>481598 µs</td>
</tr>
<tr>
<td></td>
<td>481137 µs</td>
<td>475365 µs</td>
</tr>
<tr>
<td></td>
<td>484845 µs</td>
<td>494735 µs</td>
</tr>
<tr>
<td></td>
<td>478743 µs</td>
<td>487059 µs</td>
</tr>
<tr>
<td>Average</td>
<td>483936.2 µs</td>
<td>482765.2 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-0.24%</td>
<td>-0.01%</td>
</tr>
</tbody>
</table>

### Table 3: ocean/non_contiguous_partitions

<table>
<thead>
<tr>
<th></th>
<th>With initialization</th>
<th>Without initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mutex locking</td>
<td>QD locking</td>
</tr>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>56845 µs</td>
<td>56731 µs</td>
</tr>
<tr>
<td></td>
<td>59520 µs</td>
<td>60619 µs</td>
</tr>
<tr>
<td></td>
<td>57576 µs</td>
<td>56401 µs</td>
</tr>
<tr>
<td></td>
<td>60629 µs</td>
<td>59858 µs</td>
</tr>
<tr>
<td></td>
<td>56551 µs</td>
<td>55989 µs</td>
</tr>
<tr>
<td>Average</td>
<td>58224.2 µs</td>
<td>57919.6 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-0.52%</td>
<td>+0.21%</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>23784 µs</td>
<td>23518 µs</td>
</tr>
<tr>
<td></td>
<td>23442 µs</td>
<td>23361 µs</td>
</tr>
<tr>
<td></td>
<td>32184 µs</td>
<td>23800 µs</td>
</tr>
<tr>
<td></td>
<td>23890 µs</td>
<td>24265 µs</td>
</tr>
<tr>
<td></td>
<td>33638 µs</td>
<td>33462 µs</td>
</tr>
<tr>
<td>Average</td>
<td>27447.6 µs</td>
<td>25681.2 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-6.44%</td>
<td>-6.85%</td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>23734 µs</td>
<td>25243 µs</td>
</tr>
<tr>
<td></td>
<td>24535 µs</td>
<td>23825 µs</td>
</tr>
<tr>
<td></td>
<td>25127 µs</td>
<td>24555 µs</td>
</tr>
<tr>
<td></td>
<td>23853 µs</td>
<td>27919 µs</td>
</tr>
<tr>
<td></td>
<td>25071 µs</td>
<td>23780 µs</td>
</tr>
<tr>
<td>Average</td>
<td>24464 µs</td>
<td>25064.4 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+2.45%</td>
<td>+1.39%</td>
</tr>
</tbody>
</table>
The ocean benchmarks, shown in tables 2 and 3, only show tiny differences between pthreads mutex locking and QD-locking, which may be due to poor threading; we are only just about seeing a doubling of performance going from 1 thread to 8, QD locking or not. The 4 thread runs of non_contiguous_partitions appear to show significant improvement until we consider the variance between runs as well as the 8 thread runs which are inconsistent with this, telling us that this apparent improvement is insignificant. The main reason for the increased variance in non_contiguous_partitions compared to contiguous_partitions is that contiguous_partitions was run with the -n1026 parameter, increasing the grid size of the simulation from 258*258 to 1026*1026, making each run longer and in effect lowering the impact of background processes, giving more stable numbers. This was not done with non_contiguous_partitions because that benchmark does not support grid sizes above 258*258 without non-trivial modifications which were not done.

The radiosity benchmark turns out to have broken threading as it runs on one single CPU core regardless of what -p parameter is given for both Mutex and QD versions, thus no table was generated. It turns out that this is due to a known race condition [19]. The numbers indicated no significant difference in single core performance.

```
./RAYTRACE -m64 -a16 -p8 ./inputs/car.env
```

<table>
<thead>
<tr>
<th></th>
<th>With initialization</th>
<th>Without initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mutex locking</td>
<td>QD locking</td>
</tr>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>1174977 µs</td>
<td>1199371 µs</td>
</tr>
<tr>
<td></td>
<td>1166883 µs</td>
<td>1188579 µs</td>
</tr>
<tr>
<td></td>
<td>1167289 µs</td>
<td>1193945 µs</td>
</tr>
<tr>
<td></td>
<td>1170403 µs</td>
<td>1184223 µs</td>
</tr>
<tr>
<td></td>
<td>1173775 µs</td>
<td>1194041 µs</td>
</tr>
<tr>
<td>Average</td>
<td>1170649.4 µs</td>
<td>1192031.8 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+1.83%</td>
<td>+1.83%</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>355833 µs</td>
<td>330270 µs</td>
</tr>
<tr>
<td></td>
<td>383864 µs</td>
<td>334734 µs</td>
</tr>
<tr>
<td></td>
<td>410143 µs</td>
<td>331251 µs</td>
</tr>
<tr>
<td></td>
<td>385050 µs</td>
<td>332290 µs</td>
</tr>
<tr>
<td></td>
<td>356150 µs</td>
<td>331309 µs</td>
</tr>
<tr>
<td>Average</td>
<td>379108 µs</td>
<td>331970.8 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+12.43%</td>
<td>-12.44%</td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>295935 µs</td>
<td>272410 µs</td>
</tr>
<tr>
<td></td>
<td>295006 µs</td>
<td>269742 µs</td>
</tr>
<tr>
<td></td>
<td>294223 µs</td>
<td>269152 µs</td>
</tr>
<tr>
<td></td>
<td>293887 µs</td>
<td>272450 µs</td>
</tr>
<tr>
<td></td>
<td>295488 µs</td>
<td>275416 µs</td>
</tr>
<tr>
<td>Average</td>
<td>294907.8 µs</td>
<td>271834 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-1.82%</td>
<td>-4.32%</td>
</tr>
</tbody>
</table>

Table 4: raytrace

The raytrace benchmark, shown in table 4, shows significant performance improvements for both multithreaded configurations with insignificant additional overhead on the single threaded runs. I would speculate that a big part of the
reason for the improvement is that this benchmark appears to benefit well from threading: the performance nearly quadruples going from 1 thread to 4. The below graph illustrates the difference in runtime between Mutex locking and QD locking.

Raytrace runtime difference between Mutex locking and QD locking

For water-nsquared and water-spatial, shown in tables 5 and 6, the threading appears to be good (3.5 to 4 times the performance going from 1 thread to 4) but there are no consistent performance differences, only random fluctuations due to background processes. This is probably due to the very short length (on modern systems) of these benchmarks.
### Table 5: water-nsquared

<table>
<thead>
<tr>
<th></th>
<th>Mutex locking</th>
<th>QD locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>61303 µs</td>
<td>63496 µs</td>
</tr>
<tr>
<td></td>
<td>61050 µs</td>
<td>63342 µs</td>
</tr>
<tr>
<td></td>
<td>61027 µs</td>
<td>63251 µs</td>
</tr>
<tr>
<td></td>
<td>61342 µs</td>
<td>62613 µs</td>
</tr>
<tr>
<td></td>
<td>62932 µs</td>
<td>63243 µs</td>
</tr>
<tr>
<td>Average</td>
<td>61530.8 µs</td>
<td>63189 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+2.70%</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>25985 µs</td>
<td>18815 µs</td>
</tr>
<tr>
<td></td>
<td>18670 µs</td>
<td>18786 µs</td>
</tr>
<tr>
<td></td>
<td>18529 µs</td>
<td>18940 µs</td>
</tr>
<tr>
<td></td>
<td>18563 µs</td>
<td>18836 µs</td>
</tr>
<tr>
<td></td>
<td>18746 µs</td>
<td>18812 µs</td>
</tr>
<tr>
<td>Average</td>
<td>20098.6 µs</td>
<td>18837.8 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-6.27%</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>16172 µs</td>
<td>15896 µs</td>
</tr>
<tr>
<td></td>
<td>15158 µs</td>
<td>20504 µs</td>
</tr>
<tr>
<td></td>
<td>14040 µs</td>
<td>15772 µs</td>
</tr>
<tr>
<td></td>
<td>13777 µs</td>
<td>14121 µs</td>
</tr>
<tr>
<td></td>
<td>15098 µs</td>
<td>14425 µs</td>
</tr>
<tr>
<td>Average</td>
<td>14849 µs</td>
<td>16143.6 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+8.72%</td>
<td></td>
</tr>
</tbody>
</table>

### Table 6: water-spatial

<table>
<thead>
<tr>
<th></th>
<th>Mutex locking</th>
<th>QD locking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>87129 µs</td>
<td>87269 µs</td>
</tr>
<tr>
<td></td>
<td>87314 µs</td>
<td>86933 µs</td>
</tr>
<tr>
<td></td>
<td>91926 µs</td>
<td>89237 µs</td>
</tr>
<tr>
<td></td>
<td>92352 µs</td>
<td>89289 µs</td>
</tr>
<tr>
<td></td>
<td>87213 µs</td>
<td>88165 µs</td>
</tr>
<tr>
<td>Average</td>
<td>89186.8 µs</td>
<td>88178.6 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-1.13%</td>
<td></td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>23410 µs</td>
<td>30765 µs</td>
</tr>
<tr>
<td></td>
<td>24314 µs</td>
<td>23189 µs</td>
</tr>
<tr>
<td></td>
<td>23440 µs</td>
<td>23077 µs</td>
</tr>
<tr>
<td></td>
<td>30667 µs</td>
<td>23863 µs</td>
</tr>
<tr>
<td></td>
<td>23704 µs</td>
<td>23059 µs</td>
</tr>
<tr>
<td>Average</td>
<td>25107 µs</td>
<td>24790.6 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-1.26%</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>20418 µs</td>
<td>15578 µs</td>
</tr>
<tr>
<td></td>
<td>15452 µs</td>
<td>15673 µs</td>
</tr>
<tr>
<td></td>
<td>15417 µs</td>
<td>16659 µs</td>
</tr>
<tr>
<td></td>
<td>17895 µs</td>
<td>15614 µs</td>
</tr>
<tr>
<td></td>
<td>18235 µs</td>
<td>18252 µs</td>
</tr>
<tr>
<td>Average</td>
<td>17483.4 µs</td>
<td>16355.2 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-6.45%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mutex locking</td>
<td>QD locking</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>1 thread</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>383022 µs</td>
<td>382353 µs</td>
</tr>
<tr>
<td></td>
<td>381808 µs</td>
<td>383624 µs</td>
</tr>
<tr>
<td></td>
<td>382876 µs</td>
<td>380247 µs</td>
</tr>
<tr>
<td></td>
<td>381667 µs</td>
<td>393252 µs</td>
</tr>
<tr>
<td></td>
<td>383257 µs</td>
<td>384513 µs</td>
</tr>
<tr>
<td>Average</td>
<td>382526 µs</td>
<td>384797.8 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>+0.59%</td>
<td></td>
</tr>
<tr>
<td><strong>4 threads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>249504 µs</td>
<td>237734 µs</td>
</tr>
<tr>
<td></td>
<td>238627 µs</td>
<td>248889 µs</td>
</tr>
<tr>
<td></td>
<td>236519 µs</td>
<td>238342 µs</td>
</tr>
<tr>
<td></td>
<td>250272 µs</td>
<td>237165 µs</td>
</tr>
<tr>
<td></td>
<td>273073 µs</td>
<td>259135 µs</td>
</tr>
<tr>
<td>Average</td>
<td>249599 µs</td>
<td>244253 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-2.14%</td>
<td></td>
</tr>
<tr>
<td><strong>8 threads</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All runs</td>
<td>248082 µs</td>
<td>234396 µs</td>
</tr>
<tr>
<td></td>
<td>236983 µs</td>
<td>228606 µs</td>
</tr>
<tr>
<td></td>
<td>256211 µs</td>
<td>225960 µs</td>
</tr>
<tr>
<td></td>
<td>232419 µs</td>
<td>245173 µs</td>
</tr>
<tr>
<td></td>
<td>226024 µs</td>
<td>225341 µs</td>
</tr>
<tr>
<td>Average</td>
<td>239983.8 µs</td>
<td>231895.2 µs</td>
</tr>
<tr>
<td>Difference</td>
<td>-3.37%</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: volrend

Again, very minor differences between mutex locking and QD-locking for volrend, shown in table 7. Threading seems to be quite poor for this benchmark, not even doubling performance going from a single thread to 4 or 8 threads. As mentioned previously, this benchmark does not print any timing information and was therefore timed by the shell script which ran the benchmarks.

In conclusion, given that only a single benchmark, raytrace, showed significant improvement, it is hard to draw any solid conclusions about what performance benefits QDTrans may be able to provide.
6 Conclusion

I have written a (reasonably) well thought out implementation which works reasonably well and produces code which performs reasonably well. I have effectively demonstrated the feasibility of automated source code transformation from conventional mutex locking to QD-locking.

It is possible that source-to-binary (or possibly even binary-to-binary) transformation might be a better idea in the long run, especially given how particular the libTooling Clang parser (and by extension QDTrans) can be at times.

Another possible solution may be to use the GCC plugin API to do source-to-binary (or possibly even source-to-source transformation somehow) transformation, which appears less messy than the libTooling API by reading the documentation [5], although that API is also unstable across releases, like the libTooling API, and I have no actual experience with it outside of skimming through the documentation, so while I do not know, it is certainly possible that the GCC plugin API may not be as friendly as it first seems.

I believe QDTrans is a legitimately useful tool for automatically converting programs which use conventional mutex locks into programs which use QD-locking with some manual work, which could in the future be improved to the point of being able to fully automatically transform the vast majority of programs.

6.1 Future work

The implementation could be given support for “delegate and copy” [22]. It could also, of course, be given a nice GUI and some settings to control exactly how it balances attempting to optimize for ideal performance against attempting not to break any functionality.

Better optimization would be nice, for example analyzing execution paths (probably using the Clang Static Analyzer [1] as a library), which would allow for more sophisticated optimizations (based on things like being able to know whether a variable is accessed again or not in every case, as QDTrans currently only scans the function containing the critical section it can only know if a variable is accessed within this function after the critical section as opposed to knowing whether it is ever accessed again in the entire program), which would improve the performance of the resulting code. QDTrans currently errrs on the side of caution, which means that QDTrans will sometimes make suboptimal decisions in trying to avoid the possibility of outputting broken code.

Furthermore, QDTrans could also be rewritten to consider all files involved in a project simultaneously (perhaps by following includes). This would make QDTrans able to handle programs which have locks which are included from one file to another (see section 5.2.1 of this document).
References


