Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση Συστήματος Τύπων για την Αποφυγή Αδιεξόδων σε C/Pthreads

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΒΕΚΡΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2011

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση Συστήματος Τύπων για την Αποφυγή Αδιεξόδων σε C/Pthreads

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΠΑΝΑΓΙΩΤΗΣ ΒΕΚΡΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Επίκ. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου 2011.

Νικόλαος Παπασπύρου Επίκ. Καθηγητής Ε.Μ.Π.	Κωστής Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2011

Παναγιώτης Βεκρής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Τα συστήματα παράλληλης επεξεργασίας έχουν γνωρίσει ραγδαία ανάπτυξη τις τελευταίες δεκαετίες, η οποία έχει επιδράσει πολλαπλασιαστικά στην υπολογιστική τους ισχύ, αλλά παράλληλα έχει εισαγάγει αρκετές προκλήσεις στον προγραμματισμό τους, μία από τις οποίες είναι η απουσία των αδιεξόδων που οφείλονται στη χρήση κλειδωμάτων. Σε αυτή τη διπλωματική αρχικά εκθέτουμε στατιστικά στοιχεία σχετικά με τη χρήση των προτύπων κλειδώματος σε μια βάση πραγματικών προγραμμάτων που συγκεντρώσαμε. Εν συνεχεία παρουσιάζουμε την υλοποίηση ενός συστήματος τύπων για την αποφυγή αδιεξόδων, όπου η έρευνα μέχρι στιγμής έχει αποδώσει περιορισμένα αποτελέσματα σε σχέση με την πρόληψη αδιεξόδων. Η υλοποίησή μας έχει ως στόχο προγράμματα σε C που χρησιμοποιούν το πρότυπο των Pthreads.

Οι καινοτομίες του εργαλείου που αναπτύξαμε είναι ότι μπορεί να αντιμετωπίσει περιπτώσεις μη δομημένου κλειδώματος μεταβλητών, δεν επιβάλει αυστηρή σειρά στο κλείδωμα μεταβλητών, όπως οι περισσότερες προσεγγίσεις στην πρόληψη αδιεξόδων, και δεν απαιτεί από το χρήστη να επισημειώσει με οποιονδήποτε τρόπο το αρχικό πρόγραμμα. Το εργαλείο αποτελείται τόσο από ένα τμήμα στατικής ανάλυσης των προγραμμάτων εισόδου, το οποίο συγκεντρώνει πληροφορίες σχετικά με το κλείδωμα μεταβλητών, όσο και από μια βιβλιοθήκη χρόνου εκτέλεσης που χρησιμοποιώντας την στατικά συγκεντρωμένη πληροφορία αποφαίνεται κατά πόσο είναι ασφαλές να γίνει ένα κλείδωμα, ώστε το σύστημα να μην περιέλθει σε αδιέξοδο. Τέλος, παρουσιάζουμε μια σειρά από παραδείγματα εκτέλεσης όπου καταδεικνύεται η ορθότητα λειτουργίας του εργαλείου και αξιολογούνται οι επιδόσεις του σε σχέση με την αρχική έκδοση των προγραμμάτων.

Λέξεις κλειδιά

Αποφυγή αδιεξόδων, ταυτόχρονος προγραμματισμός, σύστημα τύπων, C, Pthreads.

Abstract

The last few decades have seen great rise in the development of parallel systems, which has multiplied their computational power, but at the same time has introduced several challenges in their programming, deadlock freedom being one of the most important. In this thesis, we begin with presenting statistics concerning the usage of lock primitives that were gathered from a code base of real world programs. We then introduce the implementation of a type and effect system for deadlock avoidance, which is a field where research results so far have been limited compared to deadlock prevention. Our approach targets C programs that use the Pthreads specifications.

The novelties of our tool include its ability to handle non block-structured locking, the fact that it does not impose a strict lock acquisition order, like most deadlock prevention approaches, and that it does not require the user to provide any annotation on the original source code. The tool comprises both a static analysis part, which collects information regarding locking, and a runtime library which uses this gathered information and decides whether it is safe to grant a lock, in order to avoid deadlocks. We finally present a series of benchmarks that manifest the correctness of our tool and demonstrate its performance in terms of the original program.

Key words

Deadlock avoidance, concurrent programming, type and effect system, C, Pthreads.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Νικόλαο Παπασπύρου και τον καθηγητή Κωστή Σαγώνα για την καθοδήγηση που μου παρείχαν για την πραγμάτωση αυτής της εργασίας, καθώς επίσης και τον διδακτορικό φοιτητή Πρόδρομο Γερακιό για την υποστήριξη του σε όλες της φάσεις της υλοποίησης της διπλωματικής μου.

Τέλος, ευχαριστώ την οικογένειά μου για την ηθική και υλική υποστήριξή τους σε όλη τη διάρκεια των μαθητικών και φοιτητικών μου χρόνων.

Παναγιώτης Βεκρής,

Αθήνα, 18η Ιουλίου 2011

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-11, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2011.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

Chapter 1

Introduction

Some of the main findings of this thesis are contained in a manuscript, co-authored by P. Gerakios, N. Papaspyrou and K. Sagonas and the author of the present thesis, that has been submitted for publication.

1.1 Objectives

This thesis aims at implementing a tool for deadlock avoidance that targets C/Pthreads programs. The tool is based on a type and effect system, developed by Gerakios *et al.* [31] and fully presented as a part of Prodromos Gerakios' forthcoming doctoral dissertation. It uses a static analysis phase, during which useful information is gathered for each locking primitive in the program. This information is used at runtime, to decide whether a lock can be safely granted without the possibility of a future deadlock. The tool that we present and evaluate in this thesis is an extension of the tool implemented earlier by Prodromos Gerakios. We also provide statistics regarding the use of locks in C/Pthreads programs, collected from a relatively large code base of real-world open-source applications that are available on the Internet.

1.2 Motivation

Concurrent execution of multiple threads has drastically boosted the performance of multicore and multiprocessor systems. For instance, UI applications that use separate threads to handle their front-end interface are more responsive and multithread servers have become more robust. However, at the same time new hazards unknown to single thread execution have arisen. These bugs are usually quite burdensome, as they are notoriously hard to detect, reproduce and amend.

Among the most common hazards connected with multithreaded execution are deadlocks caused by faulty usage of locking primitives. Deadlocks are usually the consequence of a cyclic lock acquisition order between threads. Two or more threads are deadlocked when each of them is waiting for a lock that has been acquired and is held by another thread. A deadlock situation is a particularly difficult one to recover from. Once a system has fallen into a deadlock it can only be restored by using highly invasive means, for example forcibly removing a lock from a thread that holds it, or even killing a thread that is involved in a deadlock. Therefore, eliminating deadlocks after they have occurred is not the optimum solution; deadlocks should be treated by using preventive measures.

Several approaches have been proposed to achieve deadlock freedom, namely to exempt programs from deadlocks. Most of them are static type-based approaches for deadlock prevention. They usually impose a strict lock acquisition order, so as to exclude cases of circular wait in resource acquisition, which is a condition that must hold in order for a deadlock to occur. This approach, however, is rather

restrictive in the sense that it rejects programs that will not necessarily lead to a deadlock, and therefore hinders the programming language's expressiveness.

To allow a larger class of programs to type check, an alternative approach had to be followed that would *dynamically* avoid deadlocks by utilizing locking information gathered at compile time. Boudol [13] followed this direction, but the type system he created was still restrictive as it required that locking was done in a block structured way, just like Java's `synchronized` blocks, and that matching lock operations resided in the same function (a common example of the opposite case are wrapper functions for lock operations).

What needed to be done towards this direction was to create a system that would guarantee deadlock freedom and allow even more programs to type check by raising the requirement that locks should be used in a block structured way. This system was provided by Gerakios *et al.* [31]. The implementation of this system was the primary motivation for this thesis.

One might ask, however, whether it was worth creating a new system that would handle cases of non block structured locking, as using locks in a structured way is the prevalent way of using locks. To justify that the effort put into that direction would be meaningful, we ran a series of statistics regarding the use of lock primitives in real-world projects in C/Pthreads. We gathered a code base of freely distributed projects and ran an analysis that among others extracted information about the type of locking patterns that were used. As it turned out, approximately 63% of the sample of projects we gathered were using locks in a non block structured way, and a fair amount of projects used wrapper functions around lock and unlock operations. These cases would have been rejected by an implementation of the type system proposed by Boudol.

Furthermore, we also wanted to have an actual tool that would guarantee deadlock avoidance, as the related approaches we found in bibliography were either never tested in a low-level language like C [13], or their implementation was not distributed freely [58], so we had no point of reference. Having a concrete implementation of such a tool would allow us to evaluate its performance in real applications and determine whether it would be a viable solution for applications or system code prone to deadlocks. We chose to target C programming language and the Pthreads API as both of them have been around for quite a long time and a considerable amount of projects have been developed using them. At the end of the day, the modest overhead that our tool induced on several projects that we tested allowed us to conclude that our approach could be used more broadly on mutlithread projects.

Motivated by the reasons outlined in the preceding paragraphs, this thesis presents the implementation of a type and effect system [31] that guarantees deadlock avoidance for C/Pthreads. The tool that we developed takes projects written in C as input, gathers information regarding lock usage, and annotates the original source code. The output is a modified C code that is then compiled normally and linked to a runtime library we created, which basically overrides the original Pthread lock and unlock functions. At execution time, before granting any lock, our runtime system checks whether granting the lock would lead to a deadlock prone state, and if it does it defers the acquisition of the lock.

1.3 Outline of the Thesis

The rest of this thesis is organized as follows: Chapter 2 gives an introduction to concurrent programming and explains some of the hazards associated with it. Chapter 3 describes how lock primitives are used in C programming language, introduces the tools we used to analyze C programs and presents some statistics concerning lock usage in real-world multithread projects. In Chapter 4 we describe and reason about the main idea of our deadlock avoidance strategy, and in Chapter 5 we elaborate on some of the main aspects of our deadlock avoidance tool, by discussing the major parts of its workflow. Finally, Chapter 6 evaluates the performance of our tool on a series of benchmarks and Chapter 7 concludes.

Chapter 2

Concurrent Programming

2.1 Overview

2.1.1 Definition

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel. The driving motion behind this plot is translating the raw potential of computer architecture into greater performance and expanded capability of computer systems. Concurrent programs can be executed sequentially on a single processor by interleaving the execution steps of each computational process, or executed in parallel by assigning each computational process to each of a set of processors that may be close or distributed across a network.

Traditionally, the word *parallel* is used for systems in which the executions of several programs overlap in time by running them on separate processors. The word *concurrent* is reserved for potential parallelism, in which the executions may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of a small number of processors, often only one. Concurrency is an extremely useful abstraction because it helps us think of all processes as being executed in parallel. Conversely, even if the processes of a concurrent program are actually executed in parallel on several processors, understanding its behavior is greatly facilitated if we impose an order on the instructions that is compatible with shared execution on a single processor. Like any abstraction, concurrent programming is important because the behavior of a wide range of real systems can be modeled and studied without unnecessary detail.

2.1.2 Challenges

The main challenges in designing concurrent programs are ensuring the correct sequencing of the interactions or communications between different computational processes, and coordinating access to resources that are shared among processes. If the processes were totally independent, the implementation of concurrency would only require a simple scheduler to allocate resources among them. But even trivial examples can illustrate that the communication of computing processes is inevitable. If an I/O process accepts a character typed on a keyboard, it must somehow communicate it to the process running the word processor, and if there are multiple windows on a display, processes must somehow synchronize access to the display so that images are sent to the window with the current focus.

Safe and efficient synchronization and communication is extremely difficult to implement. Situations where the execution of a program halts are often attributed to errors occurring in synchronization or communication of processes. However, since such problems are time- and situation-dependent, they are difficult to reproduce, diagnose and correct.

2.1.3 Processes and Threads

At this moment a distinction should be made between the terms *process* and *thread*:

Process: A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently. A computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program. A computer process consists of the following resources:

- An *image* of the executable machine code,
- *Memory*, which includes the executable code, a call stack (that keeps track of subroutines) and a heap (to hold intermediate computation data)
- Operating system *descriptors* of resources that are allocated to the process.
- *Security* attributes, such as the process owner and the process's set of permissions.
- Processor *state* (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

Thread: A thread of execution is the smallest unit of processing that can be scheduled by an operating system. In most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).

On a single processor, multithreading generally occurs by time-division multiplexing (multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multicore system, the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.

Switching the CPU from one process or thread to another, known as a *context switch*, requires saving the state of the old process or thread and loading the state of the new one. Since there may be several hundred context switches per second, context switches can potentially add significant overhead to an execution.

In order to implement concurrent programs there is a number of different methods that can be used, such as implementing each computational process as an operating system process, or implementing the computational processes as a set of threads within a single operating system process. For the rest of this thesis we only deal with threads.

2.1.4 Communication Methods

Communication between concurrent components can either be hidden from the programmer, or handled explicitly.

Explicit communication falls under two main classes:

Shared memory: In this case of communication, memory can be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.

As the communicating components may alter the contents of shared memory locations, this style of programming usually requires the application of some form of synchronization, for example mutexes, semaphores or monitors.

Message passing: In this case, concurrent components communicate by exchanging messages (exemplified by *Erlang* and *occam*). The exchange of messages may be carried out asynchronously, or may use a rendezvous style in which the sender blocks until the message is received. Asynchronous message passing may be reliable or unreliable. Message-passing concurrency tends to be easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming.

In this thesis, we focus our attention on shared memory and explicit synchronization methods.

2.1.5 Thread Synchronization

A fundamental problem of concurrent programming is the *critical section problem*. A code segment that accesses shared variables (or other shared resources) and that has to be executed as an atomic action is referred to as a *critical section*. To avoid the simultaneous use of a common resource by critical sections, programmers use *mutual exclusion* algorithms. There are both software and hardware solutions for enforcing mutual exclusion.

Hardware On a uniprocessor system a common way to achieve mutual exclusion inside kernels is to disable interrupts for the smallest possible number of instructions that will prevent corruption of the shared data structure, the critical section. This prevents interrupt code from running in the critical section, that also protects against interrupt-based process-change.

In a computer in which several processors share memory, an indivisible test-and-set of a flag could be used in a tight loop to wait until the other processor clears the flag. The test-and-set performs both operations without releasing the memory bus to another processor. When the code leaves the critical section, it clears the flag. This is called a “spinlock” or “busy-wait”.

Similar atomic multiple-operation instructions, e.g., compare-and-swap, are commonly used for lock-free manipulation of linked lists and other data structures.

Software Software solutions that provide mutual exclusion have two major requirements: (a) safety: it should be impossible for two processes to be in critical section simultaneously, and (b) liveness: it should be impossible for any process to be waiting when no other other process is contending or using a critical section. Some solutions that correctly solve this issue, by using busy-wait are: Dekker’s algorithm, Peterson’s algorithm and Lamport’s bakery algorithm.

Unfortunately, spin locks and busy waiting take up excessive processor time and power and are considered anti-patterns in almost every case. In addition, these algorithms do not work if Out-of-order execution is utilized on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread.

Hybrid The solution to these problems is to use synchronization facilities provided by an operating system’s multi threading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist. Some of the most common synchronization primitives are locks, reentrant mutexes, semaphores and monitors.

Of the above mechanisms our concern is focused on locks, and more specifically the lock support provided by the POSIX thread API. The idea behind the lock is simple: enforce limits on access to a resource in an environment where there are many threads of execution.

2.2 Concurrency Hazards

Correctly engineered concurrent programming must abide by an extra set of rules compared to its sequential counterpart. Reads and writes to memory and access to shared resources need to be regulated so that conflicts do not arise. One of the most common ways to achieve this is by using synchronization methods. Additionally, it often becomes necessary for threads to coordinate to get a certain job done. As a direct result of these additional requirements, it becomes nontrivial to ensure that threads are continually making consistent and adequate forward progress. Synchronization and coordination deeply depend on timing, which is nondeterministic and hard to predict and test for. Therefore, there are hazards that can cause programs to break either by crashing or by corrupting memory.

Lock-based resource protection and thread synchronization may have some negative side effects, for example:

- Blocking: threads have to wait until a lock (or a whole set of locks) is released.
- Lock handling adds overhead for each access to a resource, even when the chances for collision are very rare.
- Lock contention limits scalability and adds complexity.
- Balances between lock overhead and contention can be unique to given problem domains (applications) as well as sensitive to design, implementation, and even low-level system architectural changes. These balances may change over the life cycle of any given application/implementation and may entail tremendous changes to update (re-balance).
- Priority inversion. High priority threads cannot proceed if a low priority thread is holding the common lock.
- There must be sufficient resources - exclusively dedicated memory, real or virtual - available for the locking mechanisms to maintain their state information in response to a varying number of contemporaneous invocations, without which the mechanisms will fail, or “crash” bringing down everything depending on them and bringing down the operating region in which they reside.

Locks are also vulnerable to fatal conditions that are often very subtle and may be difficult to reproduce reliably:

- Failure to protect shared data is a fault called *data race*.
- The situation where no thread can make progress because each is blocked on a lock held by some other thread is called a *deadlock*, and is thoroughly discussed in this thesis.

Data races

Data races occur in multithreaded programs when one thread accesses a memory location at the same time another thread writes to it. While some races are benign, those that are erroneous can have disastrous consequences, as they had for example in the *Therac-25* accidents [38], or the Northeast Blackout [46] of 2003 (August 14th 2003) that was attributed to a race condition existing in General Electric Energy’s Unix-based XA/21 energy management system. Moreover, race freedom is an important program property in its own right, because race-free programs are easier to understand, analyze and transform.

A simple example of a race condition is the following:

Assume that two threads T_1 and T_2 each want to increment the value of a global integer by one. This could occur if both thread executed the simple snippet of C code shown in Figure 2.1.

Ideally, the following sequence of operations would take place:

1. Integer var = 0 (memory)
2. T_1 reads the value of var from memory into register1: 0
3. T_1 increments the value of var in register1: (register1 contents) + 1 = 1
4. T_1 stores the value of register1 in memory: 1
5. T_2 reads the value of var from memory into register2: 1
6. T_2 increments the value of var in register2: (register2 contents) + 1 = 2
7. T_2 stores the value of register2 in memory: 2
8. Integer var = 2 (memory)

```
1  #include <pthread.h>
2
3  int var = 0;
4
5  void* child_fn ( void* arg ) {
6    var++; /* Unprotected relative to sibling */
7    return NULL;
8  }
9
10 int main ( void ) {
11   pthread_t child1, child2;
12   pthread_create(&child1, NULL, child_fn, NULL);
13   pthread_create(&child2, NULL, child_fn, NULL);
14   pthread_join(child1, NULL);
15   pthread_join(child2, NULL);
16   return 0;
17 }
```

Figure 2.1: Simple race condition.

In the case shown above, the final value of `var` is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

1. Integer var = 0 (memory)
2. T_1 reads the value of var from memory into register1: 0
3. T_2 reads the value of var from memory into register2: 0
4. T_1 increments the value of var in register1: (register1 contents) + 1 = 1
5. T_2 increments the value of var in register2: (register2 contents) + 1 = 1
6. T_1 stores the value of register1 in memory: 1
7. T_2 stores the value of register2 in memory: 1
8. Integer var = 1 (memory)

The final value of `var` is 1 instead of the expected result of 2. This occurs because the increment operations of the second case are not mutually exclusive (operations that cannot be interrupted while accessing some resource such as a memory location). In the first case, T_1 was not interrupted while accessing the variable `var`, so its operation was mutually-exclusive. A correct program would protect `var` with a lock of type `pthread_mutex_t`, which is acquired before each access and released afterwards.

Data Race Detection

The example illustrated above was a rather obvious case of a data race. Finding data races, however, is generally a difficult task. Detecting such errors with testing often depends on intricate sequences of low-probability events, which makes them sensitive to timing dependencies, workloads, the presence or absence of print statements, compiler options, or slight differences in memory models. Furthermore, even if a test case happens to trigger an error it can be difficult to tell what has happened. Data races are in particular hard to observe since they often quietly violate data structure invariants rather than cause immediate crashes. The effects of this violation only manifest millions of cycles after the error occurred, making it hard to trace back to the root cause.

There have been many tools developed to attack the problem: dynamic, post-mortem, and static as well as model checking.

Most dynamic data race detection tools are based on one of the following algorithms: happens-before, lockset or both (the hybrid type). Lamport's *happens-before* algorithm [37] checks that conflicting memory accesses from different threads are separated by synchronization events. Eraser [49] data race detector, one of the best known dynamic tools, dynamically tracks the set of locks held during program execution and uses them to compute the intersection of all locks held when accessing shared state. Shared locations that have an empty intersection are flagged as not being consistently protected. Tools similar to Eraser have been developed for Cilk programs [17] and Java [59, 28]. There have been several other approaches for dynamic datarace detection [18, 45, 47]. Additionally, Valgind [42], a programming suite for memory debugging, memory leak detection, and profiling, also includes some tools for detecting data races: *Helgrind* is a thread error detector that uses Eraser's algorithm to detect data races and *ThreadSanitizer* [50] uses the hybrid algorithm (based on happens-before and locksets).

The strength of dynamic tools is that by operating at runtime they only visit feasible paths and have accurate views of the values of variables and aliasing relations. However, dynamic monitoring has a heavy computational price, due to the runtime overhead involved. Furthermore, their reliance on invasive instrumentation typically precludes their use on low-level code such as OS kernels, device drivers and embedded systems, that nonetheless are applications prone to concurrency errors. Finally, they only find errors on executed paths, which by no means account for the entire set of feasible paths.

Post-mortem techniques [32] analyze log or trace data after the program has executed in a manner similar to dynamic techniques, hence suffering from the same limitations of only finding errors along executed paths. Model checking is a formal verification technique that can be viewed as a more comprehensive form of dynamic testing that exhaustively tests a simplified description of the code on all inputs [56, 16, 21]. Unfortunately, while model checking manages to explore a wider state space, it is rarely used for large systems, as it requires significant effort to specify the system and scale it down enough to execute in the model checker environment.

At the other side of the spectrum are static tools. These techniques can provide significant advantages for large code bases. Unlike dynamic approaches, static analyses do not require executing code: they immediately find errors in obscure code paths that are difficult to reach with testing. Some of the first approaches were the Warlock tool for finding races in C programs and the Extended Static Checking [23, 29]. There have also been effective techniques for static race detection that can be applied to tens of thousands of lines of C code [48] and a little over a hundred thousand lines of Java code [40]. Some attempts that can run on millions of lines of code, like RacerX [25], have been considered extremely unsound, as they miss many errors. Finally, RELAY [57] introduced the notion of a relative offset, which allows functions to be summarized independent of the calling contexts, and enabled a race detection analysis that scales to millions of lines of C code.

2.3 Deadlocks

2.3.1 Definitions

One of the most annoying problems with concurrent systems, which is the main topic of this thesis, is the risk of entering into a deadlock situation. The notion of deadlock in computer science was introduced by E. G. Coffman *et al.* [19]:

A Coffman deadlock refers to a specific condition where two or more processes are waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain. Deadlock is a common problem in multiprocessing where many processes share mutually exclusive resources.

As a somewhat simplified example, suppose we have a thread T_1 that has just produced a request for a file on disk, and that T_1 must suspend operation until the disk becomes available. Suppose another thread T_2 has control of the (single) disk channel but that it is still waiting for T_1 to complete the updating of a (shared) file that contains information required by T_2. In order for work to proceed, the disk channel must be given over to T_1. But if we assume that T_2 is in the middle of some file on disk, then, because of hardware or software limitations, a preemption of the disk from T_2 can be effectively tantamount to a sacrifice in the processing already accomplished by T_2. Under these circumstances, if T_1 and T_2 have no (temporary) alternative to waiting, they will be deadlocked and never able to proceed.

As shown by this example, deadlocks, or "deadly embraces", as E.W. Dijkstra has called them, can arise even though no single task requires more than the total resources available in the system. Moreover, deadlocks can arise whether the allocation of resources is the responsibility of the operating system (as is normally the ease) or of the programs' themselves. The example also illustrates that the term "resource" in the context of deadlocks, can apply not only to devices, processors and storage media, but also to programs, subroutines, and data. Many of these permit only *exclusive* use by one thread at a time, but some (for example read-only programs) may be *shared* by more than one thread. The mutual exclusion of several threads using the same resource may be achieved by allocating the use of that resource to only one thread, or, if access to the resource is not directly under the control of the operating system, by using locking operations.

According to Coffman *et al.*, a set of threads reaches a deadlocked state when the following conditions hold:

- *Mutual exclusion*: Threads claim exclusive control of the locks that they acquire.
- *Hold and wait*: Threads already holding locks may request (and wait for) new locks.
- *No preemption*: Locks cannot be forcibly removed from threads; they must be released explicitly by the thread that acquired them.

The first three conditions are necessary but not sufficient for a deadlock to exist. For a deadlock to actually take place, a fourth condition is required:

- *Circular wait*: two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain holds. When circular waiting is triggered by mutual exclusion operations it is sometimes called lock inversion.

A simple example that illustrates the fourth condition often arises when manipulating bank accounts. In our setting, a bank account will simply be a memory location containing an integer value. Suppose we want to define a function to deposit some amount x on the account y. Using ML notation $(!y)$ to get the contents of the memory location y, this function can be defined as $\lambda x.\lambda y.(y :=!y + x)$, i.e. a

function that gets a number x and a memory pointer y as arguments, and assigns the previous value of $!y$ increased by x to y. This definition, however, is prone to race conditions: two concurrent deposits may have the effect of only one of them, if both read the current amount before it has been updated by the other thread.

To solve this problem, it is enough to make the deposit function, taking for the update operation $y :=!y + x$, an exclusive access to the bank account to update, that is y. For simplicity we assume that there is a construct in the programming language, say lock y in e to lock the reference y for the purpose of performing the operation e with an exclusive access to y. In this example the locks appear to be, transparently to the programmer, associated with the resources in a block-structured way, as in Java. For the rest of this thesis this is not mandatory. Using unstructured locking this example could be: (lock y; e; unlock y). The deposit function could be defined as follows:

$$\text{deposit } = \lambda x.\lambda y.(\text{lock } y \text{ in } y :=!y + x)$$

Similarly, a function to withdraw some amount from an account would be:

$$\text{withdraw } = \lambda x.\lambda y.(\text{lock } y \text{ in (if } !y \geq x \text{ then } y :=!y - x \text{ else error)})$$

From this another function can be defined, to transfer some amount x from an account y to another one z, as $\lambda x.\lambda y.\lambda z.(\text{withdraw } x\ y\,;\ \text{deposit } x\ z)$. It has been argued [34] that this function should ensure the property that another thread cannot see the intermediate state where y has decreased, but z has not yet been credited. This can be achieved by defining:

$$\text{transfer } = \lambda x.\lambda y.\lambda z.(\text{lock } y \text{ in withdraw } x\ y\,;\ \text{deposit } x\ z)$$

We introduce here the notion of *reentrant* locks: a thread that temporarily "possesses" a reference, like y in this example, is not blocked in locking it twice. Now suppose that two transfers are performed concurrently, from account a to account b, and in the converse direction. That is, we have to execute something like:

$$(\text{transfer } 100\ a\ b)\ \|\ (\text{transfer } 100\ b\ a)$$

Clearly there is a danger of deadlock here: if both operations first perform the withdrawals, locking respectively a and b, they are then blocked in trying to lock the other account in order to perform the deposits.

2.3.2 Deadlock Freedom

Deadlock freedom can be guaranteed by denying at least one of the conditions mentioned earlier *before* or *during* program execution. Thus, the following three strategies guarantee deadlock freedom:

- *Deadlock prevention*: At each point of execution, ensure that at least one of the above conditions is not satisfied. Thus, programs that fall into this category are correct by design.
- *Deadlock detection and recovery*: A dedicated observer thread determines whether the above conditions are satisfied and preempts some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress.
- *Deadlock avoidance*: Using information that is computed in advance regarding thread resource allocation, determine whether granting a lock will bring the program to an unsafe state, i.e., a state which can result in deadlock, and only grant locks that lead to safe states.

Several type systems have been proposed that guarantee deadlock freedom, the majority of which is based on deadlock prevention and avoidance.

Deadlock Prevention

In the deadlock prevention category, one finds type and effect systems that guarantee deadlock freedom by statically enforcing a global lock acquisition order that must be respected by all threads. We proceed by presenting some notable approaches towards this end.

- Flanagan and Abadi [27] show that both race conditions and deadlocks are often avoided through careful programming discipline, which can be captured by a set of static rules. They created a type system that eliminates race condition by providing rules for proving that a thread holds a given lock at a given program point, based on *singleton types* and *permissions*, and enhanced it with universal and existential types. They also extended their type system so that it avoids deadlocks, by imposing a strict partial order on locks, and respecting this order when acquiring them.

- Boyapati *et al.* [15] introduced ownership types, which provide a statically enforceable way of specifying object encapsulation and enable local reasoning about program correctness in object-oriented languages. They also provided [14] a type system that allows programmers to partition the locks into a fixed number of equivalence classes and specify a partial order among the equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in descending order. Their system also allows programmers to use recursive tree-based data structures to describe the partial order. For example, programmers can specify that nodes in a tree must be locked in the tree order. Additionally, mutations to the data structure that change the partial order at runtime are also allowed. The type checker statically verifies that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. Ownership types are useful for preventing data races and deadlocks because the lock that protects an object can also protect its encapsulated objects.

- Kobayashi presented [35] a type system for the π-calculus that guarantees deadlock-freedom, by incorporating information flow analysis for the π-calculus and a complete type inference algorithm. One of the key ideas in his system was to extend channel types with *channel usages*, which express how each channel is used for input/output and whether each input/output is guaranteed to succeed. He later extended [36] his previous work by making it more expressive and enabling the analysis of more realistic programs by handling recursive data structures like lists.

- Suenaga presented [53] a type-based deadlock-freedom verification for concurrent programs with non block-structured lock primitives and mutable references, which are heavily used in real-world software. His type system verifies deadlock-freedom by guaranteeing that locks are acquired in a specific order by using lock levels and that an acquired lock is released exactly once by using obligations and ownerships.

- Vasconcelos *et al.* [20] developed a type system and a type checker for a multithreaded lock-based polymorphic typed assembly language (MIL) that ensures that well-typed programs do not have race conditions. They later extended [55] this work by taking into consideration deadlocks. The extended type system verifies that locks are acquired in the proper order. Towards this end they require a language with annotations that specify the locking order. Rather than asking the programmer (or the compiler's back-end) to specifically annotate each newly introduced lock, they present an algorithm to infer the annotations. The result is a type checker whose input language is non-decorated as before, but that further checks that programs are exempt from deadlocks.

Deadlock Avoidance

Despite the existence of the well-known Dijkstra's Banker's algorithm [24], deadlock prevention and deadlock detection and recovery are in practice, by far, the most popular. Deadlock prevention, lends itself to using static analysis techniques. Using a strict lock acquisition order, however, is a constraint we want to avoid. It is not hard to come up with an example that shows that imposing a partial order on locks is too restrictive. The simplest of such examples, which include the example with the bank transactions mentioned earlier, can be reduced to program fragments of the form:

$$(\text{lock } x \text{ in } \ldots \text{ lock } y \text{ in } \ldots) \parallel (\text{lock } y \text{ in } \ldots \text{ lock } x \text{ in } \ldots)$$

In a few words, there are two parallel threads which acquire two distinct locks, x and y, in reverse order. When trying to find a partial order $\leq$ on locks for this program, the type system or static analysis tool will fail: it will deduce that $x \leq y$ must be true, because of the first thread, and that $y \leq x$ must be true, because of the second. In short, there is no partial order that satisfies these constraints. Thus, programs containing such patterns will be rejected in all of the systems discussed above. Even the type and effect system of Boyapati *et al.* [14], which allows for some controlled changes to the partial order of locks at runtime by permitting conservative updates on directed acyclic lock graphs, because there is no acyclic data structure that captures the cyclic dependencies between locks x and y of this program fragment.

This thesis explores a different direction, namely *deadlock avoidance*. Recent work on this field includes Boudol's type and effect system [13]. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The runtime system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is available to the requesting thread. The main advantage of Boudol's type system is that it allows a larger class of programs to type check and thus increases the programming language expressiveness as well as concurrency by allowing arbitrary locking schemes.

The previous example can be rewritten in Boudol's language as follows, assuming that the only lock operations in the two threads are those visible:

$$(\text{lock}_{\{y\}} x \text{ in } \ldots \text{ lock}_{\{\emptyset\}} y \text{ in } \ldots) \parallel (\text{lock}_{\{x\}} y \text{ in } \ldots \text{ lock}_{\{\emptyset\}} x \text{ in } \ldots)$$

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in any order. At runtime, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x. The second lock operation need not ensure anything, as its future lockset is empty. (The handling is symmetric for the second thread.)

The disadvantage in Boudol's work is that it heavily relies on the assumption that locking is block-structured. In fact, the soundness of his system in the presence of lock aliasing is guaranteed by assuming that locks are reentrant and are released in the reverse order in which they were acquired. The example in Figure 2.2, written in a simple λ-calculus style language, illustrates this. It uses three shared variables, x, y and z, ensuring at each step that no unnecessary locks are held. It is assumed here that the long computations do not acquire or release any locks.

In our naïvely extended (and broken, as we will see) version of Boudol's type and effect system, the program in Figure 2.2(left) will type check. The future lockset annotations of the three locking operations in the body of f are $\{y\}$, $\{z\}$ and $\{\emptyset\}$, respectively. Now, function f is used by instantiating both x and y with the same variable a, and instantiating z with a distinct variable b. In terms of the calling function's context, the locking sequence would naïvely be translated in the sequence shown in Figure 2.2(right). In order for the program to work, locks have to be reentrant. This roughly means that if a thread holds some lock, it can try to acquire the same lock again; this will immediately succeed, but then the thread will have to release the lock *twice*, before it is actually released.

```
let f = λx.λy.λz.
    lock_{y} x ;                          lock_{a} a ;
    long_computation x;                   long_computation a;
    lock_{z} x ;                          lock_{b} a ;
    long_computation x y;                 long_computation a a;
    unlock x ;                            unlock a ;
    lock_{∅} z ;                          lock_{∅} b ;
    unlock z ;                            unlock b ;
    long_computation y z;                 long_computation a b;
    unlock y                              unlock a
in f a a b
```

Figure 2.2: A program which is typable by a naïve extension of Boudol's system before substitution (left) but not after (right).

Even with reentrant locks, however, it is easy to see that the program in Figure 2.2(right) does not type check with the present annotations. The first lock operation for a now matches with the last (and not the first as it did previously) unlock operation; this means that a will remain locked during the whole execution of the program. In the meantime b is locked, so the future lockset annotation of the first lock operation should contain b, but it does not, because the future lockset was computed statically in terms of the context of function f. (The annotation of the second lock operation contains b, but blocking there if lock b is not available does not prevent a possible deadlock, as lock a has already been acquired.) From a more pragmatic point of view, if a thread running in parallel with the thread in Figure 2.2(right) already holds b and, before releasing it, is about to acquire a, a deadlock can occur. The naïve extension also fails for another reason: Boudol's system is based on the assumption that calling a function cannot affect the set of locks that are held. This is obviously not true, if non lexically-scoped locking operations are to be supported.

Another tool that combines static and dynamic techniques, in a way quite similar to the one discussed in this thesis, is Gadara [58]. Gadara employs whole program analysis to model programs and discrete control theory to synthesize a concurrent logic that avoids deadlocks at run time. Gadara targets C/Pthreads programs and claims to avoid deadlocks quite efficiently because it performs the majority of its deadlock avoidance computations offline. It uses the notion of control places to decide whether it is safe to admit a lock acquisition. More precisely, a lock acquisition can only proceed when all the control places associated with the lock are available. The mostly static approach followed by Gadara, as well as the lack of alias analysis, results in an over-approximation of the set of runtime locks associated with a control place.

Chapter 3

Locking in C Programming Language

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. This interface has been specified by the IEEE POSIX 1003.1c standard (1995) and the implementations adhering to this standard are referred to as *POSIX threads*, or *Pthreads*. Our deadlock avoidance tool targets C language programs that use the Pthreads API, so it is rather meaningful to present some of its main features.

3.1 POSIX Threads

The primary motivation of using Pthreads instead of processes is the cost of creating and managing a process. Creating threads requires much less operating system overhead and managing them fewer system resources.

There are several considerations when designing parallel programs, such as:

- What type of parallel programming model to use
- Problem partitioning
- Load balancing
- Communications
- Data dependencies
- Synchronization and race conditions
- Memory issues
- I/O issues

The Pthreads API provides several subroutines that can be grouped into four major groups:

3.1.1 Thread management

In this group one finds routines that work directly on threads — creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.). Initially the main program comprises a single thread. All other threads must be explicitly created by using the `pthread_create` function, which creates a new thread and makes it executable. The `pthread_create` routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the `pthread_create` routine.

A thread can be terminated by:

– Returning from its starting routine.
– Making a call to the `pthread_exit` subroutine.
– Being cancelled by another thread via the `pthread_cancel` routine.
– Terminating the entire process to which the thread belongs, by calling the `exec` or `exit` subroutines.

Several other operations regarding thread management are supported by the Pthreads specification, that include: passing arguments to threads, joining and detaching threads and managing the stack.

3.1.2 Mutexes

"mutex" is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes. Mutex variables are some of the primary means of implementing thread synchronization and of protecting shared data when multiple writes occur.

A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks it. Mutexes can be used to prevent "race" conditions, which would exist if locks were absent.

A typical sequence in the use of a mutex is as follows:
– Create and initialize a mutex variable.
– Several threads attempt to lock the mutex.
– Only one succeeds and that thread owns the mutex.
– The owner thread performs some set of actions.
– The owner unlocks the mutex.
– Another thread acquires the mutex and repeats the process.
– Finally the mutex is destroyed.

Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used. There are two ways to initialize a mutex variable:

1. Statically, when it is declared, for example:
   `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
2. Dynamically, with the `pthread_mutex_init()` routine. This method permits setting mutex object attributes, *attr*.

The *attr* object is used to establish properties for the mutex object, and must be of type `pthread_mutexattr_t` if used (may be specified as `NULL` to accept defaults). The Pthreads standard defines some optional mutex attributes:

Protocol: Specifies the protocol used to prevent priority inversions for a mutex. This determines how a thread behaves in terms of priority when a higher-priority thread wants the mutex.

Prioceiling: Specifies the priority ceiling of a mutex, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion (a problematic scenario in scheduling when a higher priority task is indirectly preempted by a lower priority task), the priority ceiling of the mutex shall be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex.

Process-shared: Specifies the process sharing of a mutex. This attribute can be set to `PTHREAD_PROCESS_SHARED` to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE` (which is the default value), the mutex shall only be operated upon by threads created within the same process as the thread that initialized the mutex.

Type: The mutex type attribute is used to create mutexes with different behaviors. Depending on this type there can be a classification of mutex types:

A *normal mutex* (`PTHREAD_MUTEX_NORMAL`) cannot be locked repeatedly by the owner. Attempts by a thread to relock an already held mutex, or to lock a mutex that was held by another thread when that thread has terminated result in a deadlock condition. Attempting to unlock an unlocked mutex results in undefined behavior.

A *recursive mutex* (`PTHREAD_MUTEX_RECURSIVE` or `PTHREAD_MUTEX_RECURSIVE_NP` depending on the platform) can be locked repeatedly by the owner, maintaining a concept of a lock count. The relocking deadlock which can occur with mutexes of type `PTHREAD_MUTEX_NORMAL` cannot occur with this type of mutex. Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can acquire the mutex. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

An *errorcheck mutex* (`PTHREAD_MUTEX_ERRORCHECK`). A thread attempting to relock this mutex without first unlocking it shall return with an error. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

The code in Figure 3.1 illustrates an initialization of a recursive mutex.

```
1  pthread_mutex_t * mutex;
2  pthread_mutexattr_t mutexAttribute;
3
4  mutex = (pthread_mutex_t * ) malloc(sizeof(pthread_mutex_t));
5
6  int status = pthread_mutexattr_init (&mutexAttribute);
7  if (status != 0) { return 1; }
8
9  status = pthread_mutexattr_settype(&mutexAttribute, PTHREAD_MUTEX_RECURSIVE_NP);
10 if (status != 0) { return 2; }
11
12 status = pthread_mutex_init(mutex, &mutexAttribute);
```

Figure 3.1: Lock initialization.

Robustness: This attribute illustrates what happens when a thread acquires a mutex and the original owner dies while possessing it. Valid values for robust include:

`PTHREAD_MUTEX_STALLED` (default): No special actions are taken if the owner of the mutex is terminated while holding the mutex lock. This can lead to deadlocks if no other thread can unlock the mutex.

`PTHREAD_MUTEX_ROBUST`: If the process containing the owning thread of a robust mutex, or the owning thread itself terminates while holding the mutex lock, the next thread that acquires the mutex shall be notified about the termination by the return value [`EOWNERDEAD`] from the locking function. The notified thread can then attempt to mark the state protected by the mutex

as consistent again by a call to `pthread_mutex_consistent()`, and the mutex can be used normally by other threads.

Mutex objects can be destroyed by calling the function `pthread_mutex_destroy`. This way the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using `pthread_mutex_init()`. In the same way the `pthread_mutexattr_destroy()` function shall destroy a mutex attribute object.

Once a mutex object has been created and initialized, it can be locked by calling `pthread_mutex_lock`. If the mutex is already locked, the calling thread will block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

The `pthread_mutex_trylock()` function is equivalent to `pthread_mutex_lock()`, except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex is recursive and currently owned by the calling thread, the mutex lock count shall be incremented by one and the `pthread_mutex_trylock()` function shall immediately return success.

The `pthread_mutex_unlock()` function releases a mutex object. If there are threads blocked on the mutex that is being unlocked by `pthread_mutex_unlock()`, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

3.1.3 Condition Variables

Condition variables provide another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data. Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.

Condition variables must be declared with a type `pthread_cond_t` and are initialized in a similar way with mutexes:

1. Statically, when they are declared:
   ```
   pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
   ```
2. Dynamically, with the `pthread_cond_init()` subroutine.

The `pthread_condattr_init()` and `pthread_condattr_destroy()` routines are used to create and destroy condition variable attribute objects and `pthread_cond_destroy()` should be used to free a condition variable that is no longer needed.

`pthread_cond_wait()` takes references to a condition variable and a mutex as parameters, and blocks the calling thread until the specified condition is signalled. This routine should be called while the mutex is locked, and it will automatically release the mutex while it waits. After the signal is received and the thread is awakened, the mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking the mutex when the thread is finished with it.

The `pthread_cond_signal()` routine takes a reference to a condition variable as parameter, and is used to signal (or wake up) another thread which is waiting on the condition variable, using a mutex `m`. It should be called after mutex `m` is locked, and must unlock the mutex in order for `pthread_cond_wait()` routine to complete. The `pthread_cond_broadcast()` routine should be used instead of `pthread_cond_signal()` if more than one thread is in a blocking wait state.

3.1.4 Synchronization Structures

Finally some useful synchronization primitives that are not analyzed in this thesis are reader/writer locks and barriers.

Reader/Writer locks are useful when mutual exclusion is unnecessarily restrictive, for example when two threads are only interested in reading a shared resource. It should then be possible to allow both to access the resource at the same time. Therefore, granting a read lock suffices when only read operations are involved. On the contrary, write locks are granted when an update is about to be performed on the data that is being protected.

Barriers are used to synchronize all running threads at a particular location within the code. This is most useful when several threads execute the same piece of code in parallel. The threads are made to wait at the barrier until all threads reach it. Then all the threads are allowed to continue.

3.2 Analyzing the C Programming Language

The C programming language is well-known for its flexibility in dealing with low-level constructs. Unfortunately, it is also well-known for being difficult to understand and analyze, both by humans and by automated tools. This necessitated the use of advanced and well-tested tools to perform parts of the static analysis. The tools employed are CIL [41], for parsing C and transforming it to an intermediate language, and the symbolic execution module from RELAY [57], a static race detection tool.

3.2.1 C Intermediate Language

CIL (C Intermediate Language) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. CIL is both lower-level than abstract-syntax trees, by clarifying ambiguous constructs and removing redundant ones, and also higher-level than typical intermediate languages designed for compilation, by maintaining types and a close relationship with the source program. The main advantage of CIL is that it compiles all valid C programs into a few core constructs with a very clean semantics. Also CIL has a syntax-directed type system that makes it easy to analyze and manipulate C programs. Its front-end is able to process not only ANSI-C programs but also those using GNU C extensions, which are quite common amongst the code base on which we tested out tool.

CIL is a highly-structured, “clean” subset of C, featuring a reduced number of syntactic and conceptual forms. For example, all looping constructs are reduced to a single form, all function bodies are given explicit return statements and syntactic sugar like “`->`” is eliminated, which reduces considerably the number of cases that have to be taken into account when performing a transformation. CIL also separates type declarations from code and flattens scopes within function bodies, which makes the program amendable to easy analysis and transformation. In addition, a really handy feature is its ability to compute the types of all program expressions and make casts explicit. At the same time CIL remains close to C, so that it is fairly easy to inject and remove code statements into and from CIL’s representation, and then transform back into C.

The CIL project comes along with a fair amount of frameworks and modules that perform various commonly used analyses:

Control Flow Analysis: CIL provides both high-level program structure and low-level control-flow information. The program structure is captured by a recursive structure of statements, with every statement annotated with successor and predecessor control-flow information. This way the

program can be treated both as an Abstract Syntax Tree, when control-flow is not important, and as a Control Flow Graph (CFG), to assist analyses that need this kind of information, for example dataflow analysis.

Dataflow analysis: is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's CFG, that has been computed in advance, is used to determine those parts of a program to which a particular value assigned to a variable might propagate. A simple way to perform dataflow analysis of programs is to set up dataflow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fix-point. In essence, dataflow analysis is a means of deriving information about the dynamic behaviour of a program by only examining the static code.

CIL's dataflow module contains a parameterized framework for forward and backward dataflow analyses. The only things that needs to be provided are the initialization and transfer functions and this module performs the analysis.

Points-to Analysis: is a static analysis technique that establishes which pointers, or heap references, can point to which variables or storage locations. In our tool, pointer analysis is a necessity as *mutable references* to locks are common in real-world C programs and it is crucial to determine the set of storage locations referenced by a pointer. Several algorithms have been suggested to solve this issue and picking the most suitable one is not a trivial task [33]. Some of these approaches are more precise and others are more efficient. The main factors that affect the cost/precision trade-off are:

- Flow-sensitivity: whether or not control-flow information is used during the analysis. Not using this information results in a less precise analysis, which was unacceptable for the needs of our tool.
- Context-sensitivity: whether or not the calling context is considered when analyzing a function. The approach followed in our analysis, as will be shown later on, evades the need for a context-sensitive analysis (which is more expensive than a context-insensitive one).
- Heap modelling: whether objects are named by allocation site or a more sophisticated shape analysis is performed.
- Field-sensitivity: whether components are distinguished or collapse into a single object. In our case we definitely need this feature, as mutex pointers are usually fields of structures.

CIL's pointer analysis module did not meet our needs, as it is not flow-sensitive and has a particularly weak heap model, so a more precise and sophisticated analysis was required. This was provided by RELAY. However, our tool employs CIL's (per file) *function pointer* analysis, that provides alias information for function pointers.

3.2.2 RELAY

RELAY [57] is a static race detection tool for C, based on the CIL framework. Its key feature is that it can scale to sufficiently big input programs, by analyzing each function separately and gathering function information in isolation to compute summaries. These summaries capture the behaviour of every function and they can be used at any calling context, as they are expressed in terms of the function parameters and global variables.

The intuition described above leads to a *bottom-up* context insensitive analysis over the *call graph*. A call graph is a directed graph that represents calling relationships between subroutines in a computer

program. Specifically, each node represents a function and each edge {f,g} indicates that function f calls function g. Thus, a cycle in the graph indicates recursive function calls.

An important advantage of RELAY is that it can create a single call graph from an entire project, which can comprise several source files and provide a unique identifier to every function in the call graph. In the end of the analysis and after the annotation of each function, the processed code will be dumped back into separate files, as it was originally.

The reason however for employing RELAY in our tool is its symbolic execution module, which in essence is a flow-sensitive intra-procedural pointer analysis. RELAY's symbolic execution analysis keeps track of the values contained in memory locations in terms of the incoming values of the formals and the globals. The analysis is fairly standard: it relies on the standard dataflow technique (provided by CIL), which propagates pointer information (state) to reachable program points. RELAY defines the transfer functions for handling statements, instructions, etc. This procedure is run before every function is analyzed by our tool and its results can be used afterwards when needed.

RELAY also provides a handful of routines that implement various operations on CIL's representation for lvalues and expressions. However, it also has some drawbacks:

- RELAY ignores reads and writes that happen inside assembly code segments, and does not handle corner cases of pointer arithmetic correctly.
- RELAY completely lacks a heap model: in our tool we offer some support for dynamically allocated mutexes, which had to be patched in RELAY's sources.
- RELAY is conservative about the effect that other threads may have on the state of variables, so it computes the set of locations that may escape the current thread and maps these locations to *any possible value* after each invocation of the symbolic flow function. We have loosened this restriction in our analysis by taking into account all memory mutations and by treating accesses to shared variables containing or pointing to locks as read-only.

3.3 Code base and Statistics

In order to obtain a rough estimate on what the common practice is in the use of lock primitives in real-world C language projects, we gathered a code base and extracted statistics regarding the use of Pthread mutexes. Unfortunately, we could only analyze source code that is freely distributed, which excluded a large portion of code used in widely known operating systems and applications that is proprietary software. The code base we gathered was hosted on two of the major online project hosting platforms: *github* [4] and *Google Code* [5], We have analyzed more than a hundred projects. Among these projects there are several widely used ones, such as: *Valgrind* [12], *Memcached* [7], *MooseFS* [8], *cmus* [2] and *The FreeRADIUS server* [11].

The features that we searched for were mainly:

- the type of memory allocation used for the Pthread mutexes,
- the type of structures in which they are located, and
- the locking patterns that can be found.

With the aid of the tools described in Section 3.2 we managed to parse, analyze and extract valuable statistics from our code base. Unfortunately, some projects depended on libraries that were not present on the system where the analysis was performed. This led to failures during pre-processing (which is necessary, as CIL operates on pre-processed files). These failures account for approximately 15% of the files that were pre-processed. Another 16.5% of the files that passed the pre-process phase encountered errors during the parsing from *frontc*, which is the C front-end that CIL uses. Finally, a 6% of the remaining "correct" files were met by errors during CIL compilation into AST's. Even with

these failures, however, 66% of the files passed our statistics analysis, which allows us to draw safe conclusions about the use of locks in C/Pthreads programs.

In the following paragraphs we comment on how our statistics were gathered and what conclusions we were able to gather.

3.3.1 Memory Management

In this part of our survey, we examine how mutexes are allocated in the program's memory. There are three distinct ways to allocate memory for objects in C:

Static memory allocation: space for these objects is provided in the binary at compile-time and their lifetime extends as long as the binary which contains them is loaded into memory. Static variables can be global or local.

Variables declared as `static` at the top level of a source file (outside any function definitions) are only visible throughout that file. However, CIL operates on each file separately, so it treats these variables as globals.

Program functions can declare static data locally as well, such that these data are inaccessible in other modules unless references to them are passed as parameters or returned. A single copy of static data is retained and accessible through many calls to the function in which they is declared. In its essence, this behaviour is similar to the behaviour a locally scoped global variable would have. In cases where default mutex attributes are appropriate, the macro `PTHREAD_MUTEX_INITIALIZER` can be used to initialize mutexes that are statically allocated. CIL tends to flatten scopes, in an effort to facilitate analysis, and so it treats static locally scoped variables as globals, retaining, however, a static storage attribute. Cases of static memory allocation, both global and local, are quite frequent.

Automatic memory allocation: temporary objects can be stored on the stack, and this space is automatically freed and reusable after the block in which they are declared is exited.

There is really no good reason to use locally allocated mutexes to protect resources from other threads, as the space allocated for the mutex will only be available for the duration of the function call. However, we did find a few cases with locally allocated mutexes. This happened for three reasons:

1. The mutex is only a field of a struct that needed to be allocated on the stack for various reasons irrelevant to locking. The mutexes in these cases were not used but they were still allocated in memory. Such is the case of the code segment in Figure 3.2 at line 102.
2. The reference to the locally allocated mutex is only used as an argument to a `pthread_cond_wait` or a `pthread_cond_timedwait` function that require a mutex reference as an argument. These functions shall be called with the mutex locked by the current thread. Once called they release the mutex and cause the thread to block on the condition variable which is also an argument. `pthread_cond_timedwait` is equivalent to `pthread_cond_wait`, except that an error is returned if the absolute time specified by one of its arguments passes. `pthread_cond_timedwait()` shall nonetheless release and re-acquire the relevant mutex. This way it can be used as a sleep function for the current thread only. This is what happens in Figure 3.3: Function `DCondVar_TimedWait` called at line 740 calls `pthread_cond_timedwait` at line 96.
3. Finally, a not so widely used case is having a thread allocate a struct that contains a mutex locally, and then pass a reference to that struct as an argument to the `pthread_create()` function. The mutex will be allocated on the parent thread, so we expect child threads to

```
293 /* back-end definition */
294 typedef struct _backend {
295     int                 be_type;
296     struct addrinfo     addr;
297     int                 priority;
298     int                 to;
299     int                 conn_to;
300     struct addrinfo     ha_addr;
301     char                *url;
302     int                 redir_req;
303     SSL_CTX             *ctx;
304     pthread_mutex_t     mut;
305     int                 n_requests;
306     double              t_requests;
307     double              t_average;
308     int                 alive;
309     int                 resurrect;
310     int                 disabled;
311     struct _backend     *next;
312 }   BACKEND;
```

(a) **pound.h**

```
99  static void
100 be_prt(const int sock)
101 {
102   BACKEND be;
103   struct  sockaddr_storage    a, h;
104   int     n_be;
105
106   n_be = 0;
107   while(read(sock, (void *)&be,
108         sizeof(BACKEND)) ==
109       sizeof(BACKEND)) {
110     if(be.disabled < 0)
111       break;
112     read(sock, &a, be.addr.ai_addrlen);
113     be.addr.ai_addr =
114       (struct sockaddr *)&a;
115     if(be.ha_addr.ai_addrlen > 0) {
116       read(sock, &h,
117           be.ha_addr.ai_addrlen);
118       be.ha_addr.ai_addr =
119         (struct sockaddr *)&h;
120     }
121     if(xml_out)
122       printf( ... );
123     else
124       printf( ... );
125   }
126   return;
127 }
```

(b) **poundctl.c**

Figure 3.2: Memory management: Local allocation — Case 1: Code from The Pound program: a reverse proxy and load balancer.

join the parent thread before the mutex is deallocated from the stack. This is the case in Figure 3.4.

Dynamic memory allocation: blocks of memory of arbitrary size can be requested at runtime using library functions such as `malloc` from a region of memory called the heap; these blocks persist until subsequently freed for reuse by calling the library function `free`. Dynamic allocation of mutexes was quite common among the projects we examined.

From the sample that was analyzed we gathered the results shown in Figure 3.5 regarding memory allocation of mutexes. We have calculated the percentage of the projects from our code base that feature at least one occurrence of each type of memory allocation that we mentioned earlier.

3.3.2 Data Structures

As mentioned in Section 2.3.2, the various approaches to deadlock freedom have diverse behaviours with the different types of data structures used to store mutexes. Most of them behave well when mutexes are allocated as standalone variables, but only a few can handle recursive data structures: linear (such as arrays and lists), trees, hashes, graphs, etc. The reason for this is that static analysis approaches have difficulty in modelling complex structures containing mutexes, precisely and efficiently. Unfortunately, it is nearly impossible to determine the size and shape of these objects statically, and distinguish between the elements of these structures.

```
77  int DCondVar_TimedWait( DCondVar *self,
78      DMutex *mutex, double seconds )
79  {
80    long sec = floor( seconds );
81    long nsec =
82      (long)(( seconds - sec ) * 1E9);
83    struct timeval now;
84    struct timespec timeout;
85    int retc = 0;
86
87    gettimeofday(&now, NULL);
88    timeout.tv_sec = now.tv_sec + sec;
89    timeout.tv_nsec =
90      now.tv_usec * 1000 + nsec;
91    if( timeout.tv_nsec >= 1E9 ){
92      timeout.tv_sec ++;
93      timeout.tv_nsec -= 1E9;
94    }
95
96    retc = pthread_cond_timedwait(
97        & self->myCondVar,
98        & mutex->myMutex, & timeout );
99    return ( retc == ETIMEDOUT );
100 }
```

(a) **kernel/daoThread.c**

```
718 static void SYS_Sleep( DaoContext *ctx,
719     DValue *p[], int N )
720 {
721   DMutex    mutex;
722   DCondVar  condv;
723
724   double s = p[0]->v.f;
725   if(ctx->vmSpace->options & DAO_EXEC_SAFE){
726     DaoContext_RaiseException(ctx,DAO_ERROR,
727         "not permitted" );
728     return;
729   }
730   if( s < 0 ){
731     DaoContext_RaiseException( ctx,
732         DAO_WARNING_VALUE,
733         "expecting positive value" );
734     return;
735   }
736   /* sleep only the current thread: */
737   DMutex_Init( & mutex );
738   DCondVar_Init( & condv );
739   DMutex_Lock( & mutex );
740   DCondVar_TimedWait(&condv,&mutex,s);
741   DMutex_Unlock( & mutex );
742   DMutex_Destroy( & mutex );
743   DCondVar_Destroy( & condv );
744 }
```

(b) **kernel/daoStdlib.c**

Figure 3.3: Memory management: Local allocation — Case 2. Code from Dao Programming Language [3].

Therefore, it would be really valuable to have a rough estimate of how often mutexes are located in recursive structures. To do this in an automated way, we created a CIL module that visits every type definition in each project and searches for back-edges in the *type graph*, whose structure is shown in Figure 3.6. The type graph is visited starting from the outer type and moving towards inner types (in structs, arrays, etc.), while keeping track of the names of the types that have been visited, before they are unfolded. Cycles are usually detected in structs, when the *type* of one of the fields in the *field list* is the same with one of the types that had been visited. If one of the fields of the nodes that belong to this cycle is of type `pthread_mutex_t`, then this is indeed a case where mutexes are used in a recursive data structure.

An example of a recursive data struct that contains a mutex appears in MooseFS [8] and can be seen in Figure 3.7. The defined `struct _threc` is a list, in which each node's next element is signified by `struct _threc *next`, and each node contains a `pthread_mutex_t mutex`.

Another interesting case is the allocation of arrays of mutexes, or arrays of objects that contain mutexes. These structures are also really challenging to handle statically, as there is no easy way to calculate the size of the array or the value of an array index at a specific program point. In order to locate such cases, we looked for expressions where one of the following occurred:

- a lock operation on an expression of type `pthread_mutex_t`, that either has an array index offset, or is a binary operation (which means that is an offset from the head of an array).
- a memory allocation function that allocates memory of size of the form `N * sizeof(pthread_mutex_t)`.
- an assignment with right hand side in the same form as in the first case.

```
/* Erasure state */
struct erase_state {
    struct s3backer_store *s3b;
    s3b_block_t queue[MAX_QUEUE_LENGTH];
    u_int qlen;
    pthread_t threads[NUM_ERASURE_THREADS];
    int quiet;
    int stopping;
    uintmax_t count;
    pthread_mutex_t mutex;
    pthread_cond_t thread_wakeup;
    pthread_cond_t queue_not_full;
};

int s3backer_erase(struct s3b_config
                                *config)
{
  struct erase_state state;
  struct erase_state *const priv = &state;
  ...
  /* Initialize state */
  memset(priv, 0, sizeof(*priv));
  priv->quiet = config->quiet;
  if ((r = pthread_mutex_init
        (&priv->mutex, NULL)) != 0) {
    warnx("pthread_mutex_init: %s",
        strerror(r));
    goto fail0;
  ...
  for (i=0;i<NUM_ERASURE_THREADS;i++) {

    if ((r =
        pthread_create(&priv->threads[i],
        NULL,erase_thread_main, priv))!= 0)
      goto fail3;
  }
  ...
  /* Clean up */
fail3:
  pthread_mutex_lock(&priv->mutex);
  priv->stopping = 1;
  pthread_cond_broadcast(&priv->thread_wakeup);
  pthread_mutex_unlock(&priv->mutex);
  for (i=0;i<NUM_ERASURE_THREADS;i++) {
    if (priv->threads[i] == (pthread_t)0)
      continue;
    if ((r = pthread_join
        (priv->threads[i], NULL)) != 0)
      warnx("pthread_join: %s",
        strerror(r));
  }
  ...
  pthread_cond_destroy(&priv->queue_not_full);
fail2:
  pthread_cond_destroy(&priv->thread_wakeup);
fail1:
  pthread_mutex_destroy(&priv->mutex);
fail0:
  return ok ? 0 : -1;
}
```

Figure 3.4: Memory management: Local allocation — Case 3. Code from s3backer — FUSE-based single file backing store via Amazon S3 [10] (erase.c).

Memory type	Frequency
Global static	54.21%
Global Non-static	42.06%
Stack allocation	13.08%
Dynamically allocated	44.86%

Figure 3.5: Memory management of mutexes.

Running this module on our code base showed that approximately the 23% of the projects use locks in recursive structures and 58% use arrays of locks.

3.3.3 Locking Patterns

An interesting classification concerns the pattern in which lock-unlock pairs are encountered in the source code. It is important to make this distinction, as there are several approaches to deadlock freedom that support block-structured locking, but fail to treat cases of unstructured locking.

As far as this criterion is concerned the major classes in which locking patterns can be categorized are as follows:

Lexically scoped (block-structured): This category consists of programs in which the lock and the

$$
\begin{array}{rlll}
type ::= & Void & \mid Int(intKind) & \mid Float(floatKind) \\
\mid & Array(type,\ exp) & \mid Fun(type,\ variable\ list) & \mid Enum(enumInfo) \\
\mid & Named(string,\ type) & \mid Struct(compInfo) & \mid Union(compInfo) \\
\mid & Ptr(type) & &
\end{array}
$$

$$
\begin{aligned}
enumInfo &::= (string,\ item\ list) \\
compInfo &::= (string,\ field\ list) \\
field &::= (string,\ type)
\end{aligned}
$$

Figure 3.6: The Abstract syntax of CIL types.

```
45 typedef struct _threc {
46   pthread_t thid;
47   pthread_mutex_t mutex;
48   pthread_cond_t cond;
49   uint8_t *obuff;
50   uint32_t obuffsize;
51   uint32_t odataleng;
52   uint8_t *ibuff;
53   uint32_t ibuffsize;
54   uint32_t idataleng;
55   uint8_t sent;          // packet was sent
56   uint8_t status;        // receive status
57   uint8_t rcvd;          // packet was received
58   uint8_t waiting;       // thread is waiting for answer
59   uint32_t rcvd_cmd;
60   uint32_t packetid;     // thread number
61   struct _threc *next;
62 } threc;
```

Figure 3.7: Recursive data structure — Code from MooseFS [8] (mfsmount/mastercomm.c).

matching unlock operation of a single mutex belong to the same block of code and there are no unmatched lock operations between them. This is the simplest and the most common use of locking in C/Pthreads, and is most akin to Java-like synchronization blocks. An example is shown in Figure 3.8.

Stack based in the same function: In this type of locking both the lock and the matching unlock operation are performed in the same function, but not in the same code block. This pattern can be described as non-block-structured, however, in most cases it is easy to transform the program and the locking pattern into a block-structured one. An example for this is shown in Figure 3.9a and is taken from MooseFS [8], a distributed file system. It is clear that the programmer's intention is to unlock `aflock` right before returning from the function. Therefore, by imposing a single return point at the end of the function we only need to unlock `aflock` once at the end of the function, as shown in Figure 3.9b.

The term *stack based* can be justified if we imagine traversing the control flow graph of a function and for every lock operation we push the mutex that is being locked on a stack. In this locking pattern we expect to unlock the mutex that is currently the top of the stack on every unlock operation that might occur. This shall stand for every possible path we may be traversing in the control flow graph. The mutex which is being unlocked is subsequently popped off the stack.

```
168 static struct hostent *
169 gethostbyaddr_r(const char *addr, int len, int type,
170     struct hostent *result, char *buffer, int buflen, int *error)
171 {
172     struct hostent *hp;
173
174 #ifdef HAVE_PTHREAD_H
175     if (fr_hostbyaddr == 0) {
176         pthread_mutex_init(&fr_hostbyaddr_mutex, NULL);
177         fr_hostbyaddr = 1;
178     }
179     pthread_mutex_lock(&fr_hostbyaddr_mutex);
180 #endif
181
182     hp = gethostbyaddr(addr, len, type);
183     if ((!hp) || (hp->h_addrtype != AF_INET) || (hp->h_length != 4)) {
184          *error = h_errno;
185          hp = NULL;
186     } else {
187          copy_hostent(hp, result, buffer, buflen, error);
188          hp = result;
189     }
190
191 #ifdef HAVE_PTHREAD_H
192     pthread_mutex_unlock(&fr_hostbyaddr_mutex);
193 #endif
194
195     return hp;
196 }
```

Figure 3.8: Lexically scoped locking — Code from FreeRADIUS [11] (src/lib/getaddrinfo.c).

```
231 void fs_dec_acnt(uint32_t inode) {
232   aquired_file *afptr,**afpptr;
233   pthread_mutex_lock(&aflock);
234   afpptr = &afhead;
235
236   while ((afptr=*afpptr)) {
237     if (afptr->inode == inode) {
238
239       if (afptr->cnt<=1) {
240         *afpptr = afptr->next;
241         free(afptr);
242       } else {
243         afptr->cnt--;
244       }
245
246       pthread_mutex_unlock(&aflock);
247       return;
248     }
249     afpptr = &(afptr->next);
250   }
251
252   pthread_mutex_unlock(&aflock);
253 }
```

(a) **Original code**

```
231 void fs_dec_acnt(uint32_t inode) {
232   aquired_file *afptr,**afpptr;
233   pthread_mutex_lock(&aflock);
234   afpptr = &afhead;
235
236   while ((afptr=*afpptr)) {
237     if (afptr->inode == inode) {
238
239       if (afptr->cnt<=1) {
240         *afpptr = afptr->next;
241         free(afptr);
242       } else {
243         afptr->cnt--;
244       }
245       break;
246     }
247     afpptr = &(afptr->next);
248   }
249   pthread_mutex_unlock(&aflock);
250 }
```

(b) **Equivalent structured code**

Figure 3.9: Stackbased (same function) locking — Code from MooseFS v1.5 (mfsmount/mastercomm.c).

Stack based in different functions: It is not necessary that a pair of matching `pthread_mutex_lock` and `pthread_mutex_unlock` operations should reside in the same function. It is rather common, on the contrary, that these lock primitives are hidden within wrapper-functions. A reference to the mutex that is to be locked or unlocked is usually passed as a parameter to the wrapper-function. The result of this locking pattern would be equivalent to the one of the previous class, if the bodies of the wrapper functions were to be inlined in the context of the calling function. An example of this pattern is shown in the code segment in Figure 3.10, taken from Portland Doors [9], a free, open-source implementation of Sun Microsystems' Doors API, for inter-process communication.

```
622 static inline void lock_door_data
623   ( struct door_data* p )
624 /* Acquires a lock on a door_data
625  * structure, so that operations on it
626  * will be thread-safe and atomic.
627  */
628 {
629   assert( NULL != p );
630   if ( 0 !=
631       pthread_mutex_lock
632         (& p->lock_data))
633     fatal_system_error( __FILE__,
634         __LINE__, "Lock door_data" );
635
636   return;
637 }
638
639 static inline void
640   unlock_door_data
641     ( struct door_data* p )
642 /* Releases a lock on a door_data
643  * structure, so that other
644  * operations on it may proceed.
645  */
646 {
647   assert( NULL != p );
648   if ( 0 !=
649       pthread_mutex_unlock
650         (& p->lock_data))
651     fatal_system_error( __FILE__
652         , __LINE__,
653         "Unock door_data" );
654   return;
655 }
```

(a) **Wrapper functions**

```
666 static inline void
667   release_door_data(struct door_data* p)
668 {
669   assert( NULL != p );
670   lock_door_data(p);
671   assert( 0 <= p->pointers );
672   if ( 0 == p->pointers ) {
673     pthread_cond_destroy(& p->can_listen);
674     unlock_door_data(p);
675     pthread_mutex_destroy(& p->lock_data);
676     free(p);
677   }
678   else if ( ( ! p->revoked ) &&
679       ( 2 == p->pointers ) &&
680       ((DOOR_UNREF_MULTI & p->attr) ||
681         ((DOOR_UNREF & p->attr) &&
682           (!p->was_unref))
683       )
684     ) {
685     --p->pointers;
686     p->was_unref = true;
687     p->attr |= DOOR_IS_UNREF;
688     unlock_door_data(p);
689     invoke_unreferenced(p);
690   }
691   else {
692     --p->pointers;
693     unlock_door_data(p);
694   }
695
696   return;
697 }
```

(b) **Function**

Figure 3.10: Use of wrapper functions — Code from Portland Doors (door.c).

In this example, a `struct door_data` pointer `p` is passed as an argument at line 670 to the lock wrapper function `lock_door_data`, and the same happens with function `unlock_door_data` at lines 674 and 688. The wrapper functions are defined as `static inline`, which means that the compiler will probably substitute the code of the function in its caller body. However, this is not necessary, as not all compilers support this feature, and in terms of static analysis this case is treated as a regular function call. We can also notice that in this case, even if we inline the body of the wrapper functions in the caller's, then we still have a case of stack-based locking, and not lexically scoped locking. And this time it is not that easy to transform the locking pattern of function `release_door_data` to lexically scoped without changing the function's semantics, as there are different instructions following the unlock operations at lines 688 and 693 until the

return statement.

Unstructured locking: This is the least common case of locking patterns. Unstructured locking here is synonymous to non-stack-based locking in the sense that it is impossible to create an equivalent lexically scoped locking pattern. An example of this kind of locking is shown in Figure 3.11, and is taken from libactor [6], an actor model library for C.

```
330 #define ACCESS_ACTORS_BEGIN pthread_mutex_lock(&actors_mutex);
331 #define ACCESS_ACTORS_END pthread_mutex_unlock(&actors_mutex);
332 ...
333 actor_msg_t *actor_receive_timeout(long timeout) {
334   actor_state_t *st = NULL;
335   actor_msg_t *msg = NULL;
336   pthread_t thread = pthread_self();
337   struct timespec ts;
338   struct timeval tp;
339   memset(&ts, 0, sizeof(struct timespec));
340
341   ACCESS_ACTORS_BEGIN
342
343   ACTOR_THREAD_PRINT("actor_receive_msg()\n");
344   st = list_filter(actor_list, find_thread, (void*)PTHREAD_HANDLE(thread));
345   if(st != NULL) {
346     msg = list_pop((list_item_t**)&st->messages);
347
348     pthread_mutex_lock(&st->msg_mutex);
349
350     ACCESS_ACTORS_END
351
352       if(msg == NULL) { /* no messages available, let's wait */
353
354         if(timeout > 0) {
355           gettimeofday(&tp, NULL);
356           ts.tv_sec  = tp.tv_sec;
357           ts.tv_nsec = (tp.tv_usec * 1000) + (timeout * 1000000);
358           if(pthread_cond_timedwait(&st->msg_cond, &st->msg_mutex, &ts) == 0) {
359             msg = list_pop((list_item_t**)&st->messages);
360           }
361         } else {
362           pthread_cond_wait(&st->msg_cond, &st->msg_mutex);
363
364         }
365         msg = list_pop((list_item_t**)&st->messages);
366
367       }
368     pthread_mutex_unlock(&st->msg_mutex);
369   } else {
370     ACCESS_ACTORS_END
371   }
372
373   return msg;
374 }
```

Figure 3.11: Unstructured locking — Code from libactor [6] (src/actor.c).

In this example, the `actors_mutex` is acquired at line 341 and released at line 350. By the time of the release, `st->msg_mutex` has also been acquired at line 348, and is released at line 362, which is after `actors_mutex` is released. `pthread_cond_wait` includes an unlock followed by a lock operation. It is clear that there is no possible way of achieving the same result regarding lock operations as here by using synchronization blocks.

Locking type	Frequency
Lexically scoped	36.67%
Stack-Based (same function)	32.22%
Stack-Based (different function)	20.00%
Unstructured Locking	11.11%
Total	100.00%

Figure 3.12: Locking patterns.

In the Figure 3.12 we show what kinds of locking patterns are used by the projects from our code base. We assume that the four categories mentioned earlier are referred to in ascending order of complexity and that each project is classified according to the most complicated locking pattern it features. As we can see in the table, a fair amount of real-world projects feature non-lexically scoped locking, which justifies the effort of our approach, as some of the related work on our subject did not support it.

Chapter 4

Deadlock avoidance in C

In Chapter 2 we mentioned one of the latest contributions that used a type and effect system to achieve deadlock avoidance, made by G. Boudol [13]. We also elaborated on how Boudol's system fails to treat cases of unstructured locking patterns and cases where lock and the corresponding unlock operations reside in different functions (non lexically scoped locking). The type system proposed by P. Gerakios [31] *et al.*, which is the one underlying the implementation of our tool, supports these cases. This chapter focuses on presenting this system, not in its original form, but applied to our target language, which is the C programming language. But first we introduce some notation that is going to be useful for the rest of this thesis.

4.1 Definitions

Effect system: As stated by F. and H. R. Nielson [44], static analysis of programs comprises a broad collection of techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically during computation; this may be used to validate program transformations, to generate more efficient code or to increase the understanding of the software so as to demonstrate that the software satisfies its specification. One of the products of such techniques can be an effect system, namely, a formal system which describes the computational effects of computer programs, such as side effects. An effect system can be used to provide a compile time checking of the possible program behaviours. In our case, effects are tracked as sequences of lock and unlock operations.

Continuation effect: A *continuation effect* (γ) of an expression represents the effect of the function code following that expression. In our system, the continuation effect is computed intra-procedurally in terms of the function's formal parameters and global variables. Each lock operation and function call is annotated with its continuation effect, which is computed statically. For example, in the locking sequence in Figure 4.1, the lock operation at line 1 has continuation effect: $[l+, l-, m+, k-, m-]$, which means that after acquiring lock k in the first line, in the future l will be acquired, then released, and so on. Our system statically annotates lock operations and function calls with their continuation effect.

```
1 pthread_mutex_lock(k);
2 pthread_mutex_lock(l);
3 pthread_mutex_unlock(l);
4 pthread_mutex_lock(m);
5 pthread_mutex_unlock(k);
6 pthread_mutex_unlock(m);
```

Figure 4.1: A simple locking sequence.

Future Lockset: The future lockset of a lock operation is the set of the of all locations (mutexes) in the expression's continuation effect that are being locked until the matching unlock operation. Returning to the simple example of Figure 4.1 the future lockset for the first lock operation is therefore $\{l, m\}$. The computation of future locksets is deferred until runtime.

4.2 Technique

4.2.1 Main Idea

The intuition that guarantees deadlock avoidance is quite straightforward:

A lock operation succeeds only when both the lock and its future lockset are available.

This condition guarantees that granting a lock will not bring the program to an *unsafe* state. Indeed, the intuition behind this targets the fourth condition that must hold in order for a set of threads to reach a deadlock, namely the *circular wait*: "two or more threads form a circular chain, where each thread waits for a lock held by the next thread in the chain", combined with the second one (*hold and wait*): "threads already holding locks may request and wait for new locks".

It should be pointed out that we do not acquire all locks in the future lockset prematurely. If we did so, other threads that tried to acquire any of these locks would unavoidably have to wait until the matching unlock operation for the original mutex was executed. This would seriously damage the program's degree of parallelism.

4.2.2 Why the Idea Works

To provide the insight for the correctness of the above argument, lets imagine the simplest case of the circular wait deadlock condition, where only two "competitor" threads claim the same set of locks in a cyclic order. We note that if a partial order is imposed in the sequence that each thread requests locks, then the deadlock can be prevented, as all threads will have to wait as soon as the first thread acquires its first lock. The case of a simple deadlock prone condition is shown in Figure 4.2.

```
1 pthread_mutex_lock(x);
2 pthread_mutex_lock(y);
3 pthread_mutex_unlock(y);
4 pthread_mutex_unlock(x);
```

(a) **Thread A.**

```
1 pthread_mutex_lock(y);
2 pthread_mutex_lock(x);
3 pthread_mutex_unlock(x);
4 pthread_mutex_unlock(y);
```

(b) **Thread B.**

Figure 4.2: Simple deadlock prone code segment.

Lets assume, without loss of generality, that thread A is the first that is about to acquire lock x. Since all locks are available at the moment, it succeeds. Consequently, there are two possible executions:

1. Thread A requests lock y, before thread B continues execution. Thread A will succeed again as y is currently available. Thread B will now have to wait upon request of lock y, until both x and y have been released by thread A. Once this is done, thread B continues with its execution, and succeeds in all lock requests. In this case, the deadlock would have been prevented, even if no action was taken.

2. After thread A has acquired lock x, a *context switch* occurs, and thread B requests lock y. However y's future lockset, which contains x, is unavailable. Therefore, thread B will have to wait.

In a subsequent context switch, thread A will acquire y that thread B failed to acquire. The scenario is straightforward from now on, as thread A will release both locks and then thread B will be able to acquire them. In essence, the execution of the code in Figure 4.2 was serialized.

The second possible execution interleaving would lead to a deadlock, due to circular wait, if thread B had not deferred the acquisition of lock y.

4.2.3 Future Lockset Computation

The future lockset computation algorithm for cases where the lock and the matching unlock operation reside in the same function, like the ones in Figure 4.2, is pretty straightforward. The algorithm takes as input a lock x, a continuation effect γ, assumes an empty future lockset and adds all $y+$ events of γ to the future lockset until the matching unlock operation for x is found. If x is locked several times before it is unlocked (equally as many times), then all lock operations are taken into account, until the corresponding unlock operation (in a stack based way) for the initial lock operation is found.

Lock and unlock operations, however, do not have to reside necessarily in the same function scope. As a matter of fact it is rather common that they reside in different scopes, for example when lock and unlock wrappers are used. In these cases the future lockset cannot be computed within the scope of a single function. Therefore, we need to keep a runtime state of the calling context, which will be represented as a *stack* of continuation effects.

At function calls, the callee function's effect is pushed on the stack right before the call and popped from it as soon as it returns. This way, if the matching unlock operation has not been found within the function's stack frame, then we iterate over the stack until the matching unlock operation is found.

It is also possible that the matching unlock operation resides in the scope of a function that is being called. In order to keep track of such operations, the callee function's effect is inlined in the caller's effect right at the point of the call (it is actually a summary of this effect that is inlined but this does not affect the main idea). This way, the future lockset computation algorithm will take the callee function's effect into account, while searching for the matching unlock operation.

```
1  pthread_mutex_t k;
2  pthread_mutex_t l;
3
4  int my_lock (pthread_mutex_t * p) {
5    return pthread_mutex_lock(p);
6  }
7
8  int my_unlock (pthread_mutex_t * p) {
9    return pthread_mutex_unlock(p);
10 }
11
12 void foo () {
13   my_lock( &k );
14   pthread_mutex_lock( &l );
15   ...
16   pthread_mutex_unlock( &l );
17   my_unlock( &k );
18 }
```

Figure 4.3: Lock operations in different scope.

The example in Figure 4.3 features both of these cases. Function `my_lock` is called at line 13 with `&k` as argument. Our runtime system computes its future lockset; the matching unlock operation is not

in `my_lock`'s scope, so our algorithm iterates over the stack and enters function `foo`'s stack frame. Mutex address `&l` is added to the future lockset and the unlock operation is finally found in the effect that has been inlined by the function call to `my_unlock` at line 17. We note that the inlined effect at every function call depends on the parameters that are passed as arguments to the function.

Why compute the future lockset at runtime. In Figure 4.3, `my_lock` is only called from one site, so there is a unique future lockset for the expression locked at line 5. However, the future lockset relies on the call path to the function that attempts to acquire a mutex. If there were more call sites, there would be as many possible future locksets for the lock operation within `my_lock`. This is the main reason why future locksets are computed dynamically. Computing all possible future locksets statically and annotating the original source with them would require a considerable amount of space and some extra time to figure out which future lockset is the correct one for each calling context. However, in cases where we have a small number of possible future locksets it would probably improve the tool's efficiency. At the moment this remains future work.

Conditional expressions. Our system provides support for conditional expressions in a semi restrictive way. It does not require that all branches have the exact same lock effect, but it does require that the aggregate number of lock and unlock operations on every mutex is the same in each branch (a branch with an aggregate count of lock operations on a mutex equal to zero is equivalent with a branch with no lock operations on the mutex). If γ_1, γ_2, ..., γ_n are the effects of the n branches of a conditional expression, $\gamma_1 \ ? \ \gamma_2 \ ? \ \cdots \ ? \ \gamma_n$ is the effect of the expression itself.

```
1  int condition = ...
2
3  if (condition) {
4    pthread_mutex_lock(x);
5    pthread_mutex_lock(z);
6    pthread_mutex_unlock(x);
7    pthread_mutex_unlock(z);
8    ...
9    pthread_mutex_lock(x);
10 }
11 else {
12   pthread_mutex_lock(y);
13   pthread_mutex_unlock(y);
14   pthread_mutex_lock(x);
15 }
```

Figure 4.4: Lock operations in conditional execution.

When our lockset computation algorithm encounters conditional branches, it checks each possible path separately and gathers all mutexes that are being locked until the matching unlock operation is found in each path.

For example lets assume the code segment in Figure 4.4. The number and the sequence of the lock operations vary substantially between the two branches. However, it is easy to see that the result at both branches is the locking of mutex reference `x`.

```
if (n) pthread_mutex_lock(l);
...
if (n) pthread_mutex_unlock(l);
```

Figure 4.5: Unsupported conditional execution.

If we allowed programs that featured branches with uneven unmatched lock or unlock operations, then we risk failing to compute the future lockset correctly. Therefore, programs that feature code segments like the one in Figure 4.5, which by no means can be considered faulty, are rejected by our tool. In this example, if we try to compute the future lockset for the lock in the first line we realize that there is a path in the program's control flow graph (if the second if-condition is false) that if followed will never lead to the matching unlock operation. This behavior is not allowed in our tool.

```
61  ...
62  while (1) {
63    pthread_mutex_lock(&mutexHaveNeighbor);
64    DBG(VRFY_1, "Checking neighbor status.");
65    if (! haveNeighbor) {
66      DBG(VRFY_1, "No neighbors, waiting...");
67      pthread_mutex_unlock(&mutexHaveNeighbor);
68      pthread_cond_wait(&condNeighbor, &mutexAdvrp);
69      pthread_mutex_unlock(&mutexAdvrp);
70    } else {
71      pthread_mutex_unlock(&mutexHaveNeighbor);
72    }
73    DBG(VRFY_1, "Found a neighbor.");
74    pthread_mutex_lock(&mutexAdvrp);
75    sleeptime = check_neighbors();
76    pthread_mutex_unlock(&mutexAdvrp);
77    ...
78  }
```

Figure 4.6: A possible programmer's error — Code from ADVRP (Academic Distance Vector Routing Protocol) [1] (verify.c).

This feature, however, can point out cases where a lock operation is missing, due to a programmer's mistake. For example in Figure 4.6, the mutex `mutexAdvrp` is used by the function `pthread_cond_wait` without being locked first. This leads to undefined behaviour according to Pthread's specifications. In this case, our tool will complain that the count in the first branch of the if-statement for `mutexAdvrp` does not match the count in the second, as in the first branch we have an aggregate of one unlock operation [1], whereas the second one has a zero aggregate. The program can be corrected by inserting a lock operation on `mutexAdvrp` between the `pthread_cond_wait` operation and the first branch.

4.2.4 Locking Algorithm

To apply the rule described in 4.2.1 in the C runtime context, we created a wrapper function for the lock acquire operation. Its functionality can be summarized in the following simple algorithm, when acquiring a lock for the first time:

1. Compute the future lockset.
2. Check if every location in the future lockset is available.
3. If every location is available then acquire the mutex tentatively, otherwise wait shortly and go back to step 2.
4. Check (again) if every location in the future lockset is available.
5. If everything is available, then return successfully, otherwise unlock the mutex and go back to step 2.

[1] In our system the `pthread_cond_wait` function, when executed successfully, has a similar effect to a sequence of an unlock and lock operation on the same mutex.

Neither step 2 nor step 4 have to be executed *atomically* (as a sequence of machine instructions that are to be executed sequentially without interruption). This means that a context switch can happen at any step of the algorithm, without depriving our system from its safety properties. This can be realized if we think again of a simple example like the one in Figure 4.2:

Lets suppose that the execution of thread A reaches first the instruction `pthread_mutex_lock(x)`, and so it starts running the aforementioned algorithm. If no context switch occurs until the end of the algorithm, then everything is done normally and in the end the mutex has been successfully acquired by thread A. However, lets assume that a context switch happens while the algorithm is executed:

- After the lockset has been computed (step 1): At this moment thread A has not acquired mutex `x` even tentatively. Thread B begins executing and tries to acquire mutex `y`. Lets suppose that this operation is executed successfully. When the execution of thread A resumes, after a subsequent context switch, the algorithm will check for mutex `y` that has been already computed to be in `x`'s future lockset, and the check will fail, as the lock is held by thread B. The execution will block at that point until thread B releases mutex `y`.
- After step 2: the context switching can happen after the first check on the future lockset succeeds and mutex `x` has been acquired tentatively by thread A. Thread B will then try to acquire mutex `y`, but will fail because `x` is held, and so after the following context switch mutex `x` will be acquired permanently by thread A.
- The context switch can even happen while a future lockset is being checked. In such cases, the result might be different than what was originally expected assuming that the checks were done atomically, but we still do not enter an unsafe state. This case is better illustrated in the example in Figure 4.7.

```
1 pthread_mutex_lock(x);
2 pthread_mutex_lock(y);
3 pthread_mutex_lock(m);
4 ...
5 pthread_mutex_unlock(y);
6 pthread_mutex_unlock(x);
7 pthread_mutex_lock(x);
8 pthread_mutex_unlock(m);
```

(a) **Thread A (Low priority).**

```
1 pthread_mutex_lock(y);
2 ...
3 pthread_mutex_unlock(y);
4 ...
5 pthread_mutex_lock(y);
6 ...
7 pthread_mutex_unlock(y);
```

(b) **Thread B (High priority).**

Figure 4.7: Deadlock prone code segment – Interesting case for non atomic operations.

In this example, we assume that thread B has higher priority than thread A, which can be interpreted as if the scheduler allocated greater time portions when thread B is executed than thread A. We also assume that thread A is the first to execute `pthread_mutex_lock(x)` (line 1), so it computes the future lockset for this mutex, which is {`y`, `m`}. The check on the future lockset is not done atomically, which means that a context switch might happen any time while we iterate over the elements of the future lockset list. Lets assume that this happens right after `y` has been checked and has been found to be available. The execution moves on to thread B, which successfully acquires mutex `y`, as it is currently available and has an empty future lockset. Before the context switches again, thread B releases `y` (line 3). Thread A then proceeds with the rest of the future lockset check, and acquires `x` tentatively. Then it shall start to check again the future lockset to ensure that nothing has changed. However, right after checking `y` (the first element in the future lockset), which is free, a context switch occurs, and thread B continues execution acquiring mutex `y` (line 5), since it is available at the moment. When thread A continues execution, it finally acquires mutex `x` successfully, even though at that time, mutex `y` (which belongs to its future lockset) was already held by thread B.

The scenario that we just described could not have happened if the checks were performed atomically, but it still does not impose a hazard to our system, namely there is still no threat for a deadlock. In order for a deadlock to occur, there has to be a circular dependency in the lock acquisition order. This means that thread B should lock mutex `x` while mutex `y` is held (lines 5-7), so that `x` would belong to `y`'s future lockset. In that case, however, thread B would not have acquired `y` at line 5, because it would have been held tentatively by thread A. Instead, it would have waited for thread A to release `x`, and then it would have attempted to acquire `y`.

Chapter 5

Implementation of the Tool

Our analysis is performed in two phases. The first one is a static analysis and comprises the parsing, effect analysis, annotation and compilation of the source code. The second phase consists of a runtime system which performs our deadlock avoidance algorithm dynamically.

Figure 5.1 describes our tool's workflow.

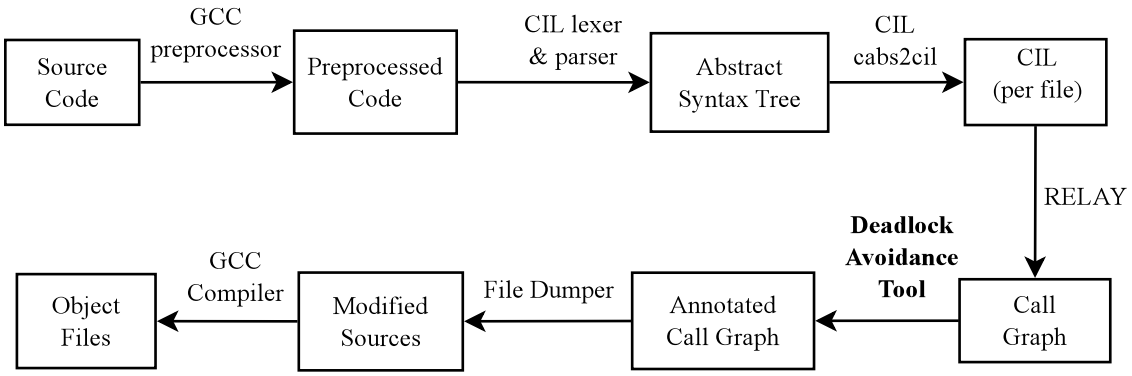

Figure 5.1: Our tool's workflow.

The first section of our analysis is the preprocessing of the C files using the GCC preprocessor. Then we use CIL to parse the files, create the abstract syntax tree and convert it into C Intermediate Language. This is done for each file individually. We then use RELAY to create a call graph for all files processed by CIL.

Before continuing with presenting the main features of our contribution, we will present some basic aspects of the file representation in CIL and the form of the inferred locking effects.

5.1 Abstract Syntax

In C Intermediate Language every variable and function has a unique integer identifier. These identifiers are produced per file, so when RELAY creates a single call graph for the whole (possibly multi-file) project, it gives a new unique identifier for each variable and function in the project.

Our tool also uses a module, taken from RELAY, that computes all the *Strongly Connected Components* (SCCs) based on the call graph. Knowing the strongly connected components and their function members is useful for our analysis, as functions of the same SCC indicate recursion, which requires special treatment.

From now on our analysis will run on the already computed call graph in a bottom-up manner, meaning that we will start processing first the function that does not call any other functions of the call graph,

or calls functions that belong to the same SCC. We do so because it is rather convenient to have all the information needed about a specific function (which is only possible if we already have analysed it) when we encounter a function call to it.

Before describing the actions taken in each part of our analysis, we are going to present a simplified version of the C Intermediate Language using the original Ocaml notation.

5.1.1 Program Parts

File. A *file* in CIL contains a list of *globals*, as shown in Figure 5.2.

```
(** Top-level representation of a C source file *)
type file =
    { mutable fileName: string;     (** The complete file name *)
      mutable globals: global list; (** List of globals as they will appear
                                        in the printed file *)
      ...
    }
...

(** global: The main type for representing global declarations and
 * definitions.*)
and global =
  | GType of typeinfo * location
    (** A typedef. All uses of type names (through the [TNamed] constructor)
        must be preceded in the file by a definition of the name. The string
        is the defined name and always not-empty. *)

  | GCompTag of compinfo * location
    (** Defines a struct/union tag with some fields. *)

  | GCompTagDecl of compinfo * location
    (** Declares a struct/union tag. *)
  ...
  | GVarDecl of varinfo * location
    (** A variable declaration (not a definition). If the variable has a
       function type then this is a prototype. There can be several
       declarations and at most one definition for a given variable. If both
       forms appear then they must share the same varinfo structure. *)

  | GVar  of varinfo * initinfo * location
    (** A variable definition. Can have an initializer. *)

  | GFun of fundec * location
     (** A function definition. *)
```

Figure 5.2: File and Globals in CIL.

The structures `compinfo` and `typ` were described in Figure 3.6. The structure `typeinfo` is just a named type.

The `varinfo` structure shown in Figure 5.3 contains information about each variable used in the program.

Function. A function definition is always introduced with a `GFun` constructor at the top level. All the information about the function is stored into a `fundec`, which is shown in Figure 5.4. Some of the information (e.g. its name, type, storage, attributes) is stored as a `varinfo`, which is a field of `fundec`.

```
(** Information about a variable. *)
varinfo = {
  mutable vname: string;      (** The name of the variable. *)

  mutable vtype: typ;         (** The declared type of the variable. *)
  ...
  mutable vstorage: storage;  (** The storage-class *)

  mutable vglob: bool;        (** True if this is a global variable*)
  ...
  mutable vdecl: location;    (** Location of variable declaration. *)

  mutable vid: int;           (** A unique integer identifier. *)
  ...
  mutable vaddrof: bool;      (** True if the address of this variable is taken. *)

  mutable vreferenced: bool;  (** True if this variable is ever referenced. *)
}
```

Figure 5.3: Variable information in CIL.

```
(** Function definitions. *)
fundec =
{ mutable svar:     varinfo;
     (** Holds the name and type as a variable, so we can refer to it
      * easily from the program. All references to this function either
      * in a function call or in a prototype must point to the same
      * [varinfo]. *)
  mutable sformals: varinfo list;
    (** Formals. These must be in the same order and with the same
     * information as the formal information in the type of the function.*)
  mutable slocals: varinfo list;
    (** Locals *)
  ...
  mutable sbody: block;          (** The function body. *)
  ...
  mutable sallstmts: stmt list;  (** After the control flow graph is computed
                                  * this field is set to contain all
                                  * statements in the function. *)
}
```

Figure 5.4: Function definitions in CIL.

The function definition contains, in addition to the body, a list of all the local variables and separately a list of the formals. Both kinds of variables can be referred to in the body of the function.

Statement. The function body (`sbody`) is essentially a list of all the statements [5.5] that belong to the function. Initially, the body of statements does not contain any control flow information. Each statement is identified uniquely and contains lists of all the predecessor (executed right before the current) and successor (executed right after the current) statements. These identifiers and lists are filled after we invoke CIL's control flow graph computation module. This procedure is based on the `kind` of the statement and the context in which the statement appears. The control flow graph is a directed graph whose nodes are the function's statements.

```
(** Statements. *)
stmt = {
    mutable skind: stmtkind;
    (** The kind of statement *)

    mutable sid: int;
    (** A number (>= 0) that is unique in a function. *)

    mutable succs: stmt list;
    (** The successor statements. They can always be computed from the skind
     * and the context in which this statement appears. *)

    mutable preds: stmt list;
    (** The inverse of the succs function. *)
  }
...
(** The various kinds of control-flow statements statements *)
and stmtkind =
  | Instr  of instr list
  (** A group of instructions that do not contain control flow. *)

  | Return of exp option * location
   (** The return statement. This is a leaf in the CFG. *)

  | Goto of stmt ref * location
   (** A goto statement. Appears from actual goto's in the code or from
    * goto's that have been inserted during elaboration. The reference
    * points to the statement that is the target of the Goto. *)

  | Break of location
   (** A break to the end of the nearest enclosing Loop or Switch *)

  | Continue of location
   (** A continue to the start of the nearest enclosing [Loop] *)

  | If of exp * block * block * location
   (** A conditional. Two successors, the "then" and the "else" branches.
    * Both branches fall-through to the successor of the If statement. *)

  | Switch of exp * block * (stmt list) * location
   (** A switch statement. The statements that implement the cases can be
    * reached through the provided list. For each such target you can find
    * among its labels what cases it implements. The statements that
    * implement the cases are somewhere within the provided [block]. *)

  | Loop of block * location * (stmt option) * (stmt option)
    (** A [while(1)] loop. The termination test is implemented in the body of
     * a loop using a [Break] statement. The first stmt option points to the
     * stmt containing the continue label for this loop and the second points
     * to the stmt containing the break label for this loop. *)
  ...
```

Figure 5.5: Statements in CIL.

Instruction. As soon as the control flow analysis is complete, we will only be interested in statements of the form `Instr of instr list`. The rest of the statement kinds are only used to compute the control flow graph, and since our analysis uses the graph directly these statements are no longer useful. There are three kinds of instructions, but since our tool does not support inline assembly code, we are only interested in two of them: assignments and function calls [5.6]. The locking operations belong to the latter.

```
(** Instructions. *)
instr =
    Set        of lval * exp * location
   (** An assignment. The type of the expression is guaranteed to be the same
    * with that of the lvalue *)
  | Call       of lval option * exp * exp list * location
   (** A function call with the (optional) result placed in an lval. It is
    * possible that the returned type of the function is not identical to
    * that of the lvalue. In that case a cast is printed. The type of the
    * actual arguments are identical to those of the declared formals. The
    * number of arguments is the same as that of the declared formals, except
    * for vararg functions. *)
  | ...
```

Figure 5.6: Instructions in CIL.

5.1.2 Values

Expressions and L-values. The argument of a lock or an unlock function is an expression [5.7]. Expressions in CIL are side-effect free. Values that have addresses programmatically accessible, namely lvalues, are described in Figure 5.8. These values can appear at the left side of an assignment or as an operand to the address-of operator.

An lvalue denotes the contents of a range of memory addresses. This range is denoted as a host object along with an offset within the object. The host object can be of two kinds: a local or global variable, or an object whose address is in a pointer expression. We distinguish the two cases so that we can tell quickly whether we are accessing some component of a variable directly or we are accessing a memory location through a pointer. To make it easy to tell what an lvalue means CIL represents lvalues as a host object and an offset. The host object (represented as `lhost`) can be a local or global variable or can be the object pointed-to by a pointer expression. The offset is a sequence of field or array index designators.

```
(** Expressions (Side-effect free)*)
and exp =
    Const      of constant              (** Constant *)
  | Lval       of lval                  (** Lvalue *)
  | SizeOf     of typ                   (** sizeof(<type>) *)
  | SizeOfE    of exp                   (** sizeof(<expression>) *)
  ...
  | UnOp       of unop * exp * typ
    (** Unary operation. Includes the type of the result. *)
  | BinOp      of binop * exp * exp * typ
    (** Binary operation. Includes the type of the result. The arithmetic
     * conversions are made explicit for the arguments. *)
  | CastE      of typ * exp
  | AddrOf     of lval
  | StartOf    of lval
    (** Conversion from an array to a pointer to the beginning of
     * the array. *)
```

Figure 5.7: Expressions in CIL.

Each lvalue has an offset part. Each offset can be applied to certain kinds of lvalues and its effect is that it advances the starting address of the lvalue and changes the denoted type, essentially focusing to some smaller lvalue that is contained in the original one. There are three kinds of offsets:

– *No offset*: Can be applied to any lvalue and does not change either the starting address or the type. This is used when the lval consists of just a host or as a terminator in a list of other kinds of offsets.

– *Field offset*: Can be applied only to an lvalue that denotes a structure or a union that contains the mentioned field. This advances the offset to the beginning of the mentioned field and changes the type to the type of the mentioned field.

– *Array index offset*: Can be applied only to an lvalue that denotes an array. This advances the starting address of the lval to the beginning of the mentioned array element and changes the denoted type to be the type of the array element.

```
(** An lvalue *)
and lval =
    lhost * offset

and lhost =
  | Var        of varinfo
    (** The host is a variable. *)

  | Mem        of exp
    (** The host is an object of type [T] when the expression has pointer
     * [TPtr(T)]. *)

and offset =
  | NoOffset
  | Field  of fieldinfo * offset
  | Index  of exp * offset
```

Figure 5.8: L-values in CIL.

CIL's representation for expressions and lvalues is much more detailed than what is actually required for the needs of our analysis, which only cares about the expressions that are passed as arguments to lock operations. That's why we created a new simplified type of expressions that is presented in Figure 5.9. These simplified expressions in essence account only for those expressions that are of the form: `Lval of lval` or `AddrOf of lval`. However, these are sufficient for our needs, since the remaining cases are either never found as locking arguments (e.g. `BinOp`) or can be converted to simplified expressions without a sacrifice in accuracy (e.g. `Cast`).

5.1.3 Abstract Locations

As we examined in Chapter 3, C programs use a variety of ways to store mutexes. We saw cases where mutexes were:

– globally scoped,
– referenced by a formal parameter, or
– dynamically allocated.

This section describes how our static analysis distinguishes the various types of memory allocation used for mutex variables. This information will later be encoded and used by our runtime system, in order to compute runtime addresses.

In terms of memory management, an expression can belong to one of the following three categories:

```
type vinfo = Cil.fundec * Cil.varinfo

(** Simplified Abstract Expression*)
type sexp =
  (** bottom expression: not assigned or any possible *)
  SBot
  (** Address of a simplified expression *)
  | SAddrOf   of  sexp
  (** Variable: vinfo holds the function where this
   * variable was defined or used. *)
  | SVar      of  (vinfo * Cil.offset)
  (** Access the memory location through the pointer sexp. *)
  | SDeref    of  (sexp * Cil.offset)
  (** Only used for compatibility with RELAY's pointer analysis *)
  | SAbsHost  of  (AT.ptaNode * Cil.offset)
  (** Instantiation of a variable with a list of possible
   * expressions at a specific location *)
  | SInst     of
      (vinfo * Cil.offset * sexp list * Cil.location)
```

Figure 5.9: Simplified abstract locations.

Globals. These are the global variables (static or not). Here we refer both to mutex handles (variables of type `pthread_mutex_t`) and to mutex references (variables of type `pthread_mutex_t *`). These variables are distinguished by their variable information (`varinfo`) provided by CIL.

Stack pointers. A quite common practice is to lock or unlock mutex references that are passed as arguments to a function. Our approach to this issue is to gather all stack references and create an enumeration of them, so that they can be distinguished by their index in this enumeration.

Dynamically allocated. One of the major contributions of our tool is its ability to keep track of heap allocated mutexes. Heap allocated addresses cannot be known statically and cannot be referenced like global variables. Therefore, we needed a representation able to distinguish between distinct dynamically allocated addresses.

Our heap model is based on the context (path) of the allocating function. This means that we distinguish between heap allocated locations based on the path that has to be followed to reach the specific allocating function (e.g. `malloc`). A path is a sequence of function call locations. It is obvious that there might be several paths leading to an allocation point, which necessitated the use of a call path, rather than a single location. These paths correspond to distinct abstract locations and should be distinguished both at compile-time and at runtime. The paths to the allocating sites are relative to the current context, which is reasonable if we consider our tool's bottom-up approach. These will be made clear in the example in Figure 5.10.

Function `foo` is where all the heap allocations occur. To this function's context, there is a single allocation site, the one in line 2. Function `bar` that calls `foo` at two distinct sites, in lines 6 and 9, will get two distinct heap locations. Our analysis encodes these abstract locations as $[Loc\ B, Loc\ A]$, $[Loc\ C, Loc\ A]$ respectively. Function `main` with its turn will have an effect with four lock operations, all on distinct mutex references.

Even though our analysis extends RELAY's pointer analysis with context-sensitive tracking of fresh heap locations, it fails to track heap allocation (for data structures containing or pointing to locks) for recursive data structures or recursive functions.

Finally, our tool supports at most one level of dereference, and in any case it is field-sensitive, meaning that the mutex can be a field of the base value. CIL can convert field information to byte offset, which is later used to get the relevant value from the base value.

```
1  pthread_mutex_t * foo() {
2    return malloc(sizeof(pthread_mutex_t)); //Loc A
3  }
4
5  bar(){
6    pthread_mutex_t * a = foo();  //Loc B
7    pthread_mutex_lock(a);
8
9    pthread_mutex_t * b = foo();  //Loc C
10   pthread_mutex_lock(b);
11 }
12
13 int main() {
14
15   bar();    //Loc D
16
17   bar();    //Loc E
18
19 }
```

Figure 5.10: Simple example of heap allocated mutexes.

5.2 Effect Description

One of the main purposes of our analysis is to compute the effect (γ) of every function. The effect is computed using the control flow graph that has already been calculated for each function. This graph may be quite complex, as it might feature nodes with multiple predecessors and successors, and backedges. However, we have kept the effect form rather simple, to assist the runtime system compute future locksets efficiently, without missing any essential information regarding lock references. Therefore, our effect has a linear form (list). Each element is an *atomic lock operation*, which can be one of the following:

Lock/unlock operation: This is a simple event of a lock or an unlock operation applied on a mutex. Our analysis keeps track of the operation's location and the expression on which it is performed.

Joint effect: This is used in cases where many alternate effects can occur, due to multiple possible paths of execution. A joint effect usually arises in cases where a node has multiple successors and more than one of the possible paths contain lock operations or calls to functions with non-empty effects. This element consists of a list of effects, which can have an arbitrary number of elements (can also be zero). We will use the notation $\gamma_1 \ ? \ \gamma_2 \ ? \ \cdots \ ? \ \gamma_n$ to represent the joint effect of the alternate effects $\gamma_1, \gamma_2, \ldots \gamma_n$.

Keeping effects with no elements in a joint effect list makes sense if we consider cases of *if*-branches such as the one in Figure 5.11. In order to compute `x`'s future lockset at the first line we shall take into account any possible value of `condition`. If it is true then the lockset is empty, but if it is false then the branch is not executed, and control flow reaches the point where `y` is locked, which should be added to the lockset. If we omitted the empty effect here, the future lockset that would be computed would only consider the first case (true condition) and ignore the lock operation on `y`. There are cases, however, where the empty effect could be omitted. These are discussed in the optimization section.

Call to a function: We have already stressed the tendency of using wrapper functions for lock or unlock operations. Function calls may be responsible for an implicit effect, as the callee function might have an effect itself. This effect affects the caller function, so it should be included in the caller's effect. However, instead of placing the complete effect of the callee function, we only

```
pthread_mutex_lock(x);
if (condition) {
  ...
  pthread_mutex_unlock(x);
}

pthread_mutex_lock(y);
pthread_mutex_unlock(y);
pthread_mutex_unlock(x);
```

Figure 5.11: Simple code segment that creates a joint effect.

place a *summary* of it at the position of the call. When the callee's effect summary is placed in the caller's effect, the former is substituted with an equivalent effect summary where all expressions are translated in terms of the caller function's context. We discuss effect *summaries* and effect substitutions in Section 5.2.

Our tool uses a standard points-to analysis to determine the targets of function pointers. In cases where it cannot determine a unique target for a call then the original effect is replaced by a joint effect consisting of several alternative branches, one for each alternative target function.

Loop effect: Effects flowing to a node from backedges must be equivalent (with respect to the lock counts) to the input effect of the node. This restriction allows us to soundly encode loop effects: a loop may have any number of lock or unlock operations provided that upon exit of it the counts of each lock match the counts before the loop was executed. This restriction must hold as computing statically the number of iterations is undecidable in the general case. If the loop effect (γ_{loop}) has no unmatched lock or unlock operations, then no matter how many times the loop is repeated, the loop's effect will be the same as if it was repeated only once or never.

It is also possible, however, that lock operations match between successive loop iterations, as shown in the example in Figure 5.12. Here, the loop's effect is $[l+, l-, k-, k+]$. Upon successive loop iterations we expect the last element of this list ($k+$) to match with the third ($k-$), and so its future lockset should include l.

```
pthread_mutex_lock(k);
while (condition) {
  pthread_mutex_lock(l);
  pthread_mutex_unlock(l);
  ...
  pthread_mutex_unlock(k);
  ...
  pthread_mutex_lock(k);
}

pthread_mutex_lock(m);
pthread_mutex_unlock(m);
pthread_mutex_unlock(k);
```

Figure 5.12: Simple code segment with a loop effect.

To treat these cases, the effect of the entire loop is:

$$(\,)\ ?\ (\gamma_{loop}\ ?\ ((\,)\ ?\ summary(\gamma_{loop})))$$

In general, if a lock operation $l+$ in γ_{loop} has its matching unlock operation within γ_{loop} but earlier in the effect, the future lockset for $l+$ should contain all mutexes that are locked in between. These locks might actually occur earlier than $l+$ in the effect, as in the example in Figure 5.12.

We also include the alternative of an empty effect for the last time that the loop is going to be executed. In this case the matching unlock operation for $k+$ is the last unlock operation on k. Therefore m should be included in its future lockset. Again, this case can be simplified under certain conditions, which are described in the optimization section.

Summary. Placing the substituted callee's effect at function calls may lead to an exponential growth in the caller's effect. This necessitated a sound conservative approximation of the real effect to be used instead.

The runtime system will never try to compute a future lockset for a lock operation that resides in the inlined effect of a called function or in the loop effect that is being repeated for the second time in the function's effect. So in such cases we only really care about the presence of lock operations included in the effect until the matching unlock operation. This allows us to create *summaries* of effects that have the same aggregate locking effect, but are usually much smaller in size. In summarized effects multiple lock/unlock operations on the same reference are redundant and are replaced by a single one.

To compute the summary of an effect γ, we split the initial effect in two components: γ_+, which contains the unmatched lock operations, and γ_-, which contains the unmatched unlock operations. If there are more than one unmatched operations they will appear as many times in the summary, so as to have the same impact as the effect itself. We then build a third component: γ_0, which contains a pair of $[x+, x-]$ for each lock x that is acquired at any time in γ, excluding the ones that are in γ_+. Finally we take $summary(\gamma) = \gamma_+ :: \gamma_0 :: \gamma_-$. Computing a summary for a joint effect requires that each path of the join operation features the same counts on each reference. For example, consider the effect:

$$\gamma = [x+, y+, y-] \ ? \ [z+, x+, z-]$$

Then the summary for this effect is:

$$summary(\gamma) = [x+, y+, y-, z+, z-]$$

Effect Substitution. At function calls we cannot just place the callee function's effect summary at the call site, as the effect might be expressed in terms of the function's *formal parameters*. Therefore, before placing the effect we substitute every occurrence of a formal parameter in the effect, with the relevant real parameter which is passed at the function call.

Special treatment is also needed for *dynamically allocated* locations that are being locked. As we mentioned in Section 5.1.3, calling a function that allocates a mutex from different call sites creates distinct mutexes for each call. Subsequently, calling a function that allocates and locks the allocated mutexes implies lock operations on distinct mutexes. Each occurrence of a dynamically allocated mutex in an effect should contain information about the path from the caller function to the one that allocated the mutex.

To make this clearer we return to the example in Figure 5.10. In order to express the effect of function `bar` we will represent the mutex allocated by `foo` called at line 6 as $(Loc\ B, Loc\ A)$, and the one created by the call at line 9 as $(Loc\ C, Loc\ A)$. The effect is therefore:

$$[(Loc\ B, Loc\ A)+, (Loc\ C, Loc\ A)+]$$

This effect will be different when it will be substituted at the two distinct call sites of `bar` at lines 15 and 17 of `main`, and it will be:

$$[(Loc\ D, Loc\ B, Loc\ A)+, (Loc\ D, Loc\ C, Loc\ A)+]$$

and

$$[(Loc\ E, Loc\ B, Loc\ A)+, (Loc\ E, Loc\ C, Loc\ A)+]$$

respectively.

A lock operation on a *local variable* would be impossible to be substituted in the caller's context, so we will have to omit it when placing the function's effect. This does not impose a threat during runtime as the specific lock wouldn't even have been created before the function is called. Finally, *global* mutexes are left untouched at substitution.

5.3 Function Analysis

Our analysis, which is run on each function in the call graph bottom-up, consists of the following actions:

- Compute the Control Flow Graph (CFG) for the function.
- Add external declarations to the original files for the functions of our runtime system.
- Symbolic execution to determine the pointer analysis state of each program point.
- Compute the effect for the whole function and a summary for this effect.
- Create the runtime effects and insert them in the original code.
- Instrument the original code to update the effect index at every lock operation and function call.

5.3.1 Symbolic Execution

Mutable references to mutexes are quite common in programs that use the Pthreads API. Therefore, it was necessary for our analysis to establish a correspondence between mutex references and variables or storage locations.

In our tool we employ the symbolic execution analysis of RELAY. This analysis keeps track of the values contained in memory locations in terms of the incoming values of the formals and the globals. In order to support the features mentioned in the preceding sections we had to make some fixes to RELAY's symbolic execution module that targeted the drawbacks discussed in Section 3.2.2.

RELAY has its own representation for expressions and lvalues, which are shown in Figure 5.13.

formals, globals	$x \in \mathbf{X}$
PTA reps	$p \in \mathbf{P}$
symbolic lvalues	$o \in \mathbf{O} ::= x \mid x.f \mid p.f \mid (*o).f$
symbolic values	$v \in \mathbf{V} ::= \top \mid \bot \mid i \mid init(o) \mid$
	$must(o) \mid may(os)$
symbolic map	$\sigma \in \mathbf{S} = \mathbf{O} \to \mathbf{V}$

Figure 5.13: Symbolic analysis domain.

Metavariable $x \in \mathbf{X}$ is used to denote formals and globals, and metavariable $p \in \mathbf{P}$ to denote representative nodes from the Steensgard flow-insensitive points-to-analysis [52]. Steensgard points-to-analysis is used to create representative nodes which will ensure termination in the symbolic execution.

The set **O** of symbolic lvalues denotes the locations that our symbolic execution analysis keeps track of, and these include formals, globals and field/pointer accesses through these. We use $os \in 2^{\mathbf{O}}$ to represent a set of lvalues. The set **V** of symbolic values denotes the values that the symbolic analysis computes, and these include: $\perp$ which means "not assigned yet"; $\top$, which means "any possible value"; i, which represents a constant integer; $init(o)$, which denotes the incoming value of lvalue o; $must(o)$, which represents a value that must point to lvalue o; and $may(os)$, which represents a value that may point to any of the lvalues in os. Finally, a symbolic execution map $\sigma \in \mathbf{S}$ is a function from symbolic lvalues to symbolic values.

The symbolic execution keeps track of a symbolic map at each program point, and this symbolic map is updated using flow functions. The flow function for a simple assignment $x := e$ evaluates e in the current map to a symbolic value, and then updates x in the map. For assignments through pointers, namely $*x := e$, the flow function evaluates x to a symbolic value v_1 and e to a symbolic value v_2. Which lvalues are updated in the store depends on the value v_1. For example, if v_1 is $must(o)$, then only o is updated to the value v_2. As another example, if v_1 is $may(os)$, then all the lvalues in os are updated to the value v_2. The remaining cases, not shown here, are very similar in nature.

After the symbolic execution is run, we can use the information that has been gathered by querying the pointer analysis module which symbolic values a specific expression at a specific program point references. In our tool we only accept answers that are $must(o)$. Accepting may or undefined expressions would make our analysis undecidable.

Extension to RELAY's symbolic execution. Our approach diverges from RELAY's in respect to the effect that other threads may have on the state of variables: RELAY considers the set of locations that may escape the current thread as possibly mutated by other threads. These locations are mapped to $\top$ (meaning "any possible value") after each invocation of the symbolic flow function. This approach to handling thread interaction is very conservative.

Our analysis on the contrary is only interested in mutex references, so it works on the assumption that other threads do not mutate references to shared mutexes. Our approach keeps track of all visible modifications on global mutexes and references that are passed as arguments to functions. For each function we create a modifications' effect similar to the one used for locking. The modifications that interest us are the ones that are valid at the return statements of the function, i.e. out of two consecutive modifications on the same reference, only the second one will be valid at the end of the two instructions. This way in the end of each function we create a *modifications' summary* that contains the visible modifications of the whole function. One can easily guess that in cases of conditional execution we only allow the same equivalent modifications to happen in each branch of execution.

The modifications' summary is stored and used whenever the specific function is called. Specifically, the modifications (assignments) are substituted in terms of the caller function (in a way similar to how we substituted the effects at function calls) and fed to RELAY's symbolic execution at the call point, as if the assignments were made at the specific point.

Heap locations in symbolic execution. RELAY features a very weak support for heap allocated locations. To resolve this issue we mapped every allocating point in the program (e.g. function call to `malloc`) to a global variable and inserted an assignment of the lvalue of the allocation call to the address of this global variable. For example, the assignment:

```
gl = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t));
```

where `gl` is a global mutex reference, would become:

```
gl = (pthread_mutex_t *) malloc(sizeof(pthread_mutex_t)); //Location A
gl = & __Allocated_Location_A;
```

From this point on, the symbolic execution module treats heap locations in a slightly different way than it does with global locations: we mentioned in Section 5.2 that "regular" global variables are left intact at effect substitutions. The same holds for assignment substitutions. However, when modification summaries contain visible assignments of allocated locations, the entries that contain dynamically allocated locations should be updated with the location of the function call for reasons similar as before. For example, let there be a function `foo` with a modification summary that contains an assignment of the form `gl = & __Allocated_Location_A`. Function `foo` is called at a Location B, so we have to substitute the modification summary to the specific context in order to feed it to the symbolic execution module. In this case, we create a *fresh* global variable that will correspond exactly to the call at the specific location. In our case the assignment would become `gl = & __Allocated_Location_A_Location_B`. We keep a mapping from all new allocation variables to the calling contexts to which they correspond.

5.3.2 Effect Computation

The effect for each function is computed by running a standard forward dataflow algorithm on the function's control flow graph. The dataflow framework is provided by CIL. In order to use it we had to define a state for each program point, a function that initializes the data for each node of the control flow graph and a function that combines the states of a predecessor and the current node and computes the new state of the latter. We will elaborate in each one of these parts separately:

State: The nodes of the control flow graph are CIL's statements. Each node is associated with an input, a current and an output effect. The *input* effect of a node is a list of all the effects flowing in from its predecessor nodes. These effects will he combined as alternate effects. For instance, a node associated with input effects $\gamma_1, \ldots, \gamma_n$ will have an aggregate input effect $\gamma_1 \ ? \ \cdots \ ? \ \gamma_n$, which denotes a choice between alternate effects $\gamma_1, \ldots, \gamma_n$. We distinguish from the input effect, the effect that flows in from *loop* nodes (backedges), which are treated differently than the rest of the nodes. The *current* effect is the effect stemming from the node itself and it can only be non-empty when the statement is of the form `Instr of instr list`, which means that it consists of a sequence of instructions where control flow falls through. This instruction list has a straight line effect. Finally, the *output* effect is computed by appending the current effect to the input effect and is propagated to a node's successors.

State initialization: During the initialization of the states we compute the effect for each of the statements separately. Then for every loop node (we are referring to the first statement to be found to participate in a loop), we insert a *loop* effect placeholder. This placeholder will be propagated to the successor nodes like the rest of the effect elements. After the dataflow algorithm terminates the placeholders are substituted by the loop effect.

Combine predecessors: This process consist of two actions: first, we compute the new state of the current node based on the old state and the state of the incoming flow. Then we compare this new state with the old one. If there are discrepancies in the two states then we assign the new state as the node's state and continue. Otherwise, we declare that no change has occurred. When the state all nodes is unchanged, then a fixed point is reached and the algorithm ends.

The comparison of two states is based on the comparison of their output effects, which comprises checking their elements one-by-one and making sure they correspond to the same location.

The new output effect of a node is created by appending the joint effect of the input nodes to the current effect of the node. The joint effect of the input nodes passes first through a very important optimization, a *common prefix* optimization on the alternate effects. We will talk about this optimization shortly. Furthermore, in the same way we keep an input effect from all

the predecessor nodes that do not form a cycle, we also keep a loop effect, which is a list of all the effects stemming from the nodes which are connected with the current one with a backedge.

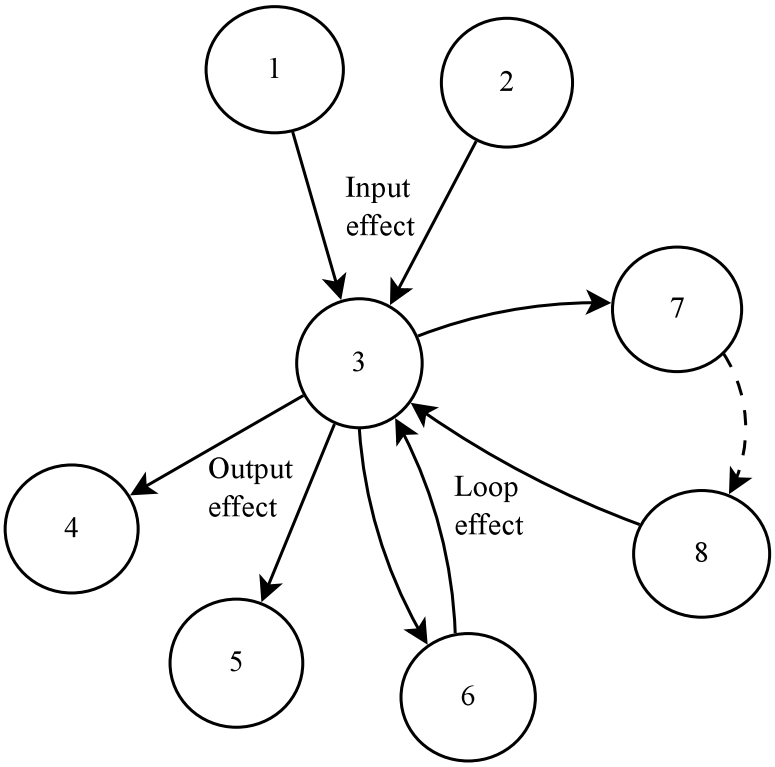

Figure 5.14: Example: effects at dataflow computation.

Figure 5.14 depicts part of a control flow graph. Node 3 is the current node. Edges $\{1, 3\}$ and $\{2, 3\}$ are front edges. The joint effect from those nodes constitutes the input effect. Edges $\{6, 3\}$ and $\{8, 3\}$ are backedges and so the effect stemming from them is a loop effect. Nodes 4, 5, 6 and 7 are successors of the current node, so the output effect of node 3 flows towards them.

Part of the effect that flows in from the backedges might have already flowed in from the front edges. In these cases we have to determine which part of the effect that flows in from the backedges is the sheer loop effect, which might be executed several times. This will be done during the backpatch (substitution) [5.3.4] part of the loop placeholders we inserted at the initialization of the nodes.

Finally, the overall function effect is computed by joining the output effects of nodes with no successors.

The result of the above procedure is the function's effect that still contains, however, placeholders which correspond to each node's loop effect. These placeholders are substituted by the relevant effect, keeping in mind the rules we mentioned in 5.2 about loop effects. This procedure will be described shortly, but first we discuss some effect optimizations.

5.3.3 Optimizations

While computing effects, several optimizations are performed so as to compact/elide effects and in general minimize the repetitions of the identical effect segments in a function's effect:

– One of the optimizations for compacting an effect computes the *common prefix and suffix* of the effects included in a join operator, to decrease the size of branches. This optimization can also compute a common prefix from part of the alternate effects in a join operator. The compare operator for atomic operations is based on the location where the operation is found in the source code, namely operations residing in the same location are considered the same operation. For example, suppose we have the effect:

$$[(A, B, C) \ ? \ (A, B, D) \ ? \ (E, F, G) \ ? \ (E, H, I) \ ? \ (J, K)]$$

(A, B, C), (A, B, D), etc. are sequences of atomic operations. Occurrences of the same symbol mean atomic operations in the same location. Our optimization in this case would manage to eliminate all multiple occurrences of the same operation, so the output would be:

$$[(A, B, [C \ ? \ D]) \ ? \ (E, [(F, G) \ ? \ (H, I)]) \ ? \ (J, K)]$$

This optimization is extremely useful as it reduces the size of the effects and speeds up the lock algorithm. In case of *call* effects this optimization is performed before the substitution of the callee's effect into the caller's.

– Another kind of optimization is to *flatten* effects that consist of nested join operators. For example, an effect of the form $[[\gamma_1 \ ? \ \gamma_2] \ ? \ [\gamma_3 \ ? \ \gamma_4]]$ can be reduced to $[\gamma_1 \ ? \ \gamma_2 \ ? \ \gamma_3 \ ? \ \gamma_4]$.

– In addition, multiple occurrences of the empty effect in alternate effects $\gamma_1 \cdots \gamma_n$ are substituted with a single empty effect. This optimization along with effect flattening are run alternately until a fixed point is reached in the size of the effect.

– During our effect processing we often perform a *cleaning* optimization, that cleans up remainders from the previous optimizations, such as join operators that only contain empty effects. There are also some cases (that we mentioned before) where effects of the form $[\gamma \ ? \ (\ )]$ can be optimized to γ. This holds when γ does not contain any unmatched lock or unlock operations. Intuitively, the empty effect in joint effect informs us that we have a branch that might not be executed. If there are unmatched unlock operations in γ, the one we are looking for could be among them and then our future lockset algorithm would stop at that point. However, we still ought to look in the rest of the effect for the matching unlock operation, as the branch might not be executed at all. If there were no unmatched operations in γ, the matching unlock operation would not be found in it anyway, so we would continue in the rest of the effect. Even though this optimization does not substantially decrease the effect size, it facilitates the application of the aforementioned optimizations.

– We have already mentioned that call effects are substituted by a *summary* of the effect corresponding to the function being called. In summarized effects multiple lock/unlock pairs for the same reference are redundant. This optimization greatly reduces the size of the final effect without compromising its precision.

Finally, an optimization that aims at reducing the compile time is that we only invoke the dataflow algorithm, which is CPU-intensive, for functions that are known to contain lock operations (by performing an in-advance linear search), to avoid additional overheads.

5.3.4 Effect Backpatch

At this stage effects still contain references to loop effects, which are stored in a mapping that maps node identifiers to loop effects. A simple case of a loop is shown in Figure 5.15.

In this example, node 2 is a loop node, so it inserts a placeholder in the effect, $[Loop\_2]$. Lets suppose that all nodes, except for nodes 2 and 4, contain lock operations. Effect γ_1 is the current effect of node 1 and so on. At the end of the dataflow algorithm, the entire effect will be:

$$\gamma_1, [Loop\_2], \gamma_3, \gamma_5$$

The loop effect at node 2 will be:

$$\gamma_3, \gamma_6$$

We observe that these two effects share a common part, γ_3. By comparing these two effects element by element, we can separate the effect that corresponds to a code segment that will be executed at least once (γ_3), from the part that will be executed conditionally (γ_6) and the rest of the function's effect (γ_5).

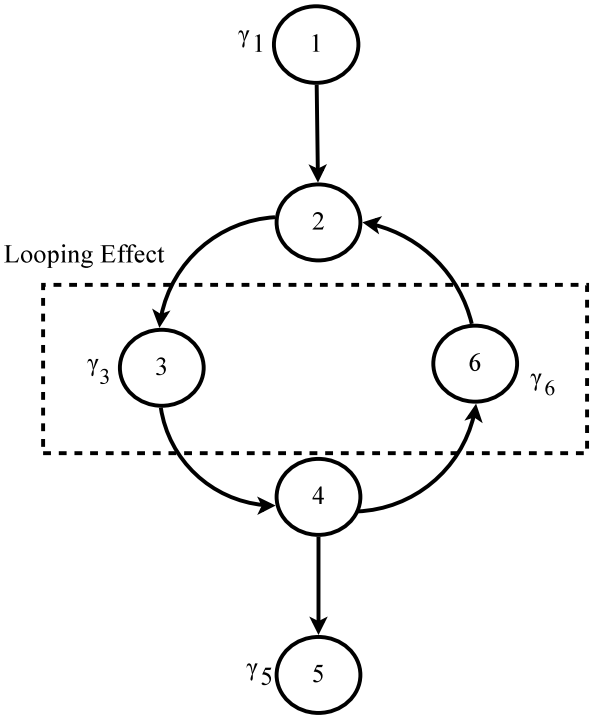

Figure 5.15: Simple loop example.

We stressed in Section 5.2 the necessity to repeat the loop effect twice for cases of lock operations that match between successive loop iterations. Therefore, the function's effect (γ_{fun}) will finally be:

$$\gamma_{fun} = \gamma_1, \gamma_3, [(\ ) \ ? \ (\gamma_{loop}, [(\ ) \ ? \ \sigma_{loop}])], \gamma_5$$

where

$$\gamma_{loop} = \gamma_6, \gamma_3$$
$$\sigma_{loop} = summary(\gamma_{loop})$$

It is easy to see that γ_3 will be repeated 3 times in the final effect: one as part of the function's main path effect, one as part of γ_{loop} to match lock operations between successive loop iterations, and finally in σ_{loop} to fix the lock counts in the loop. This greatly enlarges the overall effect. However, $[(\ ) \ ? \ \sigma_{loop}]$ part can be omitted if γ_{loop} does not contain any unmatched lock operations or it only features lock operations on a single lock.

5.4 Code Generation

After successfully completing the effect computation for the current function, our analysis proceeds with generating and injecting necessary code segments in the abstract syntax tree. CIL allows us to inject arbitrary code in various parts of the program. This code will be translated back into C with the rest of the program.

Our main goal was to provide the runtime system with a way to compute the future lockset for every lock operation and at the same time minimize the overhead induced by "effect accounting". A naive implementation of the informal description provided in earlier sections would simply allocate and initialize effect frames for each function call or lock operation, which would be unacceptable in terms of performance and required code space. Instead, the code generation phase statically creates a single block of initialization code for the effect of each function and inserts effect index update instructions (i.e., a single assignment) before each call and lock operation. Therefore, the overhead imposed for such operations is minimal.

5.4.1 Runtime Effect

The form of the runtime effect differs slightly from the effect we presented in the static analysis section, but it is still equivalent. The nodes of the runtime function effect are stored sequentially in an array. Each node is tagged with meta-info in order to emulate the tree-like form that our static effect has. There are four different types of nodes:

- A *SIMPLE* node, which is a node that represents a simple lock or unlock operation. This node contains all the details needed by the runtime system to calculate the address of the location that is locked or unlocked.
- A *SEQ* node, which is a node that represents a sequence of simple nodes. *SEQ* is used whenever there are more than one consecutive lock operations that should be treated as a block of atomic effects. This node contains the number of the simple straight-line operations and the offset in the array that they are placed.
- A *PATH* node, which is the equivalent of the join operator. This node contains the number of branches and the position in the array where the possible paths start. The possible effect paths are either single atomic operations (*SIMPLE*) or sequences of atomic operations (*SEQ*).
- A *CALL* node. This kind of node is used at function calls. Right before the function call, the effect index for the current stack frame is updated to point to this node, which is located in the array right after we have placed the callee function's effect.

Each function is also instrumented with instructions for pushing and popping effects from the runtime stack at function entry and exit points respectively. This imposes a constant overhead to function calls.

5.4.2 Runtime Lock Addresses

In Section 5.1.3 we explained how the abstract locations of *dynamically allocated* locks are distinguished by the call path in which they are allocated. Dynamically allocated locations cannot be known statically. Therefore, we use a mapping, implemented with a hashtable, that binds abstract locations to runtime addresses. Abstract locations (which are statically represented as lists of call sites) are encoded into integer values using a hashing function. An inverse mapping is also maintained for abstract heap locations. At memory allocation points we perform an insertion of the relevant abstract location representation and the allocated address to the hashtable, and when a deallocation operation is performed, the inverse mapping is searched using the physical address to be deallocated, and the binding between the abstract heap location and the physical address is removed from the heap mapping. In this way, our analysis is able to deal with locks that are dynamically deallocated and thereby avoid invalid accesses to deallocated locks.

Our tool handles *stack pointers* in a simpler way. We mentioned that static analysis collects all the stack based addresses that are used within a function and creates an enumeration on them. At runtime, every time the function is executed, an array is initialized that contains all necessary addresses, using the same indexing as in the static enumeration. All references to stack pointers from now on are made through this array.

The example in Figure 5.17 illustrates this. Function `foo` locks references `& s->a`, `& s->b` and `p`. The corresponding addresses will be inserted in array `__cil_tmp4` as soon as function `foo` is called, and the specific locations can be referenced within `foo`'s scope by their index in this array.

```
struct bar {
  pthread_mutex_t a;
  pthread_mutex_t b;
};

...

void *foo(struct bar * s,
    pthread_mutex_t * p) {
        pthread_mutex_lock( & s->a );
        pthread_mutex_lock( & s->b );
        pthread_mutex_lock( p );
}
```

(a) **Original version**

```
void *foo(struct bar *s, mypthread_mutex_t *p)
{ void *__retres3 ;
  mypthread_mutex_t *__cil_tmp4[3];
  stack_node_t __cil_tmp5 ;
  {
  __cil_tmp5.next = stack;
  __cil_tmp5.locals = __cil_tmp4;
  __cil_tmp5.current = __effect_48;
  stack = & __cil_tmp5;
  __cil_tmp4[0] =
      (mypthread_mutex_t *)(& s->a);
  __cil_tmp4[1] =
      (mypthread_mutex_t *)(& s->b);
  __cil_tmp4[2] =
      (mypthread_mutex_t *)p;

  ...
  }
}
```

(b) **Modified version**

Figure 5.17: Locking stack references.

A complication occurs when a stack pointer, pointing to a struct containing a lock, is passed casted to `void *`, as shown in Figure 5.18. In this case, our static analysis scans the function body to find an expression where the stack pointer is casted to its actual type. This happens in the assignment:

```
struct bar * tmp = (struct bar *) s;
```

This type information will be used to cast the relevant expression in case we need to dereference it. Therefore, in this case the stack pointer array will become:

```
cil_tmp4[0] = (mypthread_mutex_t *)(& ( struct bar *)s)->a);
cil_tmp4[1] = (mypthread_mutex_t *)(& ( struct bar *)s)->b);
```

During code generation, we apply an important optimization that attempts to minimize the size of the runtime effect stack, so that the future lockset calculation algorithm visits as few stack frames as possible. To achieve this we disabled code instrumentation for functions that do not directly perform any lock operations and do not contain calls to functions that will visit their effect frame at runtime.

```
struct bar {
  pthread_mutex_t a;
  pthread_mutex_t b;
};

...

void *foo(void * s) {
  ...

  struct bar * tmp = (struct bar *) s;
  pthread_mutex_lock( & tmp->a );
  pthread_mutex_lock( & tmp->b );
  ...
}
```

Figure 5.18: Stack pointer cast to void *.

5.5 Runtime System

The runtime system overrides the standard implementation of locking functions such as the Pthreads functions `pthread_mutex_lock` and `pthread_cond_wait`. If a lock is already held by the requesting thread then the lock's count is simply incremented (this occurs only when re-entrant locks are used). Otherwise, the runtime system performs two steps: it computes the future lockset of the requested lock and verifies that all locks in the future lockset are available when the lock is acquired.

The future lockset calculation algorithm uses the effect index inserted by the instrumentation phase to calculate the future lockset of the requested lock. When the matching unlock operation is not found in the function's effect, the algorithm iterates on the effects of the runtime stack. Locks held by the requesting thread are excluded from the lockset. Effect traversal is performed efficiently as effects are represented as arrays. Each atomic effect is represented by two machine words.

The next step is to acquire the lock provided that all locks in its future lockset are available. The strategy we implemented for dealing with unavailable locks employs *futexes* [30].

The futex system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address (while the addresses for the same memory in separate processes may not be equal, the kernel maps them internally so the same memory mapped in different locations will correspond for futex calls). A futex consists of a kernelspace wait queue that is attached to an aligned integer in userspace. Multiple processes or threads operate on the integer entirely in user space (using atomic operations to avoid interfering with one another), and only resort to relatively expensive system calls to request operations on the wait queue. A properly programmed futex-based lock will not use system calls except when the lock is contended.

Our locking algorithm was presented in Section 4.2.4. In Figure 5.19 we present the part of the runtime system which is the most crucial for the performance of our approach (the code segment that implements the algorithm of Section 4.2.4).

In this code segment, we assume that the future lockset has just been computed. To see how this code segments works, let's suppose that a thread A is trying to acquire a lock `x` with a future lockset containing a single lock `y`, which is currently held by thread B. Thread A will execute the algorithm and when checking lock `y` at line 741, it will set `local_val` to 1. Variable `wt` will be set to point to lock `y` (line 747) and thread A will yield. Before waiting for the futex on `y`, thread A will set `y.__data.__lock` to 2 and wait in line 779 until thread B releases `y`, and therefore sets `y.__data.__lock` to 0.

A similar scenario happens if thread A checks lock `y` and finds it available the first time, and then another thread B acquires it before thread A checks it again.

5.6 Current Limitations

Non C code. Our analysis can strictly handle the C language. Library code cannot be analyzed as it is not C code. We have assumed that by default library functions have an empty effect. The analysis cannot deal with non-local jumps (including signals) and inline assembly.

Pointer analysis. The off-the-shelf pointer analysis module fails when encountering programs with pointer arithmetic involving locks (including arrays) and recursive data structures that contain or point to locks. Even though our analysis extends the standard pointer analysis with context-sensitive tracking of fresh heap locations, it fails to track heap allocation (for data structures containing or pointing to locks) at recursive functions and loops. This limitation is dual to the aforementioned limitation

```
compute_lockset();

future_lockset_t *cu;
int lock_val, ret;
pthread_mutex_t *wt = NULL;

//Loop until successful
RETRY:
//Iterate over the future lockset and check availability
  for (cu = future_lockset; cu < gfl_iter; cu++) {
    if (cu->mutex == NULL) continue;

    // already owned by current thread
    if (cu->mutex->mutex.__data.__owner == gthread_pid) continue;

    //atomic check on the lock's availability
    lock_val = __sync_fetch_and_add(&cu->mutex->mutex.__data.__lock, 0);

    if (lock_val != 0) {
      //If the lock is held by another thread, wake one thread that is
      //waiting on the lock. Then start waiting on the futex address.
      if (wt != NULL) futex_wake(&wt->__data.__lock, 1);
      wt = (pthread_mutex_t *) cu->mutex;
      goto YIELD;
    }
  }

//Lock tentatively
  if (wt != NULL) { futex_wake(&wt->__data.__lock, 1); wt = NULL; }
  ret = pthread_mutex_lock((pthread_mutex_t *) lock);
  if (ret != 0) return ret;

//Check that versions did not change
  for (cu = future_lockset; cu < gfl_iter; cu++) {
    if (cu->mutex == NULL) continue;
    if (cu->mutex->mutex.__data.__owner == gthread_pid) continue;
    lock_val = __sync_fetch_and_add(&cu->mutex->mutex.__data.__lock, 0);
    if (lock_val != 0) {
      //Future lock not available - release lock
      pthread_mutex_unlock((pthread_mutex_t *) lock);
      if (wt != NULL) futex_wake(&wt->__data.__lock, 1);
      wt = (pthread_mutex_t *) cu->mutex;
      goto YIELD;
    }
  }
//Successful
  if (wt != NULL) { futex_wake(&wt->__data.__lock, 1); wt = NULL; }
  return 0;

YIELD:
  if (lock_val != 2) {
    __sync_bool_compare_and_swap(&wt->__data.__lock, 1, 2);
    lock_val = 2;
  }
  futex_wait(&wt->__data.__lock, lock_val);
  goto RETRY;
```

Figure 5.19: Runtime locking (runtime/runtime.c).

regarding unbounded data structures. In addition, expressions passed in lock functions must be assigned a unique abstract location. Finally, we require that lock pointers are mutated only before they are shared between threads, and that locks referenced with at least two levels of indirection (e.g., via double pointers) are not aliased at function calls.

Conditional execution. The analysis also currently rejects programs in which lock and their matching unlock operations are conditionally executed in distinct conditional statements having equivalent guards. For instance, the following program is rejected:

```
if (condition) pthread_mutex_lock(z);
if (condition) pthread_mutex_unlock(z);
```

Chapter 6

Performance Evaluation

In this section we describe our experimental results, aiming to demonstrate that our approach can achieve deadlock freedom with relatively low runtime overhead. The experiments were performed on a multiprocessor machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM. In the benchmarks involving network interaction, a second machine was also used with two Intel Xeon CPUs (2.80 GHz), having a total of 4 cores and 4 GB of RAM. Both machines were running Linux 2.6.26-2-amd64 and GCC 4.3.2.

The first two programs have as primary purpose to show that our approach avoids deadlocks, but also may result in better parallelism (dining philosophers).

bank transactions: a small multithreaded program simulating repeated concurrent circular transactions between two accounts that may deadlock. The program is based on the example we provided when addressing the deadlock in Section 2.3.1. The program creates two threads that perform a fixed number of transactions. Each transaction consists of a withdrawal step from account A and a deposit step to account B, each of which is protected by a lock. In addition, the whole transaction is protected by the lock account A; this creates a necessity for recursive (reentrant) locks and makes the program prone to deadlocks. The original program deadlocks with probability very close to 100%, and was deadlock-free in our modified version.

dining philosophers: a program implementing the deadlock-prone attempt to solve the classic multiprocess synchronization problem. The problem is summarized as five silent philosophers sitting at a circular table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. A large bowl of Spaghetti is placed in the center, which requires two forks to serve and to eat. A fork is placed in between each pair of adjacent philosophers, and each philosopher may only use the fork to his left and the fork to his right. However, the philosophers do not speak to each other. Each philosopher first picks up the stick on his left, then the stick on his right. In this setting a deadlock would arise if every philosopher held a left fork and waited perpetually for a right fork. Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a “cycle of unwarranted requests”. In this case philosopher P1 waits for the fork grabbed by philosopher P2 who is waiting for the fork of philosopher P3 and so forth, making a circular chain.

The original program deadlocks with a probability that decreases as the number of philosophers increases (for five philosophers, the probability for deadlock was roughly 70%) but increases again when the number of philosophers exceeds the number of available cores. For the performance comparison that we discuss below, we only used the deadlock-free runs of the original program.

The performance of the original and the instrumented versions are shown in Figure 6.1. For a given elapsed time (2 secs) we measured the total number of times that the philosophers ate (using a per-thread random number generator that was identical in both versions). It is interesting to see that in the

n	original	instrumented	improvement
5	126,452	127,361	0.72%
10	220,550	229,065	3.86%
15	292,080	286,841	− 1.79%
31	537,687	586,005	8.99%
63	1,082,773	1,194,669	10.33%
127	1,583,895	2,215,668	39.89%
255	1,472,359	4,175,600	183.60%

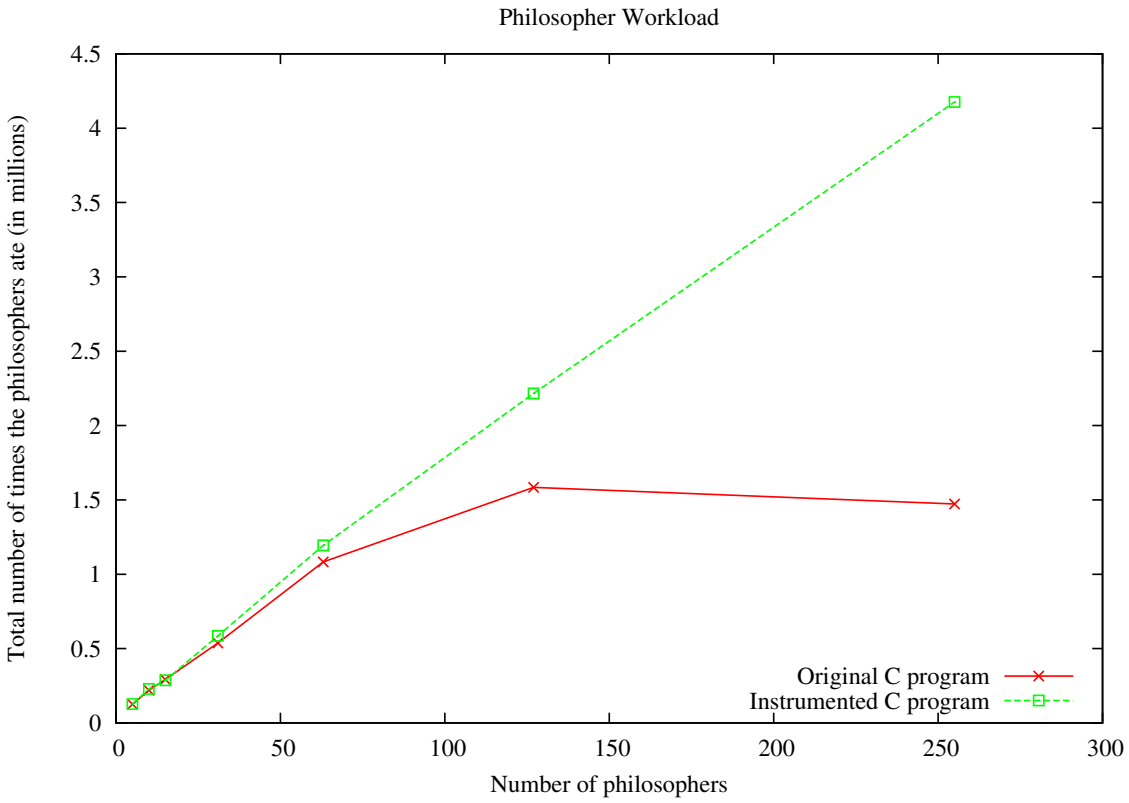

Figure 6.1: Performance comparison for the *dining philosophers*. We measure the total number of times that the n philosophers ate.

instrumented program, the number grows linearly with the number of philosophers, i.e., each philosopher eats for a (more or less) constant number of times during the 2 secs (this number is determined by the ratio of eating time versus sleeping time, which was chosen to be 0.1 in both programs). On the other hand, in the original program, the linear growth seems to last only as long as the number of philosophers is small and we do not run out of cores. In the original program, it is frequent that a philosopher holds his left stick while waiting to acquire his right stick. This is far less frequent in the instrumented program, which checks that the right stick is available before granting the left stick (if the right neighbour is fast enough, he can still get to it first but this is rather very improbable). This results in a much better degree of parallelism, which clearly shows in Figure 6.1.

We also used a total of six benchmark programs, which are real applications whose source code is publicly available. These benchmarks are used to evaluate scalability as the number of threads increases.

curlftpfs [22] and sshfs-fuse [51]: are file system clients that access hosts via the FTP and SSH network protocol respectively. Both applications create threads on demand so as to serve concurrent

n	original			instrumented			overhead
	U	S	e	U	S	e	
1	0.536	11.480	20.580	0.388	9.380	20.836	1.34%
2	0.520	9.928	22.182	0.512	9.002	22.832	2.93%
4	1.784	30.278	21.562	1.614	28.628	22.192	2.92%
8	2.762	52.366	21.416	3.174	54.334	21.340	− 0.35%
16	5.772	88.870	21.298	5.448	86.116	21.542	1.15%
32	6.042	98.912	22.660	6.316	103.434	22.678	0.08%

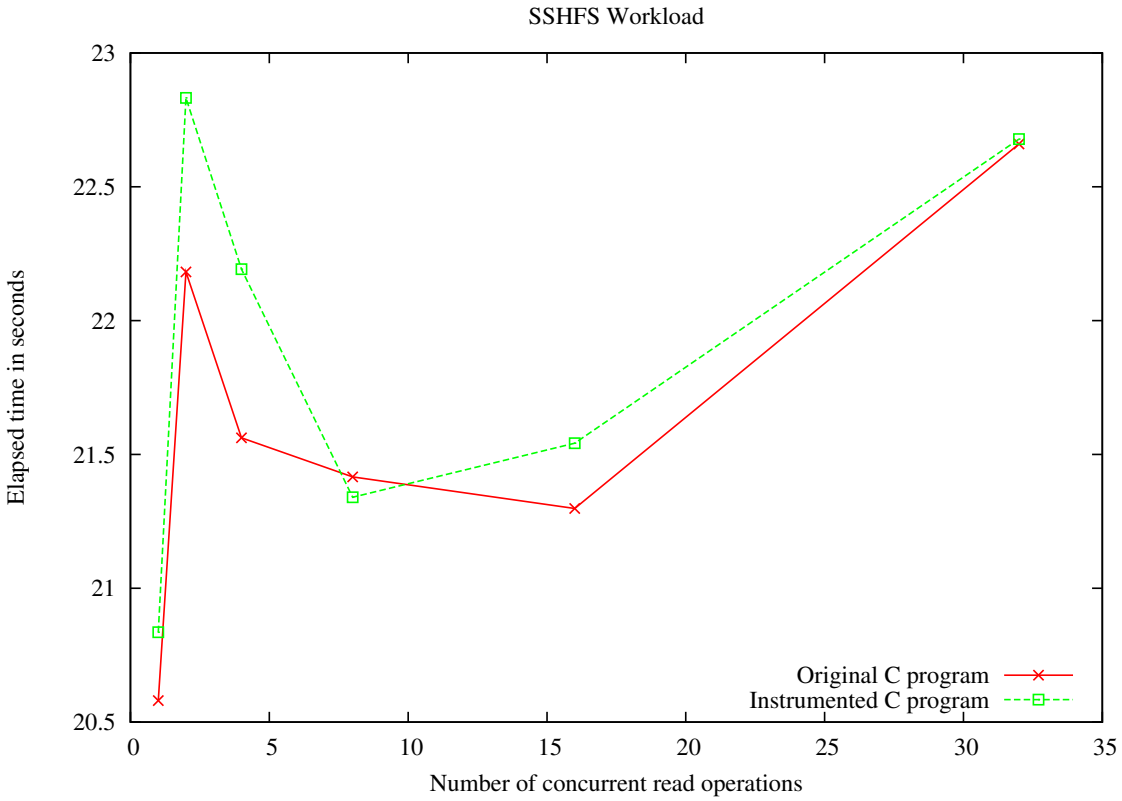

Figure 6.2: Performance comparison for *sshfs*.

read and write requests to the file systems, using two and three distinct locks respectively to synchronize data structures, logging and access to non thread-safe functions. In our experiments, we mount a remote directory over the corresponding file system and start a fixed number of concurrent threads, each of which is trying to download a number of large files. The total volume of data that is copied over the file systems is linear wrt to the number of threads. In both cases, the instrumented program has almost identical performance to the original program. The results seen in Figures 6.2 and 6.3 are an average of 10 and 5 runs respectively. Some discrepancies observed between the performance of the original and the modified version of curlftpfs are probably related to latencies due to traffic in the ftp server we used (`ftp.ntua.gr`).

flam3: a multithreaded program which creates "cosmic recursive fractal flames", i.e., (animations consisting of) algorithmically generated images based on fractals [26]. A single lock is used to synchronize access to a shared bucket accumulator that merges computations of distinct threads. We measured the time required to generate a long sequence of fractal images. The results again were almost identical for the original and the instrumented version of flam3. The results seen in Figure 6.4 are an average of 10 runs.

n	original			instrumented			improvement
	U	S	e	U	S	e	
1	0.00	0.63	32.90	0.00	0.67	31.10	5.47 %
2	0.00	0.74	33.50	0.00	0.60	32.60	2.67 %
4	0.00	0.75	32.85	0.00	0.78	32.27	1.77 %
8	0.00	0.57	26.29	0.00	0.53	31.85	− 21.16 %
16	0.00	0.76	33.45	0.00	0.68	32.86	1.76 %

Figure 6.3: Performance comparison for *curlftpfs*. Times are in seconds.

n	original			instrumented			overhead
	U	S	e	U	S	e	
1	50.06	0.17	50.37	50.29	0.21	50.56	0.38%
2	48.63	0.40	47.70	56.57	0.37	49.49	3.75%
4	59.60	0.71	48.69	61.97	0.66	48.94	0.51%
8	72.87	0.97	49.36	73.77	1.12	48.82	− 1.09%
16	71.97	2.21	48.93	72.96	2.07	48.90	− 0.06%
32	63.66	3.91	49.05	67.86	3.64	49.20	0.31%

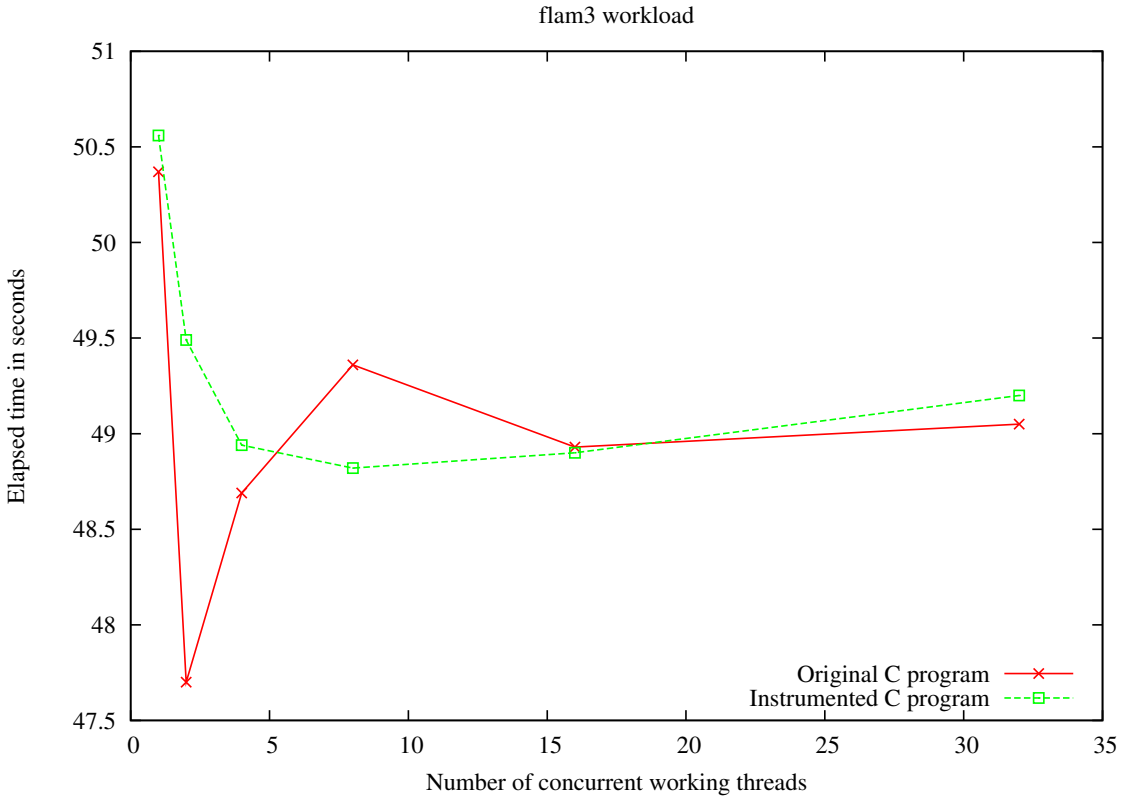

Figure 6.4: Performance comparison for *flam3*. Times are in seconds.

n	original			instrumented			overhead
	U	S	e	U	S	e	
1	674.59	0.04	676.60	734.13	0.07	737.83	9.05%
4	1873.30	772.15	2198.29	1937.72	836.03	2471.98	12.45%
8	5545.31	4631.34	4138.07	5334.92	5020.35	4625.92	11.79%

Figure 6.5: Performance comparison for *migrate-n*. Times are in seconds.

n	original			instrumented			improvement
	U	S	e	U	S	e	
1	8.309	0.004	8.316	8.240	0.000	8.250	0.79%
2	16.600	0.011	8.309	16.490	0.011	8.261	0.57%
4	33.270	0.030	8.324	32.899	0.030	8.233	1.10%
8	66.140	0.060	8.267	65.891	0.060	8.239	0.35%
16	124.846	0.126	8.266	124.467	0.126	8.243	0.28%

Figure 6.6: Performance comparison for *ngorca*. Times are in seconds.

migrate-n: estimates population parameters, effective population sizes and migration rates of n populations using genetic data [39]. The program maintains a work list of Markov chains and uses a thread pool to execute tasks until the work list becomes empty. Two locks are employed for implementing the thread pool and for accessing shared variables. It is worth mentioning that locks are dynamically allocated and several billion lock operations were executed during the program run.

ngorca: a multithreaded password recovery tool using exhaustive key search for DES-encrypted passwords in Oracle databases [43]. The program achieves speedup by splitting the search space of each encrypted password across threads, using multiple locks for implementing logging, counters and condition variables. The results again were almost identical for the original and the instrumented version of ngorca. The results seen in Figure 6.6 are an average of 7 runs.

tgrep: a multithreaded version of the utility program grep which is part of the SUNWdev suite of Solaris 10 [54]. The program achieves speedup by splitting the search space across threads, using multiple locks for implementing thread-safe queues, logging and counters. In our experiment, we looked for an occurrence of a six-letter word in a directory tree containing 100,000 files. The results seen in Figure 6.7 are an average of 10 runs. The instrumented program is up to 19% slower than the original program. This is due to the fact that tgrep is not only lock-intensive (about 1.5 million lock operations were executed in our test run), but also it is by far the benchmark with the longest effects that we could find. The maximum effect size for a function is 54 and the average effect size is 19.5, which are both about five times higher than the second next benchmark (ngorca). Furthermore, the program employs seven distinct global locks; the dynamically calculated future lockset had a maximum size of five elements and an average size of 1.3, which again were about five times higher than the second next benchmark.

n	original			instrumented			overhead
	U	S	e	U	S	e	
1	7.55	6.94	14.77	8.91	6.98	17.04	15.38%
2	7.44	6.89	9.91	8.96	7.23	11.36	14.65%
4	7.93	7.64	6.79	9.06	7.60	7.59	11.83%
8	9.36	8.89	5.31	11.71	9.76	6.14	15.47%
16	12.36	10.79	4.97	15.09	11.67	5.94	19.53%
32	13.64	11.74	5.19	15.56	12.03	6.18	19.09%
64	13.24	11.64	5.53	14.80	11.66	6.42	16.10%

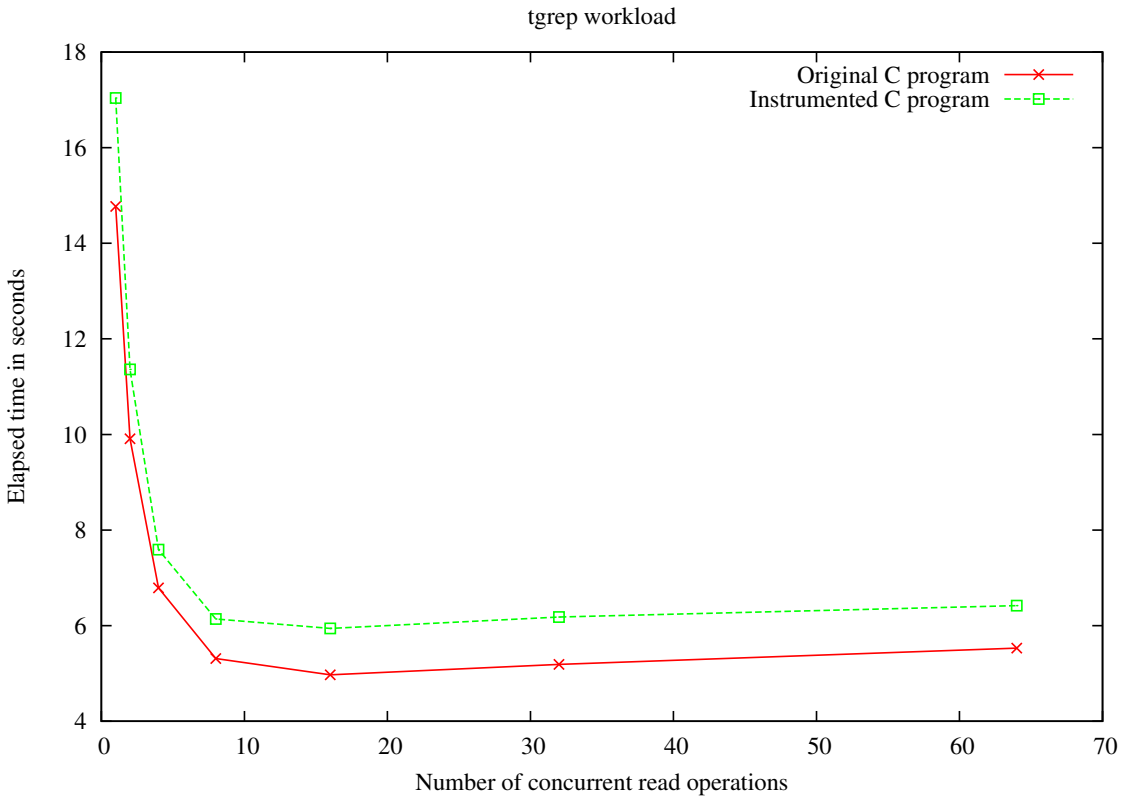

Figure 6.7: Performance comparison for *tgrep*. Times are in seconds.

Chapter 7

Conclusion

7.1 Concluding Remarks

As enabling as concurrent computing has been in exploiting the potential of modern multiprocessor computer systems, designing concurrent programs has introduced several challenges, with deadlocks being one the most important. In this thesis we have presented the implementation of a tool that dynamically avoids deadlock states for multithreaded C programs, that utilize the Pthreads API. This tool is able to treat non block-structured locking that is rather common in real-world C code, as we showed by gathering statistics from a codebase comprising of more than a hundred multithreaded open-source projects. We described the main aspects of both the static and the run-time parts of this implementation and elaborated on some of the tricky and interesting situations we encountered. We also included a benchmark section to assess the performance of our approach. As it turns out, the overhead induced by computing the future locksets and by blocking threads more often, as they wait for the future lockset to become available when trying to acquire a lock, is quite reasonable.

7.2 Future Work

As we mentioned in Section 5.6 there are several limitations in the current version of our tool. As far as library code is concerned, a future plan is to allow the user to provide effect annotation for library functions that contain lock operations (currently library functions are treated as if they bore no effect). The analysis could also be extended to deal with assembly code as well (CIL can analyze inline assembly instructions).

Additionally, one of our major limitations is handling recursive and unbounded structures, and higher levels of indirection (via pointer dereference), which are quite common in real world C projects. In order to handle such cases, our tool should be equipped with a more precise pointer analysis framework. Also, a more sophisticated alias and escape analysis could be employed to track mutations on mutex references that are shared between threads.

Furthermore, even though the type system, on which this implementation is based [31], supports recursive functions that bear non empty effects, the current version of our tool does not. Adding support for this feature remains future work.

Finally, a more far fetched plan would be to compute the future lockset for every lock operation statically. This would boost the performance of our tool, since the computation of the future lockset at runtime is one the major factors of the overhead imposed by our tool. An alternative to this would be to dynamically compute the future lockset for a lock operation just once, and then save and reuse this result for every other time this specific operation is executed. This optimization targets lock operations residing in loops, which account for the vast majority of the lock operations in most projects. Therefore, it would greatly enhance our tool's performance.

Bibliography

[1] ADVRP (Academic Distance Vector Routing Protocol). `http://code.google.com/p/advrp/`.

[2] cmus, small, fast and powerful console music player for Unix-like operating systems. `http://cmus.soceforge.com`.

[3] Dao Programming Language. `http://www.daovm.net`.

[4] github, Online project hosting usigng Git. `http://github.com`.

[5] Google Code, Google's official developer site. `http://code.google.com`.

[6] libactor, an Actor Model Library for C. `http://code.google.com/p/libactor/`.

[7] Memcached, Free and open source, high-performance, distributed memory object caching system. `http://memcached.org`.

[8] MooseFS: A File System for highly reliable petabyte storage. `http://www.moosefs.org`.

[9] Portland Doors, a free, open-source implementation of Sun Microsystems' Doors API. `http://cs.pdx.edu/~davisl`.

[10] s3backer - FUSE-based single file backing store via Amazon S3. `http://s3backer.googlecode.com/`.

[11] The FreeRADIUS Project. `http://freeradius.org`.

[12] Valgrind, a suite of tools for debugging and profiling. `http://valgrind.org`.

[13] G. Boudol. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, pages 140–154, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. *SIGPLAN Not.*, 37:211–230, November 2002.

[15] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '03, pages 213–223, New York, NY, USA, 2003. ACM.

[16] S. Chandra, B. Richards, and J. R. Larus. Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols. *IEEE Trans. Softw. Eng.*, 25:317–333, May 1999.

[17] G. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[18] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[19] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3:67–78, June 1971.

[20] T. Cogumbreiro, F. Martins, and V. T. Vasconcelos. Compiling the π-calculus into a Multithreaded Typed Assembly Language. *Electron. Notes Theor. Comput. Sci.*, 241:57–84, July 2009.

[21] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[22] A FTP filesystem based on cURL and FUSE. `http://curlftpfs.sourceforge.net/`.

[23] D. L. Detlefs, K. Rustan, K. Rustan, M. Leino, M. Leino, G. Nelson, G. Nelson, J. B. Saxe, and J. B. Saxe. Extended static checking. Technical report, December 1998.

[24] E. W. Dijkstra. "The mathematics behind the Banker's Algorithm. In *Selected Writings on Computing: A Personal Perspective*, pages 308–312. Springer-Verlag New York, Inc., 1982.

[25] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37:237–252, October 2003.

[26] flam3.com. Cosmic recursive fractal flames. `http://flam3.com/`.

[27] C. Flanagan and M. Abadi. Types for Safe Locking. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 91–108, London, UK, 1999. Springer-Verlag.

[28] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[29] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[30] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, pages 479–495, 2002.

[31] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In *Proceedings of the 6th ACM SIGPLAN workshop on Types in Language Design and Implementation*, TLDI '11, New York, NY, USA, 2011. ACM.

[32] D. P. Helmbold and C. E. McDowell. A Taxonomy of Race Detection Algorithms. Technical report, University of California at Santa Cruz, Santa Cruz, CA, USA, 1994.

[33] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[34] S. P. Jones. Beautiful Concurrency. In *Beautiful Code: Leading Programmers Explain How They Think (Andy Oram and Greg Wilson Ed.)*. O'Reilly Media, 2007.

[35] N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42:291–347, December 2005.

[36] N. Kobayashi. A New Type System for Deadlock-Free Processes. In C. Baier and H. Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer Berlin / Heidelberg, 2006.

[37] L. Lamport. Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[38] N. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[39] A tool that estimates population size and migration rate. `http://popgen.sc.fsu.edu/Migrate/Migrate-n.html`.

[40] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[41] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[42] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[43] A password recovery tool for Oracle Database. `http://code.google.com/p/ngorca/`.

[44] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 114–136, London, UK, 1999. Springer-Verlag.

[45] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[46] K. Poulsen. Tracking the blackout bug, February 2004. `http://www.securityfocus.com/news/8016`.

[47] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. *SIGPLAN Not.*, 38:179–190, June 2003.

[48] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical Static Race Detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, March 2010. Accepted for publication.

[49] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[50] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[51] SSH FileSystem. `http://fuse.sourceforge.net/sshfs.html`.

[52] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[53] K. Suenaga. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 155–170, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] Multithreaded `grep`. Part of Sun Microsystems' *Multithreaded Programming Guide*, available at `http://docs.sun.com/app/docs/doc/806-5257`.

[55] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language. In *PLACES*, pages 95–109, 2009.

[56] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engg.*, 10:203–232, April 2003.

[57] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[58] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 252–263, New York, NY, USA, 2009. ACM.

[59] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39:221–234, October 2005.