UPPSALA
UNIVERSITET

17072
Examensarbete 15 hp
Oktober 2017

A Source-to-Source Transformer for QD-locking

Robert Markovski

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

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 contented, 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 phreads 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 tranformed by QDTrans. QDTrans is currently capable of successfully converting seven out of the nine SPLASH2 benchmarks.

Handledare: Kjell Winblad
Ämnesgranskare: Konstantinos Sagonas
Examinator: Olle Gällmo
17072
Tryckt av: Reprocentralen ITC

Contents

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 races[1] and race conditions[2].

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

Mutally 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

[1]A 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].

[2]A 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 [9]

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.

```c
#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.

```c
#include "locks/locks.h"

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

QDLock lock;
int counter;

void main_critSec0(unsigned int sz, void* msgP) {
    printf("Line␣1\n");
    counter++;
    printf("Line␣3\n");
}

int main() {

    counter = 0;
    LL_initialize(&lock);
    LL_delegate(&lock, main_critSec0, 0, NULL);

    pthread_mutex_destroy(&lock);

}
```

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.

```c
#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 `RecursiveASTVisitor`s to identify critical sections and variables accessed therein, and performs the actual modification by applying `Replacement`s 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 [4], and a `Rewriter` is an object which gives access to functions which manipulate an underlying source code buffer and which `Replacement`s can be applied to [3].

The `Replacement`s 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.

```
TranslationUnitDecl 0x9e19270 <<invalid sloc>> <invalid sloc>
├─VarDecl 0x9ed9890 </home/redhotsmasher/qdstuff/newmutexes.noqd.c:5:1, col:17> col:17 used lock 'pthread_mutex_t':'pthread_mutex_t'
├─VarDecl 0x9ed9908 <line:6:1, col:5> col:5 used counter 'int'
└─FunctionDecl 0x9ed99c0 <line:8:1, line:19:1> line:8:5 main 'int ()'
  └─CompoundStmt 0x9eda118 <col:12, line:19:1>
    ├─BinaryOperator 0x9ed9aa0 <line:10:5, col:15> 'int' '='
    │ ├─DeclRefExpr 0x9ed9a58 <col:5> 'int' lvalue Var 0x9ed9908 'counter' 'int'
    │ └─IntegerLiteral 0x9ed9a80 <col:15> 'int' 0
    ├─CallExpr 0x9ed9c30 <line:11:5, col:35> 'int'
    │ ├─ImplicitCastExpr 0x9ed9c18 <col:5> 'int (*)(pthread_mutex_t *, const pthread_mutexattr_t *)' <FunctionToPointerDecay>
    │ │ └─DeclRefExpr 0x9ed9ac8 <col:5> 'int (pthread_mutex_t *, const pthread_mutexattr_t *)' Function 0x9f35510 'pthread_mutex_init' 'int (pthread_mutex_t *, const pthread_mutexattr_t *)'
    │ ├─UnaryOperator 0x9ed9b18 <col:24, col:25> 'pthread_mutex_t *' prefix '&'
    │ │ └─DeclRefExpr 0x9ed9af0 <col:25> 'pthread_mutex_t':'pthread_mutex_t' lvalue Var 0x9ed9890 'lock' 'pthread_mutex_t':'pthread_mutex_t'
    │ └─ImplicitCastExpr 0x9ed9c68 </usr/local/bin/../lib/clang/5.0.0/include/stddef.h:105:16, col:25> 'const pthread_mutexattr_t *' <NullToPointer>
    │   └─ParenExpr 0x9ed9b98 <col:16, col:25> 'void *'
    │     └─CStyleCastExpr 0x9ed9b70 <col:17, col:24> 'void *' <NullToPointer>
    │       └─IntegerLiteral 0x9ed9b38 <col:24> 'int' 0
    ├─CallExpr 0x9ed9d60 </home/redhotsmasher/qdstuff/newmutexes.noqd.c:12:5, col:29> 'int'
    │ ├─ImplicitCastExpr 0x9ed9d48 <col:5> 'int (*)(pthread_mutex_t *)' <FunctionToPointerDecay>
    │ │ └─DeclRefExpr 0x9ed9c80 <col:5> 'int (pthread_mutex_t *)' Function 0x9f35af0 'pthread_mutex_lock' 'int (pthread_mutex_t *)'
    │ └─UnaryOperator 0x9ed9cd0 <col:24, col:25> 'pthread_mutex_t *' prefix '&'
    │   └─DeclRefExpr 0x9ed9ca8 <col:25> 'pthread_mutex_t':'pthread_mutex_t' lvalue Var 0x9ed9890 'lock' 'pthread_mutex_t':'pthread_mutex_t'
    ├─CallExpr 0x9ed9e30 <line:13:5, col:22> 'int'
    │ ├─ImplicitCastExpr 0x9ed9e18 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    │ │ └─DeclRefExpr 0x9ed9d90 <col:5> 'int (const char *, ...)' Function 0x9e7d960 'printf' 'int (const char *, ...)'
    │ └─ImplicitCastExpr 0x9ed9e78 <col:12> 'const char *' <BitCast>
    │   └─ImplicitCastExpr 0x9ed9e60 <col:12> 'char *' <ArrayToPointerDecay>
    │     └─StringLiteral 0x9ed9db8 <col:12> 'char [8]' lvalue "Line 1\n"
    ├─UnaryOperator 0x9ed9eb8 <line:14:5, col:12> 'int' postfix '++'
    │ └─DeclRefExpr 0x9ed9e90 <col:5> 'int' lvalue Var 0x9ed9908 'counter' 'int'
    ├─CallExpr 0x9ed9f48 <line:15:5, col:22> 'int'
    │ ├─ImplicitCastExpr 0x9ed9f30 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    │ │ └─DeclRefExpr 0x9ed9ed8 <col:5> 'int (const char *, ...)' Function 0x9e7d960 'printf' 'int (const char *, ...)'
    │ └─ImplicitCastExpr 0x9ed9f90 <col:12> 'const char *' <BitCast>
    │   └─ImplicitCastExpr 0x9ed9f78 <col:12> 'char *' <ArrayToPointerDecay>
    │     └─StringLiteral 0x9ed9f00 <col:12> 'char [8]' lvalue "Line 3\n"
    ├─CallExpr 0x9eda030 <line:16:5, col:31> 'int'
    │ ├─ImplicitCastExpr 0x9eda018 <col:5> 'int (*)(pthread_mutex_t *)' <FunctionToPointerDecay>
    │ │ └─DeclRefExpr 0x9ed9fa8 <col:5> 'int (pthread_mutex_t *)' Function 0x9f35fd0 'pthread_mutex_unlock' 'int (pthread_mutex_t *)'
    │ └─UnaryOperator 0x9ed9ff8 <col:26, col:27> 'pthread_mutex_t *' prefix '&'
    │   └─DeclRefExpr 0x9ed9fd0 <col:27> 'pthread_mutex_t':'pthread_mutex_t' lvalue Var 0x9ed9890 'lock' 'pthread_mutex_t':'pthread_mutex_t'
    └─CallExpr 0x9eda0e8 <line:17:5, col:32> 'int'
      ├─ImplicitCastExpr 0x9eda0d0 <col:5> 'int (*)(pthread_mutex_t *)' <FunctionToPointerDecay>
      │ └─DeclRefExpr 0x9eda060 <col:5> 'int (pthread_mutex_t *)' Function 0x9f35758 'pthread_mutex_destroy' 'int (pthread_mutex_t *)'
      └─UnaryOperator 0x9eda0b0 <col:27, col:28> 'pthread_mutex_t *' prefix '&'
        └─DeclRefExpr 0x9eda088 <col:28> 'pthread_mutex_t':'pthread_mutex_t' lvalue Var 0x9ed9890 'lock' 'pthread_mutex_t':'pthread_mutex_t'
```

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[4].
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.

[4]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,

```
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 more work
    sem_post(&sem);
}
```

Listing 5: Type B example pre transformation

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

```
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 {
    }
}

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.

```
 1 #include <pthread.h>
 2 #include <semaphore.h>
 3 #include <stdio.h>
 4 #include <stdlib.h>

 6 typedef struct{
 7     pthread_mutex_t lock;
 8     int value;
 9 }SharedInt;

11 sem_t sem;
12 SharedInt* sip;

14 void *funcWCritSec(int* v2) {
15     // Do some work
16     pthread_mutex_lock(&(sip->lock));
17     sip->value = sip->value + *v2;
18     if(sip->value < 3) {
19         sip->value = sip->value + *v2;
20         pthread_mutex_unlock(&(sip->lock));
21         return NULL;
22     }
23     sip->value = sip->value - *v2;
24     pthread_mutex_unlock(&(sip->lock));
25     // Do some more work
26 }

28 void* funcWFunc(int* v2) {
29     funcWCritSec(v2);
30     sem_post(&sem);
31 }

33 int main() {
34     sem_init(&sem, 0, 0);
35     SharedInt si;
36     sip = &si;
37     sip->value = 0;
38     int v2 = 1;
39     pthread_mutex_init(&(sip->lock), NULL);
40     pthread_t thread1;
41     pthread_t thread2;
42     pthread_create (&thread1,NULL,funcWCritSec,&v2);
43     pthread_create (&thread2,NULL,funcWCritSec,&v2);
44     sem_wait(&sem);
45     sem_wait(&sem);
46     pthread_mutex_destroy(&(sip->lock));
47     sem_destroy(&sem);
48     printf("%d\n", sip->value); // Should print "2".
49     return sip->value-2;
50 }
```

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 `sip->value`, something which *only* happens if the return on line 23 *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 `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;

13 sem_t sem;
14 SharedInt* sip;

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
       funcWCritSec_critSec0_msg*)msgP;
24     int * v2 = funcWCritSec_cs0msg->v2;
25     sip->value = sip->value + *v2;
26             if (sip->value < 3) {
27             sip->value = sip->value + *v2;
28                             *(funcWCritSec_cs0msg->__retval__) = ((void
       *)0);
29     *(funcWCritSec_cs0msg->__earlyReturn__) = 1;
30     return;
31         }
32 ;
33     sip->value = sip->value - *v2;
34 }

36 void *funcWCritSec(int* v2) {
37     // Do some work
38     struct funcWCritSec_critSec0_msg funcWCritSec_cs0msg;
39     funcWCritSec_cs0msg.v2 = v2;
40     int __earlyReturn__ = 0;
41     void * __retval__ = NULL;
42     funcWCritSec_cs0msg.__earlyReturn__ = &__earlyReturn__;
43     funcWCritSec_cs0msg.__retval__ = &__retval__;
44     LL_delegate_wait(&(sip->lock), funcWCritSec_critSec0,
       sizeof(funcWCritSec_cs0msg), &funcWCritSec_cs0msg);
45     if(__earlyReturn__ != 0) {
46         return __retval__;
47     }


50     // Do some more work
51 }

53 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 [17] 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` [15]).

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.

```
#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.

```
 8 typedef struct{
 9     QDLock lock;
10     int value;
11 }SharedInt;

13 sem_t sem;
14 SharedInt* sip;

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
       funcWCritSec_critSec0_msg*)msgP;
24     int * v2 = funcWCritSec_cs0msg->v2;
25     sip->value = sip->value + *v2;
26             if (sip->value < 3) {
27             sip->value = sip->value + *v2;
28                         sem_post(&sem);
29                 *(funcWCritSec_cs0msg->__retval__) = ((void *)0);
30     *(funcWCritSec_cs0msg->__earlyReturn__) = 1;
31     return;
32         }
33 ;
34     sip->value = sip->value - *v2;
35 }

37 void *funcWCritSec(int* v2) {
38     // Do some work
39     struct funcWCritSec_critSec0_msg funcWCritSec_cs0msg;
40     funcWCritSec_cs0msg.v2 = v2;
41     int __earlyReturn__ = 0;
42     void * __retval__ = NULL;
43     funcWCritSec_cs0msg.__earlyReturn__ = &__earlyReturn__;
44     funcWCritSec_cs0msg.__retval__ = &__retval__;
45     LL_delegate_wait(&(sip->lock), funcWCritSec_critSec0,
       sizeof(funcWCritSec_cs0msg), &funcWCritSec_cs0msg);
46     if(__earlyReturn__ != 0) {
47         return __retval__;
48     }


51     // Do some more work
52     sem_post(&sem);
53 }
```

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.

```
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;
}

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 dos 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 [7] and a patch for modern computers from UDEL was applied [6]. 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[5]. 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

Benchmark	Critical sections					Transformation rate
	Type A	Type B	Type C	Total	Transformed	
ocean/contiguous_partitions	4	0	0	4	4	100%
ocean/non_contiguous_partitions	4	0	0	4	4	100%
radiosity	27	3	0	30	27	90%
raytrace	11	0	0	11	11	100%
water-nsquared	8	0	0	8	8	100%
water-spatial	8	0	0	8	8	100%
volrend	14	0	0	14	14	100%
Total	76	3	0	79	76	approx. 98.57%

Table 1: Transformation rates

[5]This 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.

[8] Excludes the first timestep of the simulation.

`./OCEAN -n1026 -p8`

		With initialization		Without initialization[8]	
		Mutex locking	QD locking	Mutex locking	QD locking
1 thread	All runs	932259 µs	817823 µs	428184 µs	430585 µs
		811970 µs	814044 µs	432022 µs	428860 µs
		800895 µs	814872 µs	427630 µs	435277 µs
		801914 µs	820820 µs	429342 µs	433519 µs
		812894 µs	821756 µs	429536 µs	435190 µs
	Average	831986.4 µs	817863 µs	429342.8 µs	432686.2 µs
	Difference	-1.70%		+0.78%	
4 threads	All runs	540352 µs	491067 µs	318207 µs	299740 µs
		486601 µs	544220 µs	293834 µs	317037 µs
		513940 µs	508720 µs	309320 µs	315751 µs
		535845 µs	537887 µs	312860 µs	309435 µs
		513402 µs	537862 µs	307972 µs	315258 µs
	Average	518028 µs	523951.2 µs	308438.6 µs	311444.2 µs
	Difference	+1.14%		+0.98%	
8 threads	All runs	482960 µs	475069 µs	294470 µs	293930 µs
		491996 µs	481598 µs	308256 µs	298830 µs
		481137 µs	475365 µs	296562 µs	292886 µs
		484845 µs	494735 µs	301467 µs	304977 µs
		478743 µs	487059 µs	293817 µs	303783 µs
	Average	483936.2 µs	482765.2 µs	298914.4 µs	298881.2 µs
	Difference	-0.24%		-0.01%	

Table 2: ocean/contiguous_partitions

`./OCEAN -p8`

		With initialization		Without initialization[8]	
		Mutex locking	QD locking	Mutex locking	QD locking
1 thread	All runs	56845 µs	56731 µs	33662 µs	34041 µs
		59520 µs	60619 µs	36108 µs	37092 µs
		57576 µs	56401 µs	34235 µs	33909 µs
		60629 µs	59858 µs	36946 µs	36371 µs
		56551 µs	55989 µs	33806 µs	33711 µs
	Average	58224.2 µs	57919.6 µs	34951.4 µs	35024.8 µs
	Difference	-0.52%		+0.21%	
4 threads	All runs	23784 µs	23518 µs	13919 µs	13928 µs
		23442 µs	23361 µs	13933 µs	13218 µs
		32484 µs	23800 µs	19704 µs	14139 µs
		23890 µs	24265 µs	13874 µs	14406 µs
		33638 µs	33462 µs	19645 µs	19851 µs
	Average	27447.6 µs	25681.2 µs	16215 µs	15108.4 µs
	Difference	-6.44%		-6.83%	
8 threads	All runs	23734 µs	25243 µs	13294 µs	13588 µs
		24535 µs	23825 µs	13705 µs	13699 µs
		25127 µs	24555 µs	14919 µs	13777 µs
		23853 µs	27919 µs	13723 µs	15969 µs
		25071 µs	23780 µs	14023 µs	13598 µs
	Average	24464 µs	25064.4 µs	13932.8 µs	14126.2 µs
	Difference	+2.45%		+1.39%	

Table 3: ocean/non_contiguous_partitions

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.

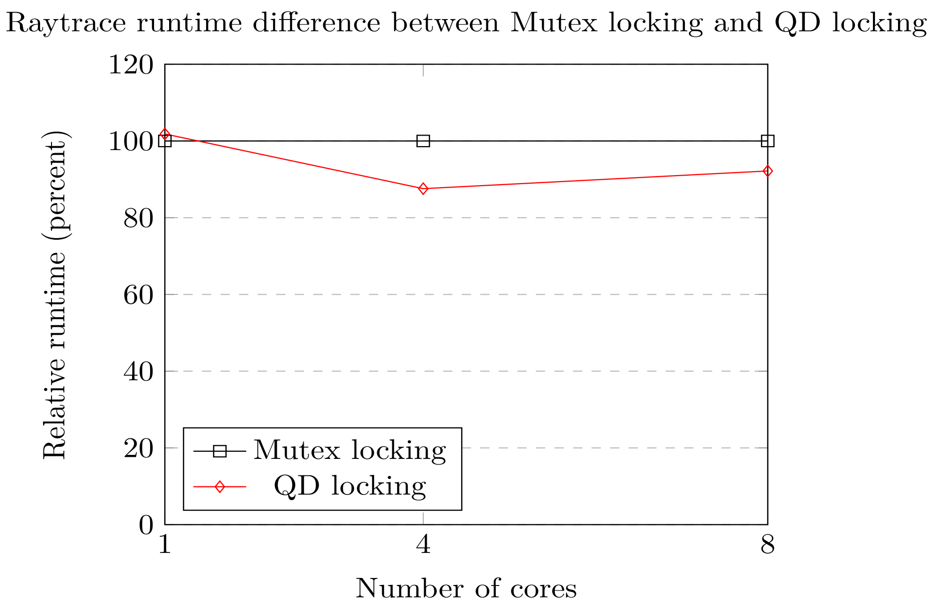
The `radiosity` benchmark turns out to have broken threading as it runs on one single CPU core regardless of what `-p` parameter is given for both Mutex and QD versions, thus no table was generated. It turns out that this is due to a known race condition [19]. The numbers indicated no significant difference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

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

		With initialization		Without initialization	
		Mutex locking	QD locking	Mutex locking	QD locking
1 thread	All runs	1174977 µs	1199371 µs	1174967 µs	1199370 µs
		1166883 µs	1188579 µs	1166883 µs	1188579 µs
		1167209 µs	1193945 µs	1167199 µs	1193944 µs
		1170403 µs	1184223 µs	1170403 µs	1184223 µs
		1173775 µs	1194041 µs	1173765 µs	1194040 µs
	Average	1170649.4 µs	1192031.8 µs	1170643.4 µs	1192031.2 µs
	Difference	+1.83%		+1.83%	
4 threads	All runs	355833 µs	330270 µs	355764 µs	330195 µs
		388364 µs	334734 µs	388287 µs	334657 µs
		410143 µs	331251 µs	410066 µs	331185 µs
		385050 µs	332290 µs	385018 µs	332223 µs
		356150 µs	331309 µs	356090 µs	331251 µs
	Average	379108 µs	331970.8 µs	379045 µs	331902.2 µs
	Difference	-12.43%		-12.44%	
8 threads	All runs	295935 µs	272410 µs	295857 µs	272361 µs
		295006 µs	269742 µs	294942 µs	269686 µs
		294223 µs	269152 µs	294170 µs	269089 µs
		293887 µs	272450 µs	293808 µs	272388 µs
		295488 µs	275416 µs	295414 µs	275337 µs
	Average	294907.8 µs	271834 µs	294838.2 µs	271772.2 µs
	Difference	-7.82%		-7.82%	

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.



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.

`./WATER-NSQUARED < ./input`

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

Table 5: water-nsquared

`./WATER-SPATIAL < ./input`

		Mutex locking	QD locking
1 thread	All runs	87129 µs	87269 µs
		87314 µs	86933 µs
		91926 µs	89237 µs
		92352 µs	89289 µs
		87213 µs	88165 µs
	Average	89186.8 µs	88178.6 µs
	Difference	-1.13%	
4 threads	All runs	23410 µs	30765 µs
		24314 µs	23189 µs
		23440 µs	23077 µs
		30667 µs	23863 µs
		23704 µs	23059 µs
	Average	25107 µs	24790.6 µs
	Difference	-1.26%	
8 threads	All runs	20418 µs	15578 µs
		15452 µs	15673 µs
		15417 µs	16659 µs
		17895 µs	15614 µs
		18235 µs	18252 µs
	Average	17483.4 µs	16355.2 µs
	Difference	-6.45%	

Table 6: water-spatial

`./VOLREND 8 ./inputs/head`

		Mutex locking	QD locking
1 thread	All runs	383022 µs	382353 µs
		381808 µs	383624 µs
		382876 µs	380247 µs
		381667 µs	393252 µs
		383257 µs	384513 µs
	Average	382526 µs	384797.8 µs
	Difference	+0.59%	
4 threads	All runs	249504 µs	237734 µs
		238627 µs	248889 µs
		236519 µs	238342 µs
		250272 µs	237165 µs
		273073 µs	259135 µs
	Average	249599 µs	244253 µs
	Difference	-2.14%	
8 threads	All runs	248082 µs	234396 µs
		236983 µs	228606 µs
		256211 µs	225960 µs
		232419 µs	245173 µs
		226024 µs	225341 µs
	Average	239983.8 µs	231895.2 µs
	Difference	-3.37%	

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 errs 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

[1] Clang static analyzer. http://clang-analyzer.llvm.org/. [Online; accessed 21-12-2016].

[2] clang::recursiveastvisitor< derived > class template reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html. [Online; accessed 10-05-2017].

[3] clang::rewriter class reference. https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html. [Online; accessed 10-05-2017].

[4] clang::tooling::replacement class reference. https://clang.llvm.org/doxygen/classclang_1_1tooling_1_1Replacement.html. [Online; accessed 10-05-2017].

[5] Gnu compiler collection documentation, chapter 23 plugins. https://gcc.gnu.org/onlinedocs/gccint/Plugins.html. [Online; accessed 21-08-2017].

[6] The modified splash-2 home page. http://www.capsl.udel.edu/splash/Download.html. [Online; accessed 21-08-2017].

[7] Splash benchmarks - gem5. http://gem5.org/Splash_benchmarks. [Online; accessed 21-08-2017].

[8] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* C++ in-depth series. Addison-Wesley, 2001.

[9] A.W. Appel and M. Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, 2004.

[10] Krste Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley" (PDF). http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf. [Online; accessed 11-05-2016].

[11] E.H. D'Hollander, G.R. Joubert, F. Peters, and U. Trottenberg. *Parallel Computing: Fundamentals, Applications and New Directions.* Advances in Parallel Computing. Elsevier Science, 1998.

[12] V. Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. http://sydney.edu.au/engineering/it/~gramoli/doc/pubs/gramoli-synchrobench.pdf. [Online; accessed 11-05-2016].

[13] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM. https://www.cs.bgu.ac.il/~hendlerd/papers/flat-combining.pdf [Online; accessed 06-08-2016].

[14] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Queue Delegation Locking. http://www.it.uu.se/research/group/languages/software/qd_lock_lib. [Online; accessed 12-05-2016].

[15] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 572–583. Springer International Publishing, 2014.

[16] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Queue delegation locking. *IEEE Transactions on Parallel and Distributed Systems*, 2017. To appear.

[17] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L Lawall, Muller, Gilles, et al. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX Annual Technical Conference*, pages 65–76, 2012. [Online; accessed 09-12-2015].

[18] Gilles Muller, Julia Lawall, Jesper Andersen, Julien Brunel, René Rydhof Hansen, Yoann Padioleau, and Nicolas Palix. Coccinelle: A Program Matching and Transformation Tool for Systems Code. `http://coccinelle.lip6.fr`. [Online; accessed 09-12-2015].

[19] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research, 04 2016.

[20] R.C. Seacord. *Secure Coding in C and C++*. SEI Series in Software Engineering. Pearson Education, 2013.

[21] David A. Wheeler. Secure Programming HOWTO Chapter 7: Design Your Program for Security. `http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html`. [Online; accessed 06-08-2016].

[22] Kjell Winblad. qd_lock_lib Tutorial. `https://github.com/kjellwinblad/qd_lock_lib/wiki/Tutorial`. [Online; accessed 19-05-2016].