School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Static and Dynamic Data Race Detection for C/C++ Programs

by

Michael Liu and Vi Nguyen

Thesis submitted as a requirement for the degree of

Bachelor of Software Engineering

Submitted:	23 May 2017	Student ID:	z5017631, z5015922
Supervisor:	Yulei Sui	Topic ID:	3544

Abstract

The C, and C++ language is heavily utilised in systems that require speed, and efficiency. Programs are often multi-threaded to achieve these requirements via concurrency. The use of concurrency within these programs can lead to bugs known as data races. These bugs can be difficult to find and debug due to the non-deterministic behaviour. That is why there is a need for static analysis tools to assist developers in locating, and debugging data races within their programs. This thesis project aims to further enhance the existing multithreaded analysis (MTA) module in the SVF tool to detect data races, paired with the development of an Eclipse plug-in utilising SVF, to allow developers using Eclipse to easily locate and debug data races. This thesis has developed the tools required to perform the analysis, and to intuitively display this information to the user, but has also discovered the limitations of such a tool.

Acknowledgements

Thank you to Yulei Sui for his guidance and advice throughout the entirety of our thesis. Who without, we would not have been able to complete our work.

Abbreviations

FSAM	Flow Sensitive Analysis Model
CFG	Control Flow Graph
MHP	May Happen in Parallel
MTA	Multithreaded Analysis
IDE	Integrated Development Environment]item[URI] Uniform Resource Identifier

Contents

List of Figures

Chapter 1

Introduction

The C, and C++ language was designed with a focus towards system programming, and embedded, resource-constrained systems with performance, and efficiency as its key design attributes[Stroustrup, 2014b]. They have also been utilised in other contexts such as software infrastructure and resource-constrained applications, which include servers and performance-critical applications (e.g space probes) [Stroustrup, 2014a].

Multiprocessor systems have existed for a while but only recently have chip manufacturers started favouring multicore designs over better performance with a single core. Consequently, multicore computers, and embedded devices are much more common. The increased computing power of these came not from a single task being executed more quickly, but from running multiple tasks concurrently [Williams, 2012].

Due to the nature of the systems in which C, and C++ implement, there is a greater need for concurrent programming in order to fully utilise the resources to fulfill the key attributes of efficiency and performance. With the advantages that do come with concurrent programming, there are also disadvantages. Data races are one of the bugs that occur from failure to uphold certain precautions.

A data race occurs when there are concurrent memory accesses by two separate threads and there are no synchronisation operations involved[Oracle, 2010]. These data races can lead to race conditions, which are a flaw that occurs when the output relies on the timing or ordering of events.

For C/C++ programs specifically, if a data race occurs, the behavior of the program is undefined[ANSI, 2011b, ANSI, 2011a]. Undefined behaviour renders the whole program illogical if certain rules of the language are violated. Race conditions also prove difficult to debug as reproducing the bug may be troublesome since the end results are non-deterministic and rely on the timing between the interfering threads. Non-deterministic implies the notion that for an algorithm given the same inputs, it can exhibit different behaviours on separate executions[Floyd, 1967].

Therefore it would be advantageous to statically detect data races rather than attempting to debug them after execution. This is the basis of the thesis.

1.1 Current available tools

There are external tools which already do analysis on multithreaded programs, namely Locksmith: practical static race detection for C, CHORD: effective static race detection for java, RELAY: static detection on million lines of code and FSAM: Sparse Flow-Sensitive Pointer Analyss for Multithreaded Programs - the tool in which we used to help discover data races. These tools for multithreaded program analysis currently detect possible data race issues. Furthermore, the reasons for picking FSAM over the other tools stems from the drawbacks and limitation of the other tools which will be discussed later on in the background.

1.2 Aims

The aims are to statically detect data races as accurately as possible in C, and C++ programs, and to display the information to the user so that they will be able to use it to debug their program. To achieve this we further enhance the SVF tool to detect and locate data races. Then develop an Eclipse plug-in utilising this tool to assist developers in debugging these data races.

1.3 Outcomes

For this project we were able to develop a tool to detect and locate data races for relatively small C and C++ programs. Due to the nature of our implemntation, and the desire to develop a precise tool that will locate all data races, it was not able to scale well with our current algorithms, which we will discuss in the evaluation.

All in all, we developed an Eclipse plug-in, utilising our data race detection tool (built upon SVF), to add functionality to the developer's Eclipse environment for data race detection. As a result, users are able to easily identify areas of code which contain a data race, and quickly investigate the other area of code in which the data race is occurring.

1.4 Structure of thesis

The rest of the thesis will contain:

- Assessing tools and approaches by other people. (Background)
- Explaining our approach and justifying our methods. (Methodology)
- Evaluating our solution. (Evaluation)
- Summarising our achievements and limitations. (Conclusion)

Chapter 2

Background

To begin resolving data race issues, four papers are examined that have contributed to detecting data races: LOCKSMITH, CHORD, RELAY, and FSAM - the tool that is used in this paper. Each paper has a particular focus, but still addresses the data race problem through the solution they present. The paper will draw ideas with each paper to present a unique solution to detect data races. The main techniques that will be examined are whether the paper uses a flow-sensitive analyser or a flow-insensitive analyser, and also the solution that they came up with in other to address the inaccurate information that came with the type of analyser they used. Further, this section will also describe some programs that exists on the eclipse marketplace.

2.1 RELAY: static detection on million lines of code

The goal of this tool is to statically detect data races. It does this by first computing a call graph, traversing it bottom up. It then does a Steensgard's flow-insensitive points-to analysis, computing conservative representitive nodes for all lvalues using the call graph. The tool then uses this information in order to execute the three main algorithm of its analysis: [Wen et al., 2007]

1. Symbolic execution.

2. Relative lockset analysis.

3. Guarded Access Analaysis.

2.1.1 Symbolic execution

RELAY maps each local lvalue to a value expressed in terms of its incoming value of formals (parameters), and incoming values of globals.[Wen et al., 2007]

2.1.2 Relative lockset analysis

Locksets are calculated by seeing the set of locks held at a particular point in the code. The analysis reveals if a lvalue is being accessed simultaneously and is computed by showing only the locks that are held on entry to a function.[Wen et al., 2007]

2.1.3 Guarded Access Analysis

Guarded Access Analysis contains a tuple comprising of a lvalue, relative lockset and the sort of access it is (read/write memory access), and will compare pairs of guarded accesses, showing the potential lvalue alias [Wen et al., 2007]. For Relay to scale on large programs, guarded access analysis iterates through all the statement functions in a flow-insensitive manner. The analysis would detect a potential error if:

- a pair of guarded access had a lockset overlap.

- the lvalue is the same (aliased/same value).

- at least one of the memory accesses is a write.

2.2 CHORD: effective Static Race Detection for Java

The goal of this tool is to statially detect data race issues within the java language and on large programs. The tool computes an initial over-approximation set of unordered pairs of memory accesses that may cause a data race [Naik et al., 2006]. From this initial list, the tool then prunes and detects data races through four key steps:

1. Reachable-pairs computation.
2. Aliasing-pairs computation.
3. Escaping-pairs computation
4. Unlocked-pairs computation

2.2.1 Reachable-pairs computation

After computing all pairs of memory accesses that may be involved in a data, the tool then recomputes pairs that are from thread spawning site, which in turn has to be reachable from main [Naik et al., 2006]. Effectively, this new list of pairs is a subset of all pairs of memory accesses and uses a call graph to detail if the memory access is called from main.

2.2.2 Aliasing-Pairs computation

Use of alias analysis to see if the pairs of instructions accesses the same memory location as data races would only occur if two instructions access the same memory space [Naik et al., 2006]. If the pair of memory do not access the same location, it is pruned from the list of pairs.

2.2.3 Escaping-Pairs computation

Removal of pairs that access thread shared data. A thread shared data are objects in memory that are reachable from some arguments of a thread spawning call site or it is reachable from a static field. This mean it remove pairs that may access the same space in memory but has no data race issue as a result. [Naik et al., 2006]

2.2.4 Unlocked-Pairs Computation

Uses a lock analysis that depends on the call-graph and alias analysis. It first characterises the set of locks held by a given thread, and will exclude pains that are protected under a common lock [Naik et al., 2006]. Any remaining pairs will then be reported as a data race issue.

2.3 Locksmith: Practical Static Race detection for C

The goal of this tool to statically detect data races within the C language. It works by enforcing one of the most common rule for race prevention, for every shared memory location p, there must be some lock l that is held whenever p is accessed [Pratikakis et al., 2011]. Locksmith detects data races in five major steps:

1. Label and constraint generation.
2. Sharing analysis.
3. Lock state analysis.
4. Correlation inferrence.
5. Escape checking.

2.3.1 Label and constraint generation

Traverses the C intermediate language in order to form two key abstractions:

- Label flow graph.
- Control flow graph (CFG).

The label flow graph models the flow of data within the program which is complemented by the control flow graph showing key actions on the data. The label flow graph is a collection of nodes depicting the static representation of the run time memory location containing locks or other data and edges representing the flow of data through the program. The label flow graph also incoporates information about functions calls. The Abstract Control Flow Graph (or CFG) depicts nodes capturing operations in the program which are important to data race detection and then relate them from the label flow graph. In this phase, Locksmith generates linearity constraints and contextual constaints. [Pratikakis et al., 2011]

2.3.2 Sharing analysis

Determines the set of locations that could be potentially simultaneously be accessed by two or more threads during a program's execution, and limits the analysis to the region of shared access.[Pratikakis et al., 2011]

2.3.3 Lock State Analysis

It computes the state of each lock at every program point. That is, it checks whether the point of execution is a lock or unlock operation. [Pratikakis et al., 2011]

2.3.4 Correlation Inferrence

The core of the race detection algorithm. For each shared variable, they intersect the sets of locks held at all of its accesses. It is called the guarded-by set for that location, and if the set is empty they report a possible data race. [Pratikakis et al., 2011]

2.3.5 Escape Checking

Removes some false positives in the linearity check, by allowing users to specify that a data structure is linear [Pratikakis et al., 2011].

2.4 FSAM: Sparse Flow-Sensitive Pointer Analysis for multithreaded Programs

FSAM (the tool used in the paper) is used to detect data race issues. FSAM focuses on scalability for large multithreaded C/C++ programs, and maintains precision and soundness. The idea stems from that the three papers did not use flow-sensitive analysis, thus improving FSAM's precision to detect data races. FSAM is implemented in LLVM, and will convert the source code of each program into bit code files using clang in order to analyse the program. In order to help with scalability, FSAM does an imprecise pre-analysis with Anderson's analysis. It is then followed by a precise flow-sensitive analysis by propagating points-to facts along pre-computed def-use chains sparsely. FSAM relies on the following analysis in order to detect data races: [Sui et al., 2016]

1. Value-flow analysis.
2. Interleaving analysis.
3. Lock Analysis.

When these analysis are complete the remaining pairs are reported back in order to reveal any data races.

2.4.1 Pre-Analysis

Traditional data-flow based flow-sensitive pointer analysis relied on computing and maintaining a collection of information at every program point in a program's control flow. This is inefficient as information is pointed to blindly between nodes in the control flow graph (CFG), diminishing the scalability in analysing large programs. FSAM addresses this issue by first applying an imprecise analysis of Andersen's Analysis then precisely analyse the the remaining def-use chains in the program sparsely. Further, an intra-thread control flow graph (ICFG) is constructed from an abstract thread. The abstract thread refers to a call of `pthread_create()`. The ICFG is used to further model fork and join operations giving information for interleaving analysis.[Sui et al., 2016]

2.4.2 Interleaving Analysis (May Happen Parallel Analysis)

Interleaving Analysis reasons about fork and join operations by using the ICFG, to discover pairs that may happen in parallel. The objective of interleaving analysis is to use ICFG to reason about the flow and context sensitivity of all threads, determining if two pieces of code are running in parallel. This information is then propagated to value-flow analysis. [Sui et al., 2016]

2.4.3 Value-Flow Analysis (Alias Analysis)

Value-Flow analysis utilises may happen pairs to find common value flow to produce aliased pairs, using the CFG and also the data from pre-analysis done at the start.[Sui et al., 2016].

2.4.4 Lock Analysis

Analyses lock/unlock operations, in a context sensitive manner to identify aliased pairs which may interfere with each other. By reasoning about statements from different mutex regions being protected by a common lock are interference free and using context sensitivity which ensures lock analysis can determine different calling contexts, many of the value-flow analysis pairs are filtered out. The remaining data that is after these three analysis are then likely to be data races. [Sui et al., 2016]

2.5 Eclipse Programs

There are very few plugin that exists in the eclipse markerplace that helps with data race detection let alone a plugin that deals with C/C++ programs. This section introduces plugins that are existent on the eclipse marketplace in relation to finding data race issues and they way they present their result.

2.6 ThreadSafe for Eclipse

ThreadSafe is a tool for finding concurrency bug and performance issues by using a static analysis tool. This program is only used on the java program, so analysis of C/C++ cannot be done. ThreadSafe result is presented in the Eclipse IDE Java perspective, outlining the possible concurrency violation in the program.

2.6.1 interface

When running the plugin, ThreadSafe will do the anlysis on the source code and will return its result on the eclipse IDE perspective. By clicking on the results in this perspective it will take the user to the line with the concurrency issue. The view details a list of location in the source code and have different warning symbols in accordance to the severity of the problem. If the access is a synchronised access (that is a lock is held when accessing the shared informtion), ThreadSafe will show that the lock is a safe access highlighted in green, whereas an unsafe access will be highlighted in purple to communicate to the user that a problem may occur in that area seen in A.6. To detect and report the data race issues in ThreadSafe, they follow these techniques [Comtemplate, 2009]

1. Context-sensitive inter-procedural analysis.
2. Inter-class analysis.
3. Aggregation and filtering of results.
4. Straightforwardd presentation of results.

2.6.2 Context-sensitive inter-procedural analysis

ThreadSafe track synchronisation calls on method calls and track references of shared data between methods. This allow ThreadSafe to increase accuracy of race detection, revealing more concurrency defects and further reduces false positives [Comtemplate, 2009]

2.6.3 Inter-class Analysis

Tracking of interactions between classes to report concurrency defects. This is monitored by tracking method calls across class hierarchy and single classes.[Comtemplate, 2009]

2.6.4 Aggregation and filtering of results

After the context-sensitive inter-procedural analysis and inter-class analysis, the information is used to filter and aggregate likely concurrency problems.[Comtemplate, 2009]

2.6.5 Straightforward presentation of results

After the discovery of the bug, ThreadSafe presents the user with the information to find and resolve the underlying problem.[Comtemplate, 2009]

Chapter 3

Methodology

To statically detect data races in C and C++ programs it was decided to use FSAM to further enhance the MTA module within SVF to develop our data race detection tool. An Eclipse plugin will later be development that implements this tool. As a result Eclipse users can detect possible data races within their programs and assist in debugging it.

3.1 Justification

As previously discussed in the background, due to FSAM's flow sensitive component, it helps detect less false positives compared to the other possible approaches.

Furthermore, alias and lock analysis is crucial in identifying a data race as they only occur when there are accesses to the same memory region and when the piece of memory is not protected by a common lock.

FSAM's interleaving analysis models flow and context sensitivity, which in turn will help identify aliases which may run in parallel. Due to the aforementioned reasons, FSAM's static race detection tool was used as a result of its precision, accuracy, and its three analytics components which will help identify possible data races.

Eclipse is an integerated development environment (IDE) which offers users more functionality when programming. Eclipse was chosen due to it being one of the most popular C/C++ IDEs, and its plug-in support. A plug-in using the data race detection tool will ideally allow users to quickly identify and debug data races.

3.2 SVF Tool Development

To emulate FSAM's approach, there are three steps required to analyse the program's instructions to determine whether there may be a possible data race. These are:

1. Alias analysis (value flow)
2. May-happen-in-parallel (MHP) (interleaving)
3. Lock analysis

3.2.1 Instruction Collection

In order to perform these analyses, all instructions are required to be collected for examination. Collection of all instructions from the program can be seen from the figure below:

```
std::set<const Instruction *> instructions;

for (Module::iterator F = module.begin(), E = module.end(); F != E; ++F) {
    for (inst_iterator II = inst_begin(&*F), E = inst_end(&*F); II != E; ++
        II) {
        const Instruction *inst = &*II;
        if (const StoreInst *st = dyn_cast<StoreInst>(inst)) {
            instructions.insert(st);
        } else if (const LoadInst *ld = dyn_cast<LoadInst>(inst)) {
            instructions.insert(ld);
        }
    }
}
```

Figure 3.1: Collecting all instructions.

It iterates through every instruction in every module in the program and checks whether it is a store or load instruction. As previously discussed, data races only occur when there are interfering reads and writes, so only those instructions are collected. In this case, we use a set to collect all instructions to guarantee uniqueness of instructions collected so we don't do repeated, redundant analysis on instructions.

3.2.2 Instruction Pairing and Analysis

Instruction Pair Class

An instruction pair class was implemented as a means to retain certain information throughout the alias, MHP, and lock analysis. Specifically, we wanted to retain the information about the alias result between the two pairs. There are four types of aliasing:

- `NoAlias`

  There is never an immediate dependence between any memory reference based on one pointer and any memory reference based the other.

- `MayAlias`

  The two pointers might refer to the same object.

- `PartialAlias`

  The two memory objects are known to be overlapping in some way, regardless whether they start at the same address or not.

- `MustAlias`

  The two memory objects are guaranteed to always start at exactly the same location.

```
class InstructionPair {
public:
    ...
private:
    const llvm::Instruction* inst1;
    const llvm::Instruction* inst2;
    const llvm::AliasResult  alias;
```

Figure 3.2: InstructionPair Class

Using this design (as seen above) we can modify the alias result of the pair during the alias analysis. This will allow the information to be retained throughout the entirety of the analaysis so at the end, when we output for the Eclipse plug-in, we can distinguish whether it may alias, must alias, etc.

Pairing and Analysis

There were substantial changes to the way we initially created, collected, and analysed pairs. No longer were we collecting pairs by storing them in a vector because it scaled quite poorly, as it quickly ran out of memory. For example, we analysed a program that contained 13,657 instructions which ended up being 93,249,996 (${}^{13657}C_2$) pairs. Memory was quickly exhausted.

To overcome this issue we no longer stored every created pair in a vector, but analysed each pair as it was created, then analysed them. If they contained a data race, then we would add it to a vector containing the pairs with data races. There is also some initial analysis performed on the individual instruction before pairing to improve the efficiency of the algorithm. This can be seen in the figure below:

```
for (auto it = instructions.cbegin(); it != instructions.cend();){
    ...
    if (!isShared(*it,module)) {
        ++it;
        continue;
    }
    for (auto it2 = ++it; it2 != instructions.cend(); ++it2){
        InstructionPair pair = InstructionPair(*it,*it2);
        if (hasDataRace(pair, module,mhp,lsa)){
            pairs.push_back(pair);
        }
    }
}
```

Figure 3.3: Instruction Pairing and Analysis

The two main drivers of this algorithms are the `isShared()` function and `hasDataRace()` function.

isShared() function

The main purpose of the `isShared()` function is to analyse the first instruction before attempting to analyse its possible pairs. The function first needs to convert the `llvm::Instruction*` to an `llvm::(Store|Load)Inst*` to be able to obtain the `llvm::Value*` to perform analysis on. This is achieved by overloading the function as seen below in Figure 3.4, and Figure 3.5.:

```
bool isShared(const Instruction *loc, llvm::Module& module){
    if (const StoreInst *p1 = dyn_cast<StoreInst>(loc)){
        return isShared(p1->getPointerOperand(), module);
    } else if (const LoadInst *p1 = dyn_cast<LoadInst>(loc)){
        return isShared(p1->getPointerOperand(), module);
    }
    return false;
}
```

Figure 3.4: Overloaded isShared() function - conversion

The passed in instruction needs to be dynamically casted as a store or load instruction. This is because you cannot call `getPointerOperand()` on an instruction to retrieve the `llvm::Value*`. But this is possible on store and load instructions. The function will return false if the instruction is neither a store or load instruction. As previously discussed, this is because only store and load instructions are involved in data races so we can safely assume a data race is not possible if they are not store or load instructions.

Once the conversion to an `llvm::Value*` is complete we can then analyse it. This can be seen below:

```
bool isShared(const llvm::Value *val, llvm::Module& module){
    PointerAnalysis* pta = AndersenWaveDiff::createAndersenWaveDiff(module);
    PAG* pag = pta->getPAG();
    const PointsTo& target = pta->getPts(pag->getValueNode(val));
    for (PointsTo::iterator it = target.begin(), eit = target.end();
            it != eit; ++it) {
        const MemObj *obj = pag->getObject(*it);
        if (obj->isGlobalObj() || obj->isStaticObj() || obj->isHeap()){
            return true;
        }
    }
    return false;
}
```

Figure 3.5: isShared() function - analysis

Firstly, we retrieve the Pointer Assignment Graph (`PAG`) from the Pointer Analysis base class (`PointerAnalysis`). These two classes allows us to retreive the `PointsTo` vector for further analysis. Then we can use the `PointsTo::iterator` to iterate through all the `NodeID` in the vector, and obtain the `MemObj` via the `NodeID`. The `MemObj` is important as it allows us to distinguish whether the `llvm::Instruction` (the original object being passed in) is pointing to a global, static, or heap object.

These objects are of interest as these objects can be involved in data races because they may potentially be shared between two threads. If the instruction is pointing to any of the objects, then we return true so that the Instruction will be paired up with other Instructions to be further analysed. If not, then we return false to skip that instruction as we know that it cannot be involved in a data race.

This analysis of the instruction assists in saving many computational cycles if the instruction cannot be involved in a data race. Without this check, every single instruction will be paired up for analysis which does not scale very well - wasting computation cycles as it will be performing redundant checks on instruction pairs.

hasDataRace() function

The main part of the pair analysis is the `hasDataRace()` function. This function performs the 3 sub-phases of analysis discussed earlier. These are:

1. Alias analysis
2. MHP analysis
3. Lock analysis

These three steps of analysis are crucial in determining whether the instruction pair has a data race. Alias analysis involves comparing the two instructions, and evaluating whether they refer to the same object, or overlap in memory.

Alias analysis is performed as it is necessary to know if there is any type of aliasing involved. Data races can only occur when two instructions are reading/writing in the same shared memory. That is why we only continue analysis on the pairs that alias. How this is done is seen below, specifically lines 2-15:

```
1  bool hasDataRace(InstructionPair &pair, llvm::Module& module, MHP *mhp,
       LockAnalysis *lsa){
2      PointerAnalysis* pta = AndersenWaveDiff::createAndersenWaveDiff(module);
3      bool alias = false;
4      if (const StoreInst *p1 = dyn_cast<StoreInst>(pair.getInst1())){
5          if (const StoreInst *p2 = dyn_cast<StoreInst>(pair.getInst2())){
6              AliasResult results = pta->alias(p1->getPointerOperand(),p2->
                   getPointerOperand());
7              pair.setAlias(results);
8              alias = (results == MayAlias || results == MustAlias || results
                   == PartialAlias);
9          } else if (const LoadInst *p2 = dyn_cast<LoadInst>(pair.getInst2()))
               {
10             ...
11         }
12     } else if (const LoadInst *p1 = dyn_cast<LoadInst>(pair.getInst1())){
13         ...
14     }
15     if (!alias) return false;
16     if (!mhp->mayHappenInParallel(pair.getInst1(), pair.getInst2())) return
           false;
17     return (!lsa->isProtectedByCommonLock(pair.getInst1(),pair.getInst2()));
18 }
```

Figure 3.6: hasDataRace() function

Firstly the Pointer Analysis base class is retrieved from the module, and the boolean which tracks whether the pairs do alias is set to false. Similar to how instructions are analysed in the `isShared()` function, instructions are dynamically casted to store and load instructions as the Pointer Analysis base class requires the instruction's `llvm::Value*` to determine whether the two instructions are alias. Once we collect the results of the `pta->alias()` function, we set the pair's alias to the result, and set the alias boolean to true if there was any sort of aliasing. We then check this boolean to see if any aliasing was involved. If there is no aliasing involved, then we can stop analysing the current instruction pair as it is guaranteed that data races cannot occur on pairs that have no form of aliasing.

If the pairs do alias, then we continue with the next phase of analysis - MHP analysis. MHP analysis is performed as data races can only occur if the instructions are on threads that are alive at the same time. So we check the two instructions to see if they may be running on threads that are running in parallel. The analysis is performed on line 16 in Figure 3.6 above. The MHP base class provides the `mayHappenInParallel()` function to determine whether the two instructions may be running on threads that are alive at the same time. If they may not happen in parallel then we return false as data races can only occur if the instructions are running on threads that are alive at the same time.

If they may happen in parallel then we continue to perform the final phase of analysis - Lock analysis. Lock analysis is performed as data races occur when they are accessing shared memory at the same time. To prevent shared memory being accessed at the same time, locks are utilised so that different threads can only exclusively access the shared memory. That is, if a thread has access to the shared memory then other threads are unable to access it.

We analyse the lock and unlock operations upon the two instructions to determine whether they are protected by a common lock. This can be seen on line 17 in Figure 3.6 above. We utilise the lock analysis base class' functions to determine whether the instruction pair is protected by at least one common lock.

We return the result of the lock analysis as it is the final phase of the pair analysis. If it has reached this point then we know that the pairs alias and may happen in parallel. So if it is protected by a common lock, then it will return false as there is no data race. If it is not, then it will return true as there is a possibility of a data race.

After the three phases of analysis, if there is a data race then the pair of instructions will be added to the pairs of with data races. After every pair has been analysed then we output the results to be used by the Eclipse plug-in.

3.2.3 Outputting for the plug-in

The final step for our data race detection tool is to output the results in a format for our Eclipse plug-in to read from. We decided that the best format for the plug-in was simply just have the line number and file name. The format can be seen in Figure 3.8. Below is how we format the results for the plug-in:

```
std::ofstream output;
output.open("output.txt");
std::string s1;
std::string s2;
std::regex line("ln: (\\d+)");
std::regex file("fl: (.*)");
std::smatch match;
for (auto it = pairs.cbegin(); it != pairs.cend(); ++it){
    s1 = getSourceLoc(it->getInst1());
    s2 = getSourceLoc(it->getInst2());
    if (s1.empty() || s2.empty()) continue;
    if (std::regex_search(s1, match, line))
        output << match[1] << std::endl;
    if (std::regex_search(s1, match, file))
        output << match[1] << std::endl;
    if (std::regex_search(s2, match, line))
        output << match[1] << std::endl;
    if (std::regex_search(s2, match, file))
        output << match[1] << std::endl;
}
output.close();
```

Figure 3.7: Generating data race pairs for the plug-in

It simply creates an output.txt file and populates it with the results of the pair analysis.

It loops through all the pairs of interest and adds the relevant information to the output.txt file. It uses regular expressions to obtain the line number and file name. There is also a check on line 11 which accounts for instructions that do not contain any information pertaining to the line number or file name. This may be due to a temporary instruction being created and involved in a data race, but since there is no information we can use to identify the location of the instruction, we do not include include it in our output for the plug-in.

3.3 Eclipse Plug-in Development

This was the next phase in the development of the thesis. There were some design decisions that had to be made in order to keep the run-time execution of eclipse markers to a minimum. As such this section describe the implementation of transitioning the output of the tool onto the eclipse editor. The development of this plugin heavily relies on the java extension points, especially the builder and nature extension point, marker extension point and marker resolution extension point.

3.3.1 Output file

After running the data race detection tool in the project there would be an output text file that is used to create the markers. The text file is saved directly under the source directory of where the project is running. Due to format of the file, it was a design implementation to read the file, store the information in memory and then delete the file as it had no useful information for the user. A drawback to this design was that there can not be a file named the same, otherwise that file is deleted. This is huge downfall, especially for the user if they were to name their file the same as the output. However, there was no other plausible alternatives in which the result of the data race detection tool could be read.

3.3.2 Reading in file

The next part of this was to read the output file line by line in order to use the information to highlight conficting pairs. The format of the file is as follows

```
linenumber for file 1
file 1 location
linenumber for file 2
file 2 location
...
```

Figure 3.8: Output File format

The four line repeat for all possible conflict pairs throughout the file. The line number is a number between 1 and the max number of line in the particular file. The file location is a uniform resource identifier (URI) giving the path to the file which is all that is needed to know where to place the marker.

3.3.3 The Markers

After reading the output from the data race detection tool, the markers get linked together so it can be quickly used to reference each other quickly and efficiently. The way the linking was achieved was having a class wrapper which had the two markers together, and then stored in an array. The container also contains both of the marker's line number for ease of access to each marker. The reason that the markers were linked together was to easily discover each other and also so that it can be deleted together. Another reason was so that it can suggest to the user where the data race pair is located.

3.3.4 Quick-Fix

There are two suggestions per data race marker in the plug-in. These suggestions are also known as quick-fix and allows the user to quickly identify or remedy the problem.

The first quick-fix suggestion allows the user to go to the other marker which is its data race pair. This quick navigation saves the user time when trying to find the other problem pair. The next quick-fix allows the user to delete the marker, giving them the freedom to free up the clustering of markers in the workspace. Although not an intuitive thing to do, this becomes an apparent feature to have considering the design limitation of the plug-in that is discussed later on in the thesis.

Chapter 4

Evaluation

The goal of this thesis was to be able to statically detect data races in C/C++ programs, and to visualise these bugs in Eclipse. How we evaluate these outcomes is by firstly evaluating the tool, and then the plug-in.

For the tool, we examine how well it analyses programs, and how accurately it is able to detect data races. This is based upon output of the tool: what information it has obtained from the analysis, and how correct it is. This will be discussed in section 4.1.

For the plugin we judge how well it visualise these bugs based upon the intuitive nature of the plugin, how well it is integrated with the rest of eclipse and the display of the data races in the IDE.This will be discussed in section 4.2

4.1 Data Race Detection Tool

4.1.1 Objective

Data race detection and identification is the core of this thesis. Visualising the bugs by developing the Eclipse plug-in would not have been possible without this tool. In order to test the tool we ran it 5 different test cases.:

1. Parallel threads without locks
2. Parallel threads with locks
3. Sequential program
4. Parallel threads without locks across multiple files
5. Open source program with parallel threads

4.1.2 Test Environment

The system we were conducting our tests on our tool on consisted of an Intel Xeon CPU E5-2637 v4 @ 3.50GHz CPU with 64GB of RAM, running on Ubuntu 16.04.3 LTS operating system.

Setup

We recommend using Ubuntu 14.04 or above to alleviate any incompatibility issues. The tool we developed is available on GitHub. Once the code is downloaded you can build it running:

```
./build.sh
```

then to setup the environment:

```
. ./setup.sh
```

To generate the intermediate files from the source files you can use the LLVM gold plugin to easily generate bitcode files from single or multiple source files - setup instructions are available here. For single files, compile with:

```
clang(++) -flto -g source_file.c -o source_file.bc
```

For projects with a makefile, to generate the bitcode file you need make sure either:

```
CC="$PREFIX/clang -g -flto"
```

or

```
CCX="$PREFIX/clang++ -g -flto"
```

where `$PREFIX` is the location in which you installed clang with the LLVM gold plugin.

Once the bitcode files are generated and ready for analysis, run:

```
mta -mhp source_file.bc
```

4.1.3 Test Methodology

1. Convert source files into an intermediate file to be used by the tool. This is achieved by using the LLVM gold plugin which can be seen above.

2. Run the data race detection tool on the generated bit code files. This will perform the analysis on the program and generate and output file displaying the results of the analysis.

3. Analyse the output of the tool and compare it with the source code. As the goal the tool is to help assist in visualising the bugs, we need to check that the output matches the expected outcome of analysing the programs.

4.1.4 Test Results

We tested a number of different types of programs which was listed above. The aim was to analyse the output and compare it to the source code to see if it had correctly identified the data races.

Parallel threads without locks

The program we analysed can be seen in Figure A.1. It is very standard program in presenting how data races occur and how it can be fixed. In this case we leave locks out of it to test our program will correctly identify a program that we know has a data race. The expected data races should be located in line 11 and 18 as there is concurrent access to the global variable x. Below is the output from our tool:

```
11
example1.c
18
example1.c
11
example1.c
18
example1.c
```

Figure 4.1: Output from analysing example1.c

The output is as expected. The reason why the result is duplicated is of the way our tool analyses data races. It identifies there is a data race between line 11 and 18, and line 18 and line 11. That is why the result seems to be duplicated.

Parallel threads without locks

This program is a slight modification to the first example. It can be seen in Figure A.2. Now each operation on the global variable x is protected by locking operations which guarantees that it will not be accessed concurrently. The tool is not expected to output anything as there are no data races possible. The output of running the analysis is as expected and there is nothing.

Sequential program

This program is just a basic sequential program that can be seen in Figure A.3. As it is a sequential program no data race is possible. When this program is analysed by the tool it should not output any results and we received the expected output - nothing.

Parallel threads without locks across multiple files

This program is similar to the first example and can be seen in figures A.4 and A.5. It is basically example1.c but the increment, and decrement functions are spread to an operations file. This test case is to make sure the tool detects data races across multiple files and output information correctly identifies the file the issue originates from. The expected output should be identifying line 10 in example.c, and lines 8 and 15 in ops.c. Below is the output of the analysis of this program:

```
10
../src/example.c
8
../src/ops.c
10
../src/example.c
15
../src/ops.c
8
../src/ops.c
15
../src/ops.c
DUPLICATES REMOVED
```

Figure 4.2: Output from analysing example1.c

The tool outputs what we have expected. Once again there duplicates due to the reasons we discussed previously in the first program.

Open source program - parallel threads

The last program we analysed was the open source project git. Git is not entirely multithreaded but some modules within git are, such as fetch. So we analysed http-fetch.c and found no data races issues involved. This is expected as git is a widely used program and has been received continued developer support since 2005. Since it has been so widely used you can expect a lot of the bugs to be ironed out, such as data race issues.

4.2 Eclipse Plug-in

4.2.1 Objective

Integrating the SVF analysis tool onto the eclipse integrated development environment (IDE) was the final goal of this thesis. After further developing the SVF tool, the output of the analysis is used to link both the tool and eclipse together. From the output file, the markers for the eclipse plug-in can be created and manipulated in order to give the user information.

4.2.2 Test Environment

In addition to testing our plugin with the system described above, we also used the Java Oxygen Package which included the Eclipse IDE for java developers, Eclipse IDE for C/C++ developers and Eclipse IDE for Java EE Devleopers (a package for the development of the plugin tool). Furthermore, the testing and development of the plugin was conducted in Java SE Runtime Environment 8. The set-up for the plugin and other pre-configuration in of eclipse can be seen in our github repository and is located here GitHub.

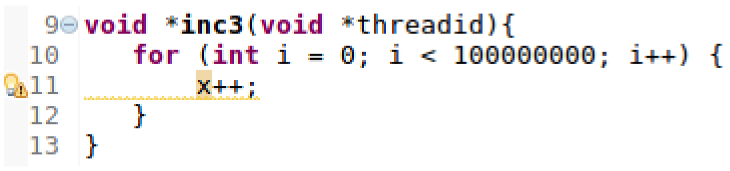
4.2.3 Test Criteria

1. Create the markers from the output file and visualise all the possible data race they present.
2. Create an intuitive interface for the user to use.
3. Create quickfixes which gives the user suggestions on how to deal with the marker.

4.2.4 Test Result

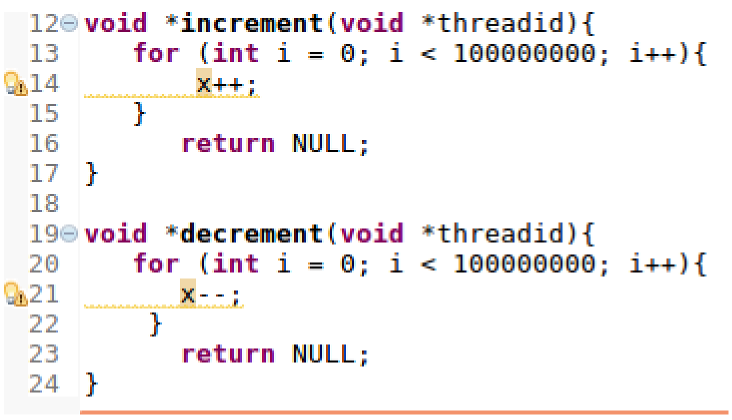
To demonstrate the inner workings of the eclipse plugin the code that is referenced in A.4 and A.5 will be used as the example code.

The following snippets of code visualises the potential data races that occur between example.c and ops.c. The snippets presents the markers on line 11 in example.c and line 14 and line 21 in ops.c, correctly displaying the data races reported by the SVF analysis.



```
 9 void *inc3(void *threadid){
10     for (int i = 0; i < 100000000; i++) {
11         x++;
12     }
13 }
```

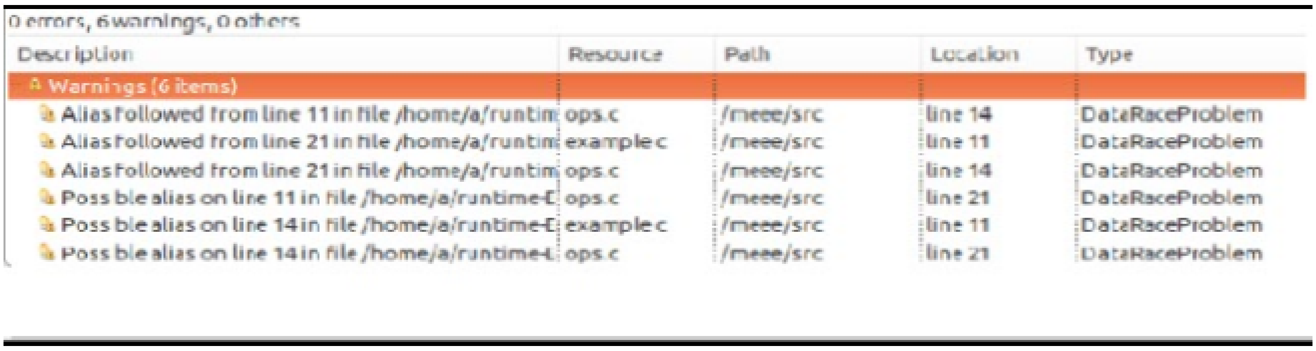
Figure 4.3: Error in example.c



```
12 void *increment(void *threadid){
13     for (int i = 0; i < 100000000; i++){
14         x++;
15     }
16         return NULL;
17 }
18
19 void *decrement(void *threadid){
20     for (int i = 0; i < 100000000; i++){
21        x--;
22      }
23         return NULL;
24 }
```

Figure 4.4: Error in ops.c

When the user scrolls to the problemview terminal, they will see all the errors that occurs inside the eclipse IDE. This outlines all the problem that occur in the source code, detailing the sort of error it is(dataraceproblem), the location file and line of the error and the problem pair inside the marker's description.

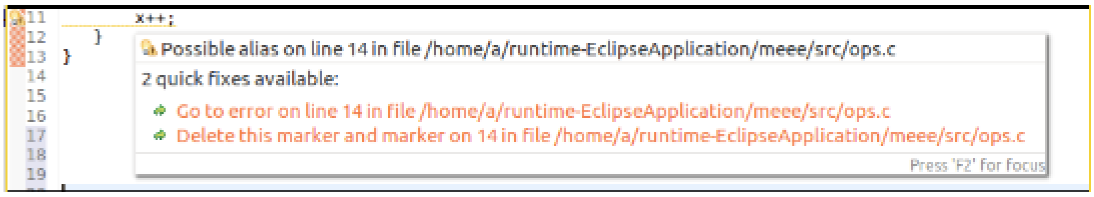


0 errors, 6 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (6 items)				
Alias followed from line 11 in file /home/a/runtim	ops.c	/meee/src	line 14	DataRaceProblem
Alias followed from line 21 in file /home/a/runtim	example.c	/meee/src	line 11	DataRaceProblem
Alias followed from line 21 in file /home/a/runtim	ops.c	/meee/src	line 14	DataRaceProblem
Possible alias on line 11 in file /home/a/runtime-E	ops.c	/meee/src	line 21	DataRaceProblem
Possible alias on line 14 in file /home/a/runtime-E	example.c	/meee/src	line 11	DataRaceProblem
Possible alias on line 14 in file /home/a/runtime-E	ops.c	/meee/src	line 21	DataRaceProblem

Figure 4.5: Problemview

When the user hovers over a marker, eclipse IDE automatically gives quick-fix suggestion on how to remedy the problem. The plugin does have suggestion for the quick-fix, indicated by the diagram below. It gives two option per marker, one is to go to the location of the other error to identify its data race pair, and the other is to delete the marker whenever the user wants.



```
        x++;
    }
}
```

Possible alias on line 14 in file /home/a/runtime-EclipseApplication/meee/src/ops.c

2 quick fixes available:

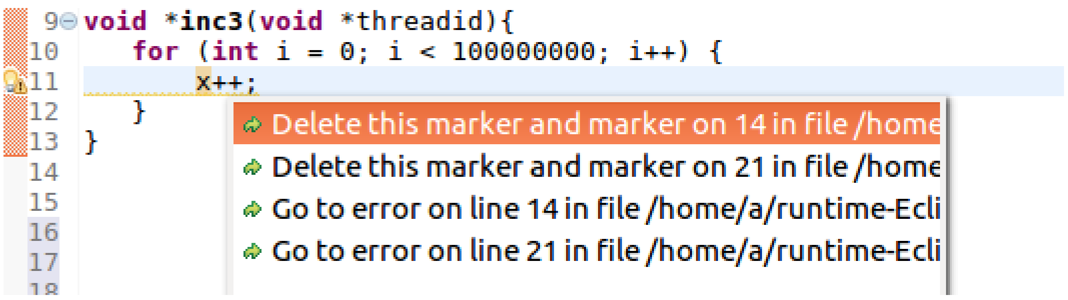
- Go to error on line 14 in file /home/a/runtime-EclipseApplication/meee/src/ops.c
- Delete this marker and marker on 14 in file /home/a/runtime-EclipseApplication/meee/src/ops.c

Press 'F2' for focus

Figure 4.6: quickfix

As explained, the quick-fix'Go to error on line 14 in file /home/a/runtime-EclipseApplication/meee/src/ops.c' will allow the user to directly jump into the ops.c file without have to navigate the eclipse. This saves the user time to manually go through (potentially) different files in different folders to find out where the data race occurs. This option was intended for user to save time in order to search for the error. The next quick-fix was to delete the marker, and the user has control over when to delete the intended marker. This feature was implemented because of the possible clustering of markers in the fille, as a precaution, the plug-in allows users to delete the marker if they think that they have fixed the marker. As the Data Race Detection Tool is only ran when the user calls for the run build to refresh and marker all the problem area, this feature was needed as it doesn't track the errors on the fly, as most other error handler.

The last feature is a hidden feature in eclipse. As the amount of errors that may appear on the marker could cause a clustering of information for the user, it was designed so that an error could only appear one a at when right-clicking a marker. However, if the user wants to see all errors on that line, and want to go to a quickfix line other than the first error that appears, they have the ability to do this. This is demonstrated in the diagram below and can be viewed what they hover over the marker and hit the ctrl + '1' key on their keyboard. This reveals all other markers in the line and gives the options of all the quick-fix pertaining to that line.



```
void *inc3(void *threadid){
    for (int i = 0; i < 100000000; i++) {
        x++;
    }
}
```

Delete this marker and marker on 14 in file /home
Delete this marker and marker on 21 in file /home
Go to error on line 14 in file /home/a/runtime-Ecli
Go to error on line 21 in file /home/a/runtime-Ecli

Figure 4.7: Error in ops.c

Chapter 5

Conclusion

The purpose of this thesis was to design and implement a tool to assist in debugging data races in C/C++ programs. A plug-in was developed utilising this tool to help visualise the bugs so developers could easily identify and debug these issues. This tool was built upon SVF's MTA module and is available on github.

We conducted research of other works to understand concepts and methods into detecting data races. We used that knowledge to determine which methods would be most appropriate for us. Throughout this, we explored data race detection methods utilised by different tools, and for different languages. Based upon this, we decided to use flow-sensitive analysis to detect data races.

To help visualise these bugs, we decided to develop the plug-in for the Eclipse integrated development environment (IDE). This was due to Eclipse being one of the most popular free IDE's with extensive plug-in support. Eclipse is also the IDE that we are most familiar with so we understand the standard functionality that Eclipse provides so we could development functionality that is visually consistent with Eclipse.

During the tool and plug-in's evaluation, we were able to statically detect all data races with few issues, which will be discussed in the next section. Data races were able to be identified by our tool, and visualised by our plug-in. This allowed users to easily identify data races within their programs, and allowed them to easily debug them.

5.1 Limitations

5.1.1 Data Race Detection Tool

One of the main limitations of this tool is the size of the programs it can quickly analyse. As we previously discussed, data races are identified by analysing instruction pairs. Pairing up n store load and store created $^{n}C_2$ which means the algorithm we use to pair up instructions and analyse them is at worse an $O(^{n}C_2)$ function. Naturally this would mean the function does not scale well at all. This is the root of the cause of the issue. We did implement a function to skip pairing instructions which would make it at best an $O(n)$ function, but on average this function isn't enough to decrease the number of computations required for the whole analysis.

Another limitation is that the LLVM gold plugin is only available for a Linux environment. This would mean that you'll only be able to compile the bit code files (from multiple sources file) required for analysis on a Linux environment.

5.1.2 Eclipse Plug-in

A major limitation of the eclipse plugin was the fact that the Data Race Detection Tool had to be ran separately in order to get an output file for the markers to generate. This meant that the eclipse editor can only update the errors when the project is rebuilt, which is not an intuitive design in eclipse as markers and errors are constantly being updated upon saving the file. This flaw stems from how the tool may take a long time to run especially for a lage codebase with many store-load instructions. To compromise, the delete feature for the marker was implemented, allowing the user to remove pairs that they have fixed and can check if they have fixed it by rebuilding their project.

Another limitation with the plugin is the lack of quick-fixes given to the user. This stems from how giving an actual fix to the user at that marker location may not be viable, for instance if two marker pairs were given the quick-fix of surrounding the code by a lock, that may cause a deadlock, causing more issues rather than fixing anything. This further the argument for having a delete marker quick-fix. When the user think that the data race has been cleared up, then rebuilding the project may clear up the data race.

A hidden limitation of the plugin is that the output file is placed inside the project folder of the source code upon the completiong of the data race tool analysis. This mean that if the name of the file is the same as the output file then it would be deleted. This is a big flaw coding, but was the only reasonable way to store the output of the data race detection tool and not have it read into memory of the user.

5.2 Future Work

5.2.1 Data Race Detection Tool

If this tool was to be improved in the future, the most important aspect will to be optimizing the pairing and analysis phase of the tool. Once that has been improved then the tool will be able to quickly analyse much larger programs.

Adding more features to the Eclipse plug-in would require more improvements and features to be added to the data race detection tool. This could include outputting the call graph to allow users to see the calling relationships between the data races.

5.2.2 Eclipse Plug-in

An improvement to the plugin would be having more quick-fix options in the future for users to identify the issue at hand. Although this may cause cluttering of the editor, it may serve as a good way for the user to get more from the plugin.

Another improvement would be finding a viable alternative to storing the output file to mark the data race pairs. One that does not store the output file in an obscure manner and deleting the file after the data has been scraped.

Integrating anymore sizable features to the plugin may also mean improving the tool itself, but if this mean that the user can find the data race and fix it easier, then it will be worth investing time into.

Bibliography

[ANSI, 2011a] ANSI (2011a). Iso/iec 14882:2011. https://www.iso.org/standard/50372.html. Accessed: 11/05/2017.

[ANSI, 2011b] ANSI (2011b). Iso/iec 9899:2011. https://www.iso.org/standard/57853.html. Accessed: 11/05/2017.

[Comtemplate, 2009] Comtemplate (2009). Find java concurrency bugs with threadsafe. http://www.contemplateltd.com/threadsafe. Accessed 22-October-2017.

[Floyd, 1967] Floyd, R. W. (1967). Nondeterministic algorithms. *J. ACM*, 14(4):636–644.

[Naik et al., 2006] Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319.

[Oracle, 2010] Oracle (2010). 1.2 what is a data race? (sun studio 12: Thread anayzer user's guide. https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html. Accessed: 11/05/2017.

[Pratikakis et al., 2011] Pratikakis, P., Foster, J. S., and Hicks, M. (2011). Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55.

[Stroustrup, 2014a] Stroustrup, B. (2014a). C++ applications. http://www.stroustrup.com/applications.html. Accessed 7/5/2017.

[Stroustrup, 2014b] Stroustrup, B. (2014b). Lecture: The essence of c++ - university of edinburgh. https://www.youtube.com/watch?v=86xWVb4XIyE. Accessed 7/5/2017.

[Sui et al., 2016] Sui, Y., Di, P., and Xue, J. (2016). Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 160–170, New York, NY, USA. ACM.

[Wen et al., 2007] Wen, V. J., Ranjit, J., and Sorin, L. (2007). 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. ACM.

[Williams, 2012] Williams, A. (2012). *C++ Concurrency in Action.* Manning Publications, Greenwich, Connecticut, 2nd edition.

Appendix 1

This section contains some of the programs the data race detection tool tested.

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

// basic data race example

int x = 0;

void *increment(void *threadid){
   for (int i = 0; i < 100000000; i++){
     x++;
    }
      return NULL;
}

void *decrement(void *threadid){
   for (int i = 0; i < 100000000; i++){
      x--;
    }
      return NULL;
}

int main(){
   pthread_t threads[2]; //using just two threads
   pthread_attr_t attr;
   void *status;

   // initialise and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   pthread_create(&threads[0], NULL, increment, (void *)1);
   pthread_create(&threads[1], NULL, decrement, (void *)2);
   pthread_join(threads[0], &status);
   pthread_join(threads[1], &status);
   printf("%d\n", x);

   return 0;
}
```

Figure A.1: Basic data race example without locks - example1.c.

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

int x = 0;
pthread_mutex_t mutex_x = PTHREAD_MUTEX_INITIALIZER;

void *increment(void *threadid){
   printf("incrementing...\n");
   long tid;
   tid = (long) threadid;
   pthread_mutex_lock(&mutex_x);
   for (int i = 0; i < 100000000; i++) {
     ++x;
   }
   pthread_mutex_unlock(&mutex_x);
   return NULL;
}

void *decrement(void *threadid){
   printf("decrementing...\n");
   long tid;
   tid = (long) threadid;
   pthread_mutex_lock(&mutex_x);
   for (int i = 0; i < 100000000; i++) {
     --x;
   }
   pthread_mutex_unlock(&mutex_x);
   return NULL;
}

int main(){
   pthread_t threads[2]; //using just two threads
   pthread_attr_t attr;
   void *status;

   // initialise and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   pthread_create(&threads[0], NULL, increment, (void *)1);
   pthread_create(&threads[1], NULL, decrement, (void *)2);
   pthread_join(threads[0], &status);
   pthread_join(threads[1], &status);
   printf("%d\n", x);

   return 0;
}
```

Figure A.2: Basic data race example with locks - example2.c

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

int x = 0;

void decrement(){
    x--;
}

int main(){
    for (int i = 0; i < 100000000; i++){
        x++;
    }
    for (int i = 0; i < 100000000; i++){
        decrement();
    }
   printf("%d\n", x);
   return 0;
}
```

Figure A.3: Basic sequential program example - example3.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "include/ops.h"

int x = 0;

void *inc3(void *threadid){
   for (int i = 0; i < 100000000; i++){
     x = x + 3;
   }
      return NULL;
}

int main(){
   pthread_t threads[3]; //using just two threads
   pthread_attr_t attr;
   void *status;

   // initialise and set thread joinable
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   pthread_create(&threads[0], NULL, increment, (void *)1);
   pthread_create(&threads[1], NULL, decrement, (void *)2);
   pthread_create(&threads[2], NULL, inc3, (void *)3);
   pthread_join(threads[0], &status);
   pthread_join(threads[1], &status);
   pthread_join(threads[2], &status);
   printf("%d\n", x);

   return 0;
}
```

Figure A.4: Data race example across multiple files - example.c

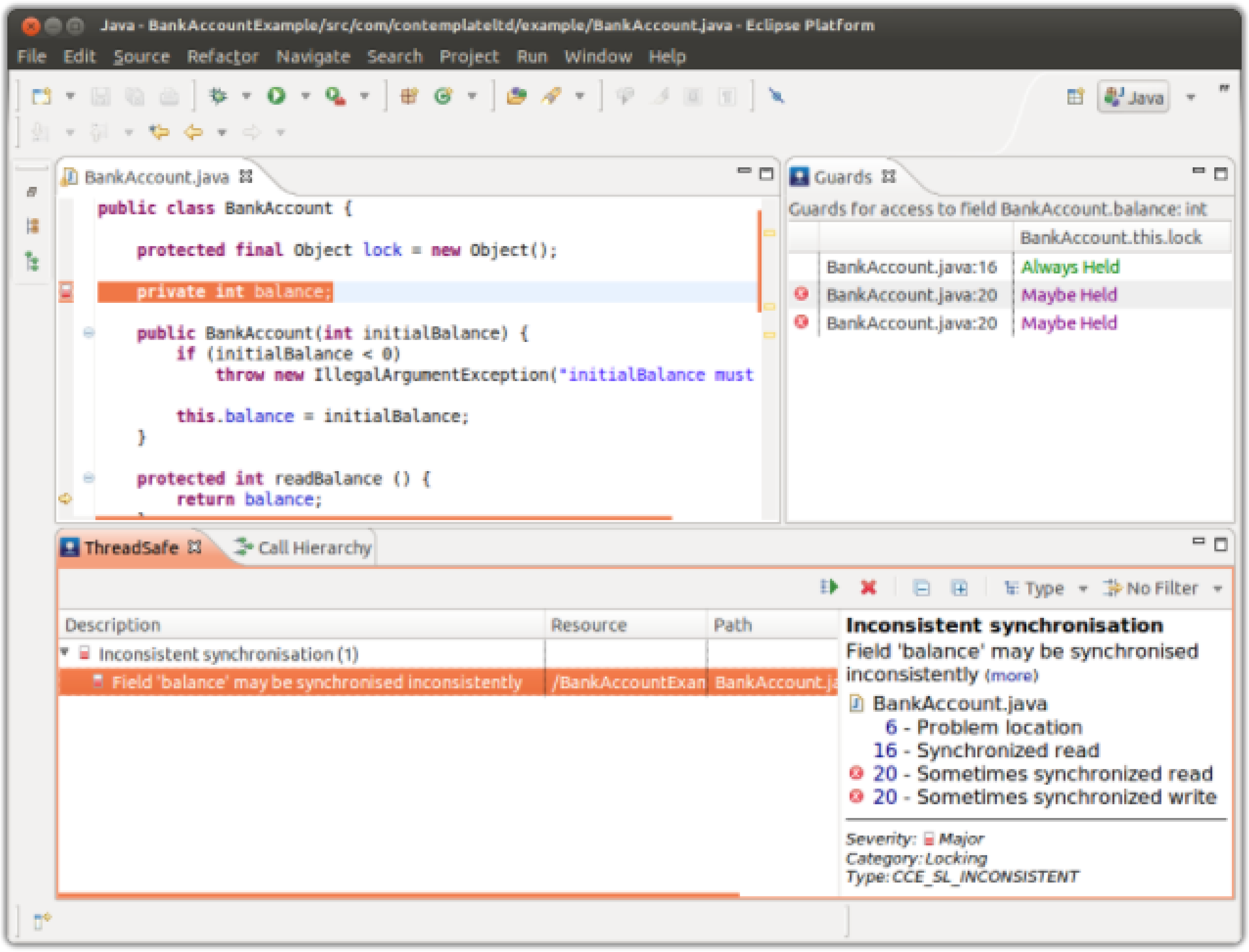
```c
#include <stdlib.h>
#include <stdio.h>

extern int x;

void *increment(void *threadid){
   for (int i = 0; i < 100000000; i++){
     x++;
    }
      return NULL;
}

void *decrement(void *threadid){
   for (int i = 0; i < 100000000; i++){
      x--;
    }
      return NULL;
}
```

Figure A.5: Data race example across multiple files - ops.c


Java - BankAccountExample/src/com/contemplateltd/example/BankAccount.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Java
BankAccount.java
Guards
ThreadSafe
Call Hierarchy
Type
No Filter

```
public class BankAccount {

    protected final Object lock = new Object();

    private int balance;

    public BankAccount(int initialBalance) {
        if (initialBalance < 0)
            throw new IllegalArgumentException("initialBalance must

        this.balance = initialBalance;
    }

    protected int readBalance () {
        return balance;
```

Guards for access to field BankAccount.balance: int

	BankAccount.this.lock
BankAccount.java:16	Always Held
BankAccount.java:20	Maybe Held
BankAccount.java:20	Maybe Held

Description	Resource	Path
Inconsistent synchronisation (1)		
Field 'balance' may be synchronised inconsistently	/BankAccountExan	BankAccount.ja

Inconsistent synchronisation
Field 'balance' may be synchronised inconsistently (more)
BankAccount.java
6 - Problem location
16 - Synchronized read
20 - Sometimes synchronized read
20 - Sometimes synchronized write

Severity: Major
Category: Locking
Type: CCE_SL_INCONSISTENT

Figure A.6: ThreadSafe Eclipse Plugin (Contemplate 2009)