Heap Reference Analysis for Concurrent Programs

B.Tech. Project 2nd Stage Report

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology (Honors)

Student:
Anshul Purohit
Roll No: 110050002

Guide:
Prof. Uday Khedker

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

This report mainly deals with designing heap reference analysis for concurrent programs in Java. The analysis is both flow-sensitive and context-sensitive at the inter-procedural level. Heap Reference analysis determines a collection of objects pointed to by program variables or their fields. Java implements pointers by references. This report also discusses about analysis techniques for concurrent programs at intra-procedural level and about extending analysis to inter-procedural level using the VASCO tool. In addition to this, this report introduces the notion of handling of thread switching and thread contexts while performing analysis over a data-race free multi-threaded program. This leads to a more precise analysis, as data flow fact propagation along spurious program execution paths are avoided.

Contents

Chapter 1

Introduction

1.1 Background on Program Analysis

Program Analysis is the set of techniques to get appropriate information about the behavior of programs. Static analysis techniques involve computing approximate information about the program without executing it. In this report, only static analysis techniques would be discussed.

Data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a program. The value at each program point is then propagated in the Control Flow graph (CFG) of the program. The information gathered is often used by compilers when optimizing a program.

1.2 Types of Data Flow Analysis

A data flow analysis can be either flow-insensitive or flow-sensitive.

Flow-insensitive analysis:

- Ignores the control-flow graph, and assumes that statements can execute in any order.
- Rather than producing a solution for each program point, produces a single solution that is valid for the entire program

Flow-sensitive analysis:

- Takes the control flow structure of a program into count

- Has the computed abstract state that represents different reachable memory states at different program points.

A flow-sensitive version of an analysis is more precise and expensive than the flow-insensitive version.

Categorizing according to the direction of traversal of program statements in the CFG, data flow analysis can be broadly of 2 types : forward and backward. Examples of data-flow analysis are Available Expressions analysis which is a forward data flow analysis and Live Variables analysis which is a backward data flow analysis.

1.3 Inter-Procedural Analysis

Inter-procedural analysis operates across an entire program makes information flow from caller to callee and vice-versa. It extends the scope of data flow analysis across procedure boundaries and it incorporates the effects of procedure calls in the caller procedures, and calling contexts in the callee procedure. A context-sensitive analysis is an **interprocedural analysis** which reanalyzes callee procedure for each context whereas context insensitive analysis performs analysis independent of calling context.

1.4 Concurrency

For concurrency, we assume the thread model in Java. We use the model of threads to uniformly refer to concurrent programs. The user needs to extend the thread class and define a run method for the object. Invoking the run method on the thread objects starts the multi-threaded concurrent execution. For accessing shared data, critical sections need to be guarded by the lock and unlock statements. Also we expect a data-race free program as input for analysis.

1.5 Problem Statement

The idea of the problem statement is to design a framework for carrying out inter-procedural analysis of concurrent programs and then implement heap reference analysis using the framework. The techniques for performing inter-procedural and concurrent analysis is discussed later in the report.

Heap reference analysis refers to determining the information like liveness, accessibility and points-to information of reference expressions. Reference expressions, for example *x.lptr.rptr.data* are primarily used to access the objects in the heap. The main focus is on the java model of heap access, in which root variables (variable x is on the stack) are stored on the stack. The root variables represent references to memory in heap. Also, root variables cannot be pointed to by any reference.

1.6 Organization of the Report

In this chapter, an introduction to the basics of Data Flow Analysis and its types were described. I have also mentioned about concurrency model and heap accesses for Java.

In Chapter 2, I will be discussing about heap reference analysis. This is an analysis defined over sequential programs. It aims to find out statically the live access links on the heap memory at any program point.

In Chapter 3, I will discuss about inter-procedural analysis techniques. Mainly will be focusing on the value contexts method. Will also mention about other approaches like functional and call strings.

In Chapter 4, I will present an analysis technique for threaded concurrent program execution. It adds synchronization edges across critical sections and runs a simple analysis over the synchronized cfg.

In Chapter 5, some problems with the concurrent analysis technique will be presented with a few examples. The reasons causing imprecision will be discussed.

In Chapter 6, improvements to the simple sync-cfg technique will be discussed. A modified technique will be presented that takes into account the execution of threaded program.

Chapter 7 presents some tools, like VASCO, CombinedUnitGraph & Call Graph, that can be used for implementation of sync cfg analysis.

Chapter 8 presents the conclusions from the report and discusses future work that can be continued.

Chapter 2

Heap Reference Analysis

Analyzing properties of heap data is not very trivial. This is because the spatial and temporal structure of stack and static data is simple to understand. The stack variables have a compile-time name(alias) associated with it. However, this is not the case with heap data. We need to devise a flow and context sensitive analysis to get information from heap data.

2.1 Difficulties in Analysis of Heap Data

A program accesses data through expressions having l-values and hence are called access expressions. The l-values can either be a scalar (x), or may involve array access such as *a[2*i]* or can be a reference expression like $x.l.data$. In the case of reference or array, the mapping of the access expression and the l-value may change. The reference expression is primarily used to access the heap.[1]

Heap analysis tries to find out the answer to the questions:

- Can an access expression a_1 at program point p_1 have the same l-value as access expression a_2 at program point p_2.
- Can there exist objects in the heap that will not be reachable from the access expressions?
- Which of the access links will be live at a particular point?

```
class A(){}                          public void set(B q){
class B(){                           this.g = q;
public A f;                          }
public void set(A p)                 }
{                                    s1 : A x = new A()
this.f = p;                          s2 : B y = new B()
}                                    s3 : C z = new C()
}                                    s4 : y.set(x);
class C(){                           s5 : z.set(y);
public B g;                          s5 : A a = z.g.f;
```

Figure 2.1: Code to illustrate heap access and points-to analysis

2.2 Pointer Analysis

Pointer analysis is a static analysis technique that establishes which pointers or heap references can point to which variables. Pointer analysis collects information about indirect accesses in programs. It can enable precise data analysis and precise inter-procedural control flow analysis. The latter will be used in the VASCO tool that will be discussed later in the report.

We have two types of pointer analysis information: points-to analysis and alias analysis. Alias analysis tell if two references point to the same location on the heap. It is transitive in nature. Whereas, points-to analysis tells which memory locations are pointed by references at run-time.

Alias information plays an important role in the liveness analysis. Must-alias information is needed to improve precision. May aliases can be found out using information from the points-to analysis.

In Java pointers are not created explicitly.[2] All objects in Java are accessed using references and these references are termed as pointers here. Every time we create an object in Java it creates pointer to the object. This pointer could then be set to a different object or to null, but the original object will still exist. Thus points-to analysis for Java programs identifies the objects pointed to by references at run time. Thus we wish to determine the objects pointed to by a reference variable or a field. Consider the Java program in Figure 2.1.

In the code given in Figure 2.1, three heap objects pointed to by x, y and

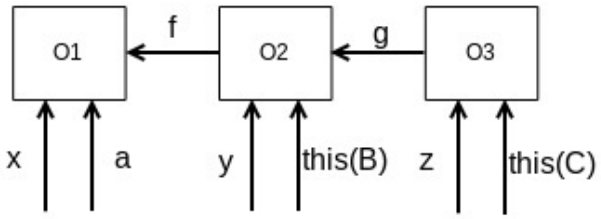

Figure 2.2: Points to graph for Java program in 2.1

z are created at s_1 , s_2 and s_3 respectively. We refer to the objects based of their allocation site as o_1, o_2 and o_3. The statement s_4 assigns the f field of o_2 to point to o_1. Similarly, the g field of object o_3 is pointed to object o_2. Finally the variable a points to the object o_1, through reference field indirections.

Liveness and Points-to Analysis : We can remove some information from the points-to analysis result by considering only information for live pointers. For field references and indirections, liveness is defined using points-to information. [3]

Points-to graph in Java contain two types of edges. The first type of edge is to represent the information that reference variable v is pointing to object o. The second type of edge represents the field f of o_1 pointing to o_2. Example of points to graph for the code in Figure 2.1 is shown if Figure 2.2.

2.3 Heap Reference Analysis

A reference can be represented by access path. In order to perform liveness analysis of heap and identify the set of live links, naming of links is necessary. This is achieved by access path. An Access Path is defined as root variable name following any number of field names and is represented as x $\rightarrow$ n1 $\rightarrow$ n2 ....nk where x is root variable, n1 , n2 .. are field names. If access path $x \rightarrow f \rightarrow d$ is live then, the objects pointed to by x, $x.f$ and $x.f.d$ are live. Example of access path is given for the expression *x.left.right.data* in figure 2.3

An access path can be unbounded in the case of loops. Thus, we require to set a bound on the representation of access paths for liveness information. This is achieved using access graphs which summarizes information based on

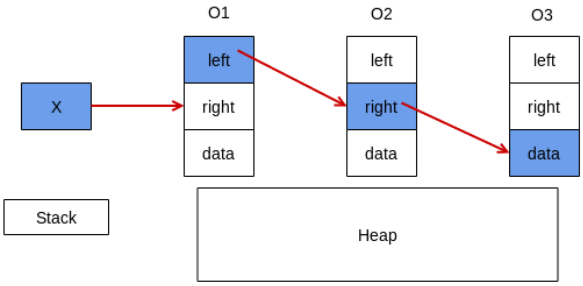

Figure 2.3: Heap reference using access expression *x.left.right.data*.

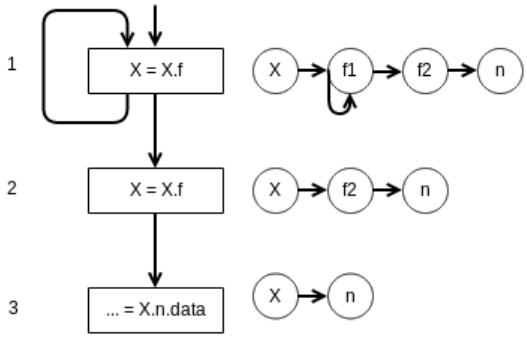

Figure 2.4: Example of use of access graph and liveness data flow values

allocation sites. Access Graph is a directed graph representing access paths starting from root variable. Root node is connected to any number of nodes each having unique labels of form n_i where n is name of the field and i is the program point. Inclusion of program points in access graphs helps in summarization and performing the merge operation.[4] Example of access graph and liveness analysis is shown in Figure 2.4.

Availability and Anitcipability analysis of heap data : An access path ρ is said to available at a program point p if the target of each prefix of ρ is guaranteed to be created along each path reaching p. An access path ρ is said to be anticipable at p if the target of each prefix of ρ will be dereferenced along every path starting from p. Note that access graphs are not needed to carry out to carry out available and anticipable analysis over heap data because the sets are bounded as as result of every control flow path nature of the problems.

Chapter 3

Inter-Procedural Analysis

Inter-procedural Analysis is required to obtain more precise results as it is very common that programs can have multiple function calls. It is essential to consider the effect of function call on the data flow value entering the node. Inter-procedural analysis takes into account call return , parameter passing , local variables of the function, return values and recursion into account. Major issue to be dealt while handling inter-procedural analysis is to deal with calling contexts. Handling concurrency also needs to be supported , which would be discussed in the next chapter.

3.1 Context Sensitivity

A context sensitive analysis is an inter-procedural analysis which analyses callee procedure for each context whereas context insensitive analysis performs analysis irrespective of calling context. Context insensitive analysis over-approximates inter procedural control flow which results in imprecision because it takes into account invalid control paths. Each function is analyzed once with single abstract context. Whereas context sensitive analysis is more precise as it considers the valid inter procedural control flow.

Java is an object oriented language supporting features like encapsulation and inheritance. Thus data access is indirect through method calls for each class. So context sensitivity plays a very important role for Object Oriented languages.

3.2 Call Graph

Call Graph is graph with nodes and edges in which nodes represent procedures and there is edge from a to b if some call-site at a calls procedure b. Hence this is a static data structure that represents the run-time calling relationships among procedures in program. Soot provides a Spark engine which generates the call graph. VASCO on the other hand returns a much precise call graph, which is generated using liveness based inter-procedural pointer analysis. A thing to note here is that construction of call graph requires inter-procedural analysis and inter-procedural analysis on the other hand requires call graph. An approach to breaking this dependency is to initially approximate the call-graph and in every iteration perform inter-procedural analysis and improve the precision of the call graph.[5]

Call multi-graph is a directed graph which represents calling relationships between procedures in a program where nodes represent procedures and edges procedure call. A cycle in the call multi-graph denotes recursion. Super graph is another representation in which callsites are connected to the callee procedure entry node and and the exit node of the callee is connected to return node in the caller procedure.

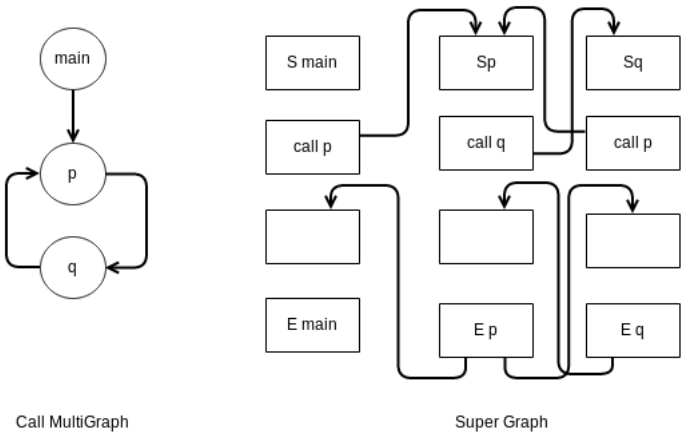

Figure 3.1: Example of call multi-graph and super-graph

Data flow analysis uses static representation of programs to compute summary information along paths. For ensuring safety, all the valid paths must be covered. A valid path is the path which represents legal control

flow. Ensuring precision is subject to merging data flow values at shared program points without including invalid paths. For ensuring efficiency, only those valid paths that yield information that affects the summary information should be covered.

3.3 Approaches to Inter-procedural Analysis

In this section approaches to perform inter-procedural analysis is discussed. A very simple approach is to perform procedure in-lining where every procedure call is replaced by the procedure body. This would however only be applicable when target of the call is known and call is not made by pointers or is virtual. However this is not good way to handle recursion as the code size can increase in an unbounded manner.

3.3.1 Functional approach

In the functional approach, summary flow functions are computed for each function. The summary flow functions are used as the flow functions for the procedure call. The summary flow function of a given procedure is influenced by the summary flow functions of the callees of r and not by the callers of r. Also in the presence of loops or recursion, iterative computation will be needed till fixed point is achieved. Termination is only guaranteed if, lattice is finite.

3.3.2 Call Strings Approach

This is a general flow and context sensitive method. In this approach the call history is stored for information to be propagated back to the correct point. Call string at a program point is the sequence of unfinished calls reaching that point starting from the main procedure call. The data flow equations are changed to incorporate the merging of the data flow values only if the contexts(call strings) are the same. At a call node c_i , c_i is appended to the call-string value at that point. Similarly at a return node the last call site ci is removed. And other data flow values are blocked. For non-recursive programs number of call strings are finite. For recursive programs, number of call strings can be infinite. However, the problem is decidable for finite lattices.

There is an approach of value based termination of call-strings. This method deals with creating equivalence classes. If two call-strings have same

data flow values at the start node of the procedure, then they will produce the same data flow values at the return node of the procedure call. Such call strings are grouped into equivalence classes.

An very simple example of call strings method is shown below.[2]

```
public static void main ()
{
B x;
// Callsites are appended to the current call string before making call f and g
C1 : x = after(f);
C2 : x = after(g);
}
B after(B a) // c1 [a = f] c2 [a = g]
{
return a.next;
// C1 [return f.next] C2 [return g.next]
}
// c1 [ object return would have its next pointed to f]
// c2 [ object returned would have its next field pointed to h]
```

3.3.3 Value Context Method

In this method, a combination of tabulation (functional approach) and value based termination (call strings) approach is adopted. The call-stings are partitioned based on the data flow value at the call site. And then analysis of the procedure can then be performed once for each partition. It also combines the two views of contexts: data flow values at call site are stored as value contexts and call strings as calling contexts. Distinct data flow values are maintained for each context of a procedure.[5]

A value context is defined by a particular data flow value reaching a procedure. It is used to enumerate and store the summary flow function of the procedure in terms of input and output pairs. In order to compute these pairs, data flow analysis is performed within the procedure for each context(input data flow value). The out value of each context is initialized with the top element. This approach also maintains a context transition table which allows flow of information along inter-procedurally valid paths. Transitions are recorded in the form $((X,c)\ ,\ Y)$ where X represents calling context, c represents call site and Y represents callee context. Hence when analysis of callee procedure completes where to return is identified by context

transition table. Therefore propagation to valid path is guaranteed.

When a new call to a procedure is encountered, the context table pairs are consulted to decide if the procedure needs to be analyzed again. If it was already analysed once for the input value, output can be directly processed else a new context is created and the procedure is analysed for this new context.

3.4 Example of value context method

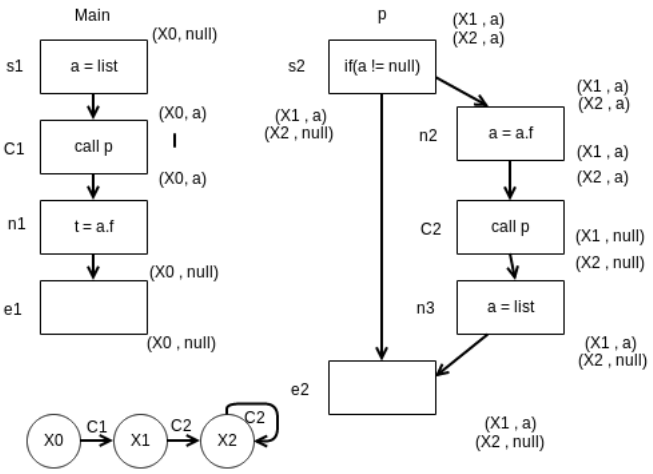

Figure 3.2: Example of value-context based IP analysis

This is an example of an inter-procedural heap liveness analysis. We wish to find out if a is live before and after the statement C_1 The procedure p is recursive and gets called with two different contexts X_1 and X_2 as shown in the context transition diagram as well.

The analysis starts with the intial context X_0 at statement e1 in main with the value null . The in value of n_1 is becomes a as $a.f$ is being set to t in the statement. On the next node C_1, we have a function call to p. So a new value context X_1 is created with input a and value $null$ and the transition from X0 to X1 is noted. Now in X_1 context, node n_3 is processed followed by c_2, which creates another value context X_2 for procedure p with input $null$ with value $null$. The transition X_1 to X_2 is also recorded in the

context transition table.

In the context X_2, node c_2 is evaluated. Since there is already a context for input value *null* we take its out value to be *null* (the top value of the lattice). Thus, the in values of C_2 and n_2 are set to *null* and a respectively. Since s_2 does not kill the access path a, the output value of context X_2 is set to a.

The callers of X_2 are X_1 and X_2 itself. Both these contexts are now added to the work-list for processing. The only change for context X_2 is in the call statement c_2 whose in value is now set to a. On evaluating context X_1, we similarly get its output value to be a. After this step, the statement C_1 in context X_0 gets the in value assigned to be a, which gets killed in s_1.

Value based contexts are used as a cache table for distinct call sites apart from terminating analysis of recursive procedures.

Chapter 4

Analysis for Concurrent Programs

We will be using the technique mentioned in the paper Dataflow Analysis for Datarace-Free Programs[6]. This technique when given a sequential data-flow analysis produces an efficient and fairly precise analysis for concurrent programs. The criteria to be met for applying this analysis is that data-flow fact should be dependent on the contents of the associated lvalues (expression referring to memory location at runtime). Sequential analysis like null-pointer analysis, interval analysis and constant propagation. In terms of precision of the data-flow facts, useful information will be derived at program points where the lvalue is read.[6]

The main challenge in converting the analysis from sequential to concurrent programs is that propagating data-flow values such that all the possible concurrent executions of the thread are taken care of. In this technique, the synchronization structure of the program is made use of to propagate data-flow values. The main insight used is that the data-flow values are only propagated between threads at the lock and unlock points in threads for access to critical sections. Also this approach will not work for programs containing data race.

4.1 Memory Model and Data Races

We will model the notion of concurrency in programs using threads at the moment. The shared variables in the thread will be accessed inside a pair of lock and unlock statements. Note that, this only covers the case in which

only one thread can execute in the critical section. We have not modeled the case of general semaphores in which multiple threads can execute inside the critical sections.

A memory model specifies the interactions of threads with memory and its shared use. Thus , memory model specify the constraints on data access and conditions on how data written by one thread is accessible to other threads. Happens before is the memory model described in the paper and thesis on Concurrent Program analysis.

The happens before memory model is based on the happens-before relation which relates the happenings of two events such that if one happens before the other, it should be reflected in the results. That is if statement A occurs before statement B then the memory written by statement A is visible to statement B.
There are three components of the happens before memory model.

1. Program Order: It is generally used to refer to the order of statement in a thread as they appear in the program. However the formal definition of program order is the intra-thread order during the execution of the program.

2. Synchronizes-with Relation: This relation is defined over synchronization relations which are lock, unlock (for acquiring/releasing locks), spawn, join (to synchronize creation/endind of multiple threads), start, end(first/last statement of a thread). The synchronizes-with relation is defined from

   - Lock statement to all previous unlock statements
   - A begin statement to the spawn statement of the thread
   - Join statement to the end statements of all the threads synchronized

3. Happens-Before Order: Statement a happens before statement b if one of the following hold

   - a appears before b in the program order
   - b synchronizes-with a
   - b can be reached transitively using happens-before relation from a.

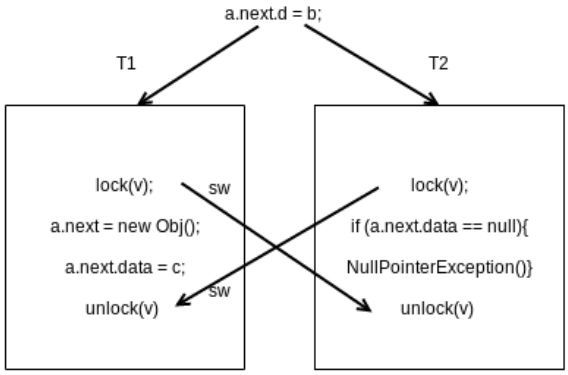

Figure 4.1: Happens Before memory model with thread synchronization

Consider the example presented in Figure 4.1. We have an example of 2 threads executing the critical section. Assume that *a.next* was initially pointing to an object o_i on the heap which contains non-null data. We need to check if the exception in T_2 can be thrown. Applying the happens-before relation, we can find out this will never be the case. The synchronization of locks and unlocks always prevents the possibility of *a.next* pointing to a dynamically created object with *null* when the *if* condition in T_2 is checked, irrespective of any possible thread scheduling. The exception would only have been raised if the statement 2 of T

Now consider the case given in Figure 4.2. It has the same statements but does not contain proper locking and hence no synchronization edges. So in this case all the actions in T1 need not happen before the evaluation of the condition. The exception can now thrown be in case the execution order is (T_1,$s1$),(T_2,$s2$). This necessitates data race free program as input to the analysis.

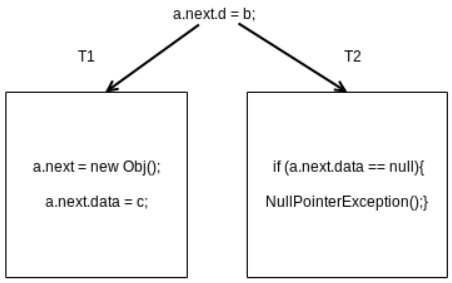

Figure 4.2: Happens Before memory model without thread synchronization

Data Race : A data race occurs when two or more threads can access shared data at the same time and try to change it at the same time. Also the order of access among the threads will not be known, so both the threads are racing to access or modify the shared data. Formally, a data race is said when

1. Two or more threads access the same memory location concurrently
2. At least one of the accesses is a write
3. he threads are not using any exclusive locks to control their accesses to that memory.

Data race can be defined in terms of happens before relation. Two statements are said to be conflicting/racy if neither a happens before b or b happens before a. We will only deal with programs which are data race free.

4.2 Concurrent analysis technique

Arnab De has explained his approach to concurrent data flow analysis[6] by giving example of access path based null-pointer analysis. He argues that the data-flow value is true only before the statement where the corresponding access path/lvalue is relevant.

The first step in the analysis is to add edges between nodes of control flow graph representing different threads. These edges map to the synchronize-with edges in the happens before memory model. Hence, the name sync-CFG is given to this control flow graph. The edges are added from the spawn statement to the first statement of each thread, from the unlock statement to lock statement if they access the same lock variable and belong to different threads. In the next step sequential data flow analysis is performed on the syn-CFG. The synchronization edges have identity flow functions. The example of the access-path based null pointer analysis inspired from[6] is in Figure 3.1

In Figure 3.1 the synchronization edges are added to generate the sync-CFG.It also shows access paths to memory locations which are discovered to be not null by the analysis. The values are marked in italics for each corresponding statement. As p.data is non-null at point M_5 in the main thread before spawning the cons thread, this value gets propagated to the

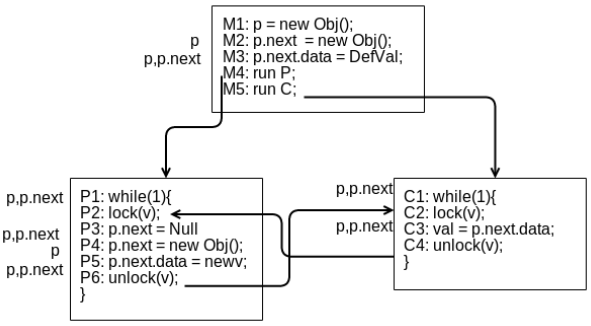

Figure 4.3: Heap Access path based null pointer analysis

first instruction C_1 of the cons thread though one of the added edges, and from there to the lock instruction at C_2. Similarly, although *p.next* is set back to non-null at P_4 before the unlock, despite being set to null at P_3 in the prod thread. This fact also gets propagated to the lock statement of the cons thread through the edge from P_6 to C_2. As p.next is not null in both the paths joining at C_2, we can conclude *p.next* to be non-null before the lock statement in all executions by the merge operation.

An important point to note about this technique is that, it may compute incorrect values at program points not containing access paths. For all data-race free programs, relevant statements will occur in the lock-unlock regions. Those statements containing access path or reference expression are considered relevant. In the given example C_1 is not a relevant statement and the data flow value *p.next* is incorrect considering the program execution order $[P_3\ C_1]$.

4.3 Concurrent Heap Liveness Analysis

Consider the Example shown in Figure 4.4, showing the first iteration. Heap liveness analysis is a backward flow problem. Figure 4.5, 4.6,4.7 shows the data flow values after iterations 2,3 and 4 respectively. At the beginning of s_1 in Thread 1, the live access path are *x.f1.r*, *x.f1.f2.r*. This is approximated by the access graph shown in Figure 4.7. Access graph representation is essential to represent finitely, the access paths(especially in the case of presence of loops). The synchronization edges introduce a structure similar to loops in the sync-cfg. Thus paths like *x.f1.f2.f1.r* and *x.f1.f2.f1.f2.r* are marked out to be live. We will obtain extra spurious live access paths by

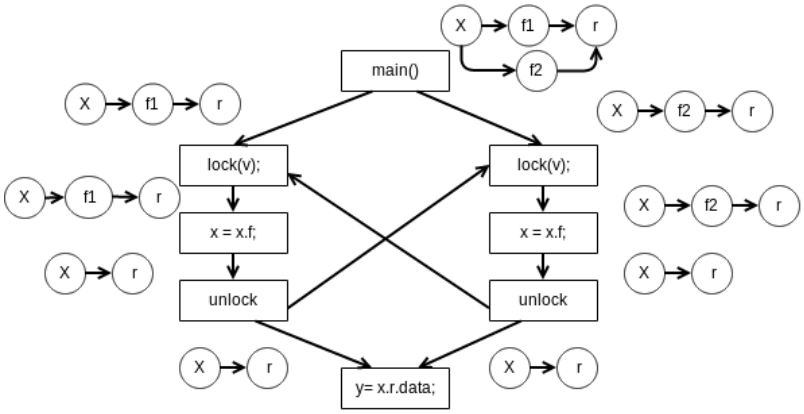

Figure 4.4: Concurrent Heap liveness analysis iteration 1

adding sync-edges, however the access graph will contain all the valid live paths at a point inside the lock-unlock statements. Also , note that at the start of the main statement, *x.f1.r*,*x.f2.r*. *x.f1.f2.r* and *x.f2.f1.r* paths are live.

Figure 4.8 shows the heap liveness analysis performed without adding the synchronization edges in the cfg. This will just mark out the access path *x.f1.r* live at the starting of s_1 in Thread 1. The absence of synchronization edges misses out the liveness of the access path *x.f1.f2.r* at the starting of s_1 in Thread 1. The access graph at the start of the statement main is same even without the effect of synchronization edges for this example. Even without synchronization edges,the access paths *x.f1.r*,*x.f2.r*. *x.f1.f2.r* and *x.f2.f1.r* are live.

Figure 4.9 illustrates the an input cfg that needs to be given as input to the concurrent heap liveness analysis. We also need to handle procedure calls inside the lock-unlock statements, using the value contexts method.

In the next chapter we will be discussing the problems with this analysis method. The fact that analysis without adding synchronization edges gives better liveness data flow value, highlights that there are some problems with this analysis. We will discuss them in greater detail in the next chapter.

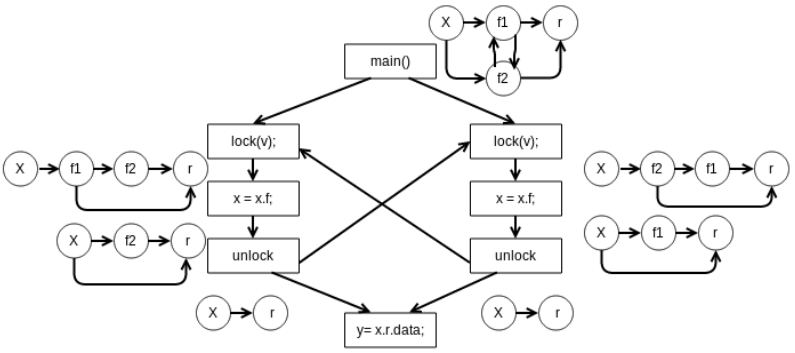

Figure 4.5: Concurrent Heap liveness analysis iteration 2

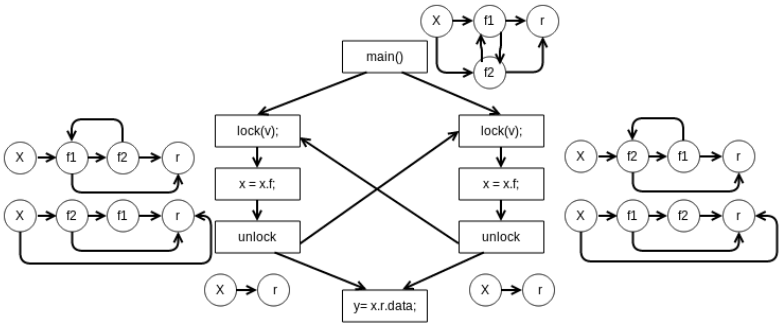

Figure 4.6: Concurrent Heap liveness analysis iteration 3

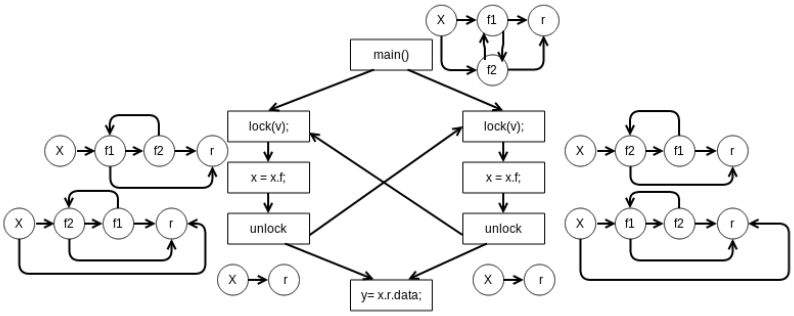

Figure 4.7: Concurrent Heap liveness analysis iteration 4

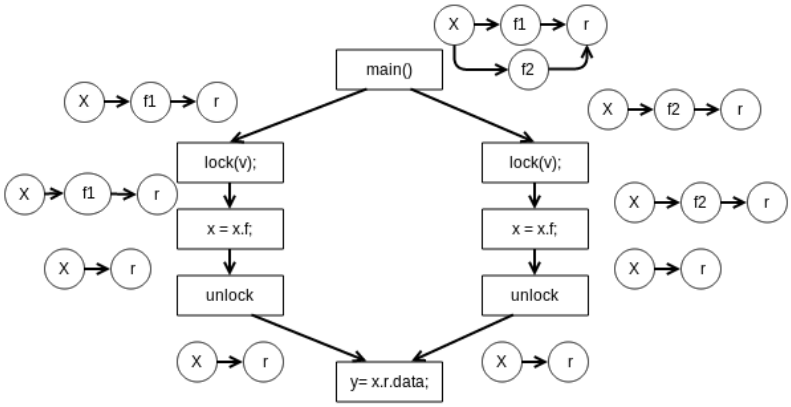

Figure 4.8: Concurrent Heap liveness analysis without synchronization edges

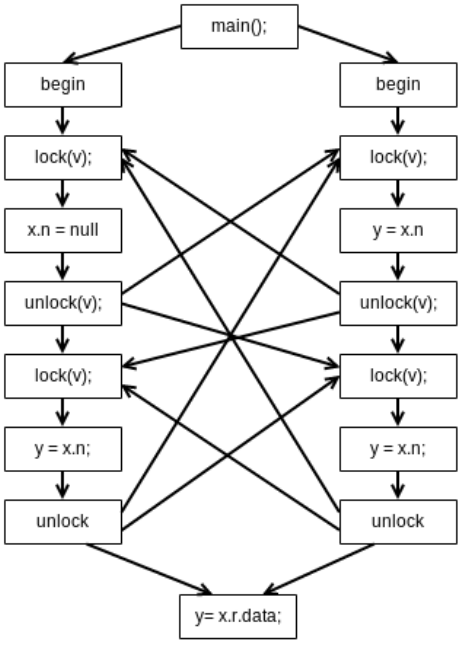

Figure 4.9: Concurrent Heap liveness analysis input

Chapter 5

Problem with the Concurrent Analysis Method

The analysis technique presented in the previous chapter, propagates the data flow values across inter-thread edges in the same way as it propagates the data flow along intra-thread edges. Consider the output of heap-liveness analysis technique on the same example of the previous chapter. The figure 5.1 is the final data flow value obtained after convergence.

At the program point containing the main statement, the possible live links would be *x.f1.f2.r* or *x.f2.f1.r*. However the final data flow value obtained at the main statement includes access links *x.f1.r*, *x.f2.r* , $x.\{f2.f1\}^+.r$, $x.\{f1.f2\}^+.r$ which are imprecise values. The major source of error is due to no bound on the number of transitions from node f_1 to f_2 and from f_2 to f_1 in the access graph.

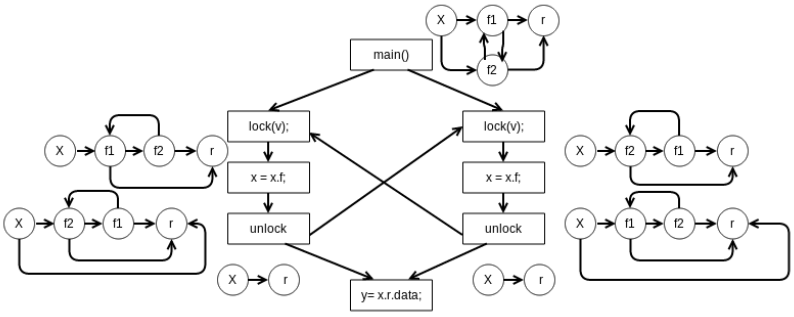

Figure 5.1: Example of the concurrent analysis technique

The imprecise data flow values are a result of taking into account execution of critical sections more than once. The thread synchronization edges introduce loops in the program graph. The current method treats these thread synchronization edges as loops and continues propagation of the facts until a converged value is obtained (as seen in the previous chapter).

5.1 Execution of Critical Sections

The execution of a multi-threaded program is an interleaving of statements of the multiple threads. Critical sections are regions where the shared variables are accessed under mutual exclusion. Note that addition of synchronization edges leads to formation of loop in the control flow graph. Due to this, data flow value is transferred to every critical section multiple times(even if there was no loop around the critical section before adding the synchronization edges). Thus there is a need to identify the critical sections that only execute once. The analysis to figure this out is thread independent. Note that a critical section can only be executed once if there are no backward program edges from a statement after unlock(v) to one before lock(v) in the thread. For such critical sections we need to keep track to thread switches encountered while performing the analysis. In the example figure 5.1, if the critical section of thread 1 is analyzed and the data flow value is transferred to thread 2 for analysis along an inter-thread edge, it is required to ensure that data flow value in not transferred back to thread 1 after analysis of thread 2 critical section.

For handling such cases, we need to examine the presence of loops within each thread.

- If there is no loop within and across a critical section, it can only be executed once.
- If a loop is present within a critical section, even then the critical section can be executed only once. This is so because if 1 thread enters a critical section, all other threads wait until the this thread completes executing the loop inside the critical section.
- It a loop is present across a critical section in a thread, then the critical section can be executed zero or more number of times. Apart from that we may have execution of other critical sections between two executions on this critical section (inter-leaving) possible.

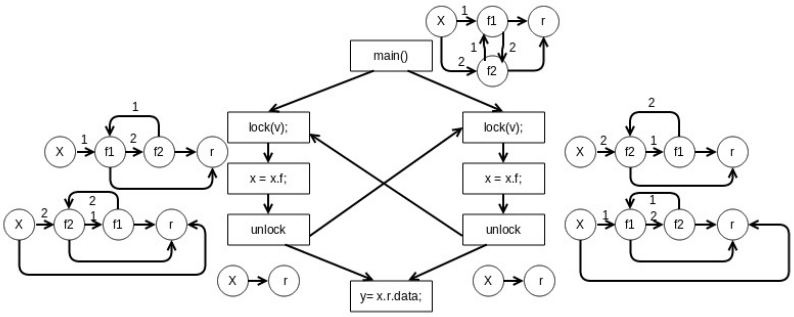

Figure 5.2: Concurrent Heap liveness analysis example. Note that edges are marked by thread id

Thus we need to figure out how to store the thread switchings in an access graph. One possible way is to store thread id corresponding to every edge in the access graph. Once each edge is labeled, we need to identify which paths are possible with respect to the execution semantics. For instance, as pointed above, we cannot allow paths with multiple thread switches into a critical section which can only be executed once. We will now need to examine all the paths where the execution semantics are followed.

5.2 Examples highlighting the problems

In figure 5.2, at the point main, we obtain an access graph which includes multiple access links. By imposing the restrictions that concurrent sections in thread 1 and 2 can only be executed once, we can discard all the access paths that contain more than 1 edges labeled 1 or 2. This way we can ensure that we get precise links.

Suppose we take another example in which we have an inner loop in a critical section (see figure 5.3). The access graph obtained in this example is again similar to the previous example. The difference being that the loop in $thread_2$'s critical section can be executed zero or more times. Due to this we get an edge from $f2$ to itself in each of the final access graphs. Note however that The possible executions of the program are either $C1.C2^*$ or $C2^*.C1$, where C_1 and C_2 are critical sections of thread 1 and 2 respectively. Even in this example, applying the basic concurrent analysis shows access links that are not live to be live. For example links like $x.f_1.{f_2}^*.f_1$, which contain multiple occurrences of f_1.

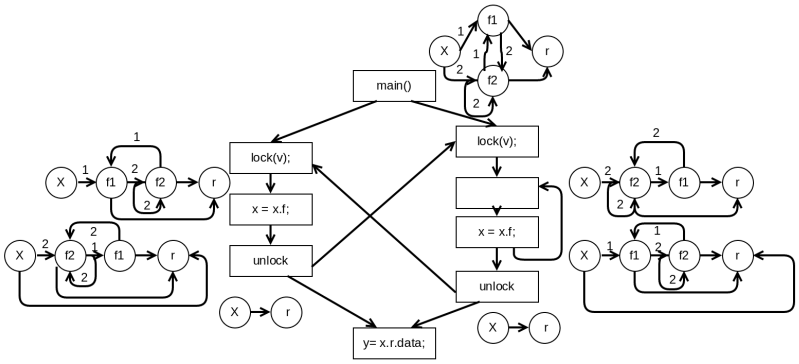

Figure 5.3: Concurrent Heap liveness analysis example (on sync cfg). Note that edges in access graphs are marked by thread id

For ensuring that only live paths are returned, we would need to impose the condition that critical section of thread 1 and thread 2 can only be executed once, while performing the analysis. We can allow 0 or more occurrence of edges labeled 2 whereas the number of edges labeled 1 must strictly be 1 for the example.

Another interesting case is when we have a loop across a critical section. This means that we can access the critical section zero or more times. In the previous 2 examples, we could only execute the critical sections once for all the threads. Consider the example shown in Figure 5.4. This is a slight modification of figure 5.3, with the loop being outside/across the lock/unlock statements. Even in this case applying the simple analysis on the program graph generated by adding the thread sync edges results in the same access graphs as the case of execution of loop inside a critical section.

Clearly the two examples have different executions possible. But the access graph obtained from the analysis on program graph(sync-cfg generated by adding synchronization edges) leads to a highly approximated access graph. Also there is no difference between the access graphs. The possible live access links at the program point main() are $x.f_1.{f_2}^*$ and $x.{f_2}^+.f_1.{f_2}^*$ for the example in figure 5.4. Example 5.4 is also different from 5.3 in a way that, the critical section of $thread_2$ in 5.4 can itself be executed zero or more times. Whereas in the case of 5.3, it would have only been executed once.

Summarizing all the problems we have encountered with the analysis technique. The first one being treatment of synchronization edges of the

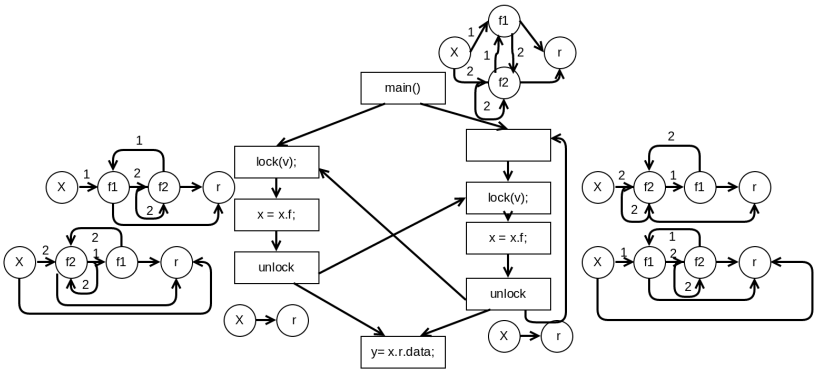

Figure 5.4: Concurrent Heap liveness analysis example (on sync-cfg).

program graph same as other edges. This leads to formation of loop in the control flow graph. During propagation of data flow values in the analysis, the values are transferred in the loop until a stable/converged value is reached. This fact corresponds to execution of the critical section(s) multiple times, which may not be the case most of the times. We would need to look at the analysis from the point of view of concurrent program execution. This analysis just assumes any starting point and merges data flow value along all the executions. In order to improve upon the precision we would need to come up with different starting points corresponding to each thread and propagate data flow values taking each thread as the starting point. Also , we would need to come up with a way to ensure that facts are propagated according the number of critical section executions possible.

Chapter 6

Improvements in the Analysis

In the previous chapter, we had show examples where the heap liveness analysis applied to the synchronized control flow graph returned a highly-over approximated access graph. The reason for this being the inter-thread synchronization edges not being treated differently from intra-thread edges. There is also a need to ensure that the final data flow value obtained can actually be a result of valid program execution.

In order to achieve this we need to analyze critical sections in all the threads. An intra-thread analysis will be required to be performed. We need to figure out the whether there exists a loop outside the critical section. If there is no such loop then we can say that the critical section will be executed once. Otherwise, The critical section can be executed zero or more times. Note that this information is independent of other threads. Once we have the thread dependent information , we can add inter-thread edges. The edges will only be triggered based on the condition that checks if the critical section can be executed. The condition would involve comparing counters of the number of times a critical section is executed.

One way to achieve this is to label the edges with the number of times the transition is possible. Considering the examples in the last chapter figure 5.1, 5.2 and 5.3, we can suggest the following access graphs at the statement main(). See figure 6.1.

In examples 5.2,5,3 and 5.4 , we had observed that access graph at the statement main() was the same. We now display the desired representation of access graph. For example 5.2, we have observed that the critical section can only be executed once for both the threads. So we need to mark the edge with either 1 or *, where 1 denotes that the transition can only be performed

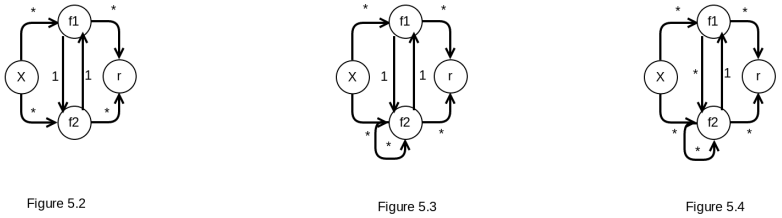

Figure 6.1: Desired Access graphs for fig 5.2,5.3 and 5.4

once and * is a doesn't care condition.

For 5.3, we had a loop inside critical section of thread 2. However the critical section could only be executed once. There will be a self loop around f_2 in the access graph. The edges from f_1 and f_2 are representative of a thread change. Since there are no loops across/outside the critical section, we can conclude that both these will execute once and mark the two edges 1.

For example 5.4, there is a thread 2 has a loop across the lock and unlock section. Thus now the edge representing transition from C_1 to C_2 can be executed any number of times. The desired access graph for 5.4, should be the similar to 5.3 except for the edge from f_1 to f_2 being labeled as * instead of 1.

We may need to modify the analysis in order to obtain the desired access graph for all the examples. One way may be to change the way we merge and summarize access graphs different threads.

6.1 Concurrent Heap Access examples

Let us try to understand the how heap memory is actually accessed by threaded programs using shared memory. This will be better understood by taking up the example of a concurrent data structure say tree. Each node of the tree has l and r fields pointing to children and a *data* field.

Consider the simplest example for this, similar to example 5.2 in figure 6.2. The access pattern is either T_1 followed by T_2 or the reverse. If T_1 is executed first then the object $x.l.r$ is accessed otherwise $x.r.l$ is accessed. This corresponds to the iteration 1 of simple concurrent heap liveness analysis. (See figure 4.4) So if the analysis is stopped at this point we would obtain

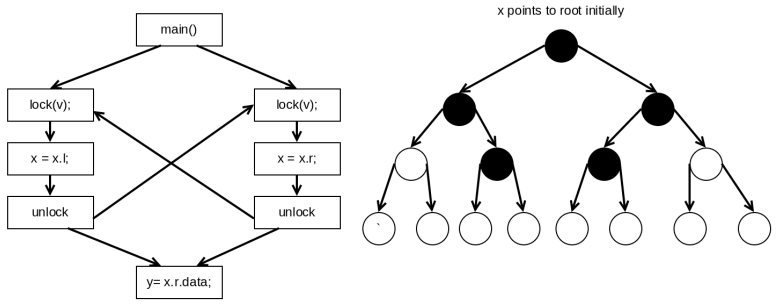

Figure 6.2: Concurrent access example in a tree

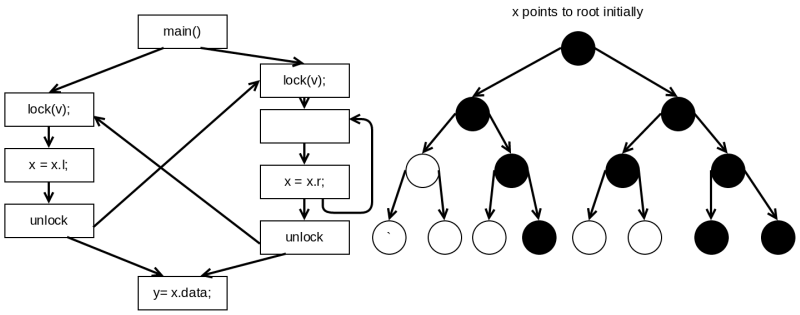

Figure 6.3: Another Concurrent access example in a tree

the desired access graph in this case.

Consider another example with a loop within the critical section in Figure 6.3. This is again inspired from Figure 5.3. Again the access pattern may be either T_1 followed by T_2 or the reverse. This implies the objects accessed will satisfy the regular expression $x.l.r^*$ or $x.r^*.l$ depending on the starting thread. Figure 6.3 shows the possible nodes that can be accessed by the concurrent program. It just maps the regular expression to the tree topology.

Now we will look into the possible executions of the example in Figure 6.4. This is the case of presence of loop across critical section. We will have two possible execution orders corresponding to start from T_1 or T_2 : $x.l.r^*$, $x.r^+.l.r^*$. Looking at the difference between this example and Figure 6.3, we notice that r^+ links from all the nodes of the type $x.r^+.l$ can be additionally accessed by this program.

Another interesting example is the case when both the loops are outside

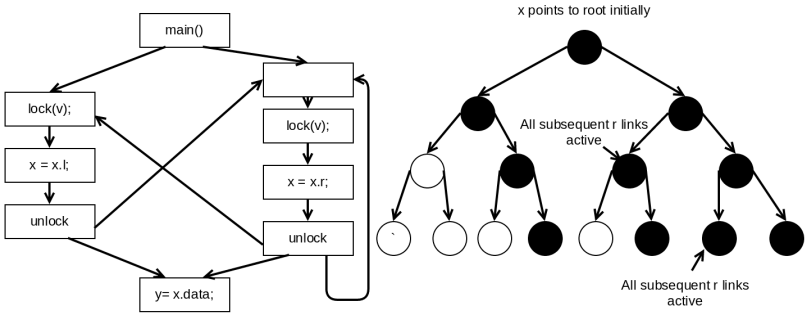

Figure 6.4: Another Concurrent access example in a tree

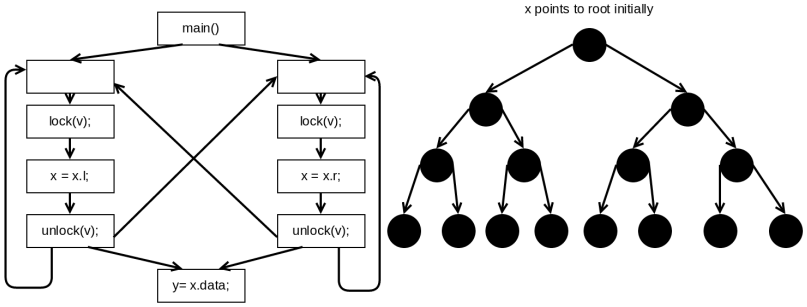

Figure 6.5: Another Concurrent access example in a tree

critical sections in Figure 6.5. This will lead to any number of executions of C_1 and C_2 and interleaving are also possible. Thus the access graph expected for this kind of execution is $x.\{l^*.r^*\}^*$. Starting with C_1, either multiple executions of C_1 and C_2 are possible. Also there is no restriction on the thread switchings. So for this example, all the nodes of the tree can be live/accessed.

From all these examples, we can think of a way to propagate data flow values/access graphs across the synchronized control flow graph. We would have n starting points , where n is the number of threads in the program. Starting from a thread, data flow values are propagated along valid execution sequences. A valid execution sequence is such that all the thread switchings are consistent. This is ensured by examining all the critical sections. In a valid execution sequence, critical sections with no loops outside it appear only once.

For instance, in Figure 6.2 execution sequences possible are T_1,T_2 or T_2,T_1. We can also write this in terms of execution over critical sections. For this

we will first need to run an analysis on each thread locally and find out the critical section(s) present. After identification, we also need to figure out whether the critical sections can be executed once or any number of times. Once we have the results from the intra-thread analysis, we will add the inter-thread edges to generate execution sequence over critical sections. For figure 6.2 and 6.3 the execution sequence can be either C_1,C_2 or C_2,C_1 since $count_c1$ and $count_c2$ is exactly 1. While in case of Figure 6.4, the possible executions are C_1,$C_2{}^*$ and C_2,C_1,$C_2{}^*$. Only $count_c1$ is exactly 1.

Consider an example for the above in Figure 6.6. Critical section of Thread 1 has 2 statements and that of Thread 2 has loop around the critical section. Nodes C_1 and C_2 are denoting these critical sections. C_2 has an edge from unto itself, denoting that it can be executed any number of times. Edges from C_1 to C_2 and from C_2 to C_1 are inter-thread edges. $count_c1$ is exactly equal to one. Let us start the analysis from thread 1. Possible execution path is C_1,$C_2{}^*$. We are considering heap liveness analysis, which is a backward analysis. Starting data flow analysis from thread 1, C_1, we get the access graph $x.l.r$ at the start of C_1. Transferring this graph to C_2, we obtain the access graph $x.l2^*.l1.r$. Suppose we start from Thread 2. The execution path is $C_2{}^+$,C_1,$C_2{}^*$. So the access graph obtained after the first step ($C_2{}^+$) is $x.l_2{}^+$. After the second step ($C_2{}^+$,C_1), we obtain $x.l_1.r_1.l_2{}^+$. And finally upon transferring the value to C_2 again we obtain, $x.l_2{}^+.l_1.r_1.l_2{}^+$. Note that, we do not merge the two l_2 nodes . This ensures that the access graph obtained will be such that, critical section C_1 will be executed only once. We have found a way to ensure this condition by not merging two nodes of the thread, if there is a thread switch to a thread whose critical section can only be executed once.

6.2 Representation for execution sequences

In the previous section, we had discussed about performing analysis along the possible execution sequences. It would ensure that the synchronized inter-thread edges do not simply act as loops. Also, merging along inter-thread execution paths was not allowed. We would store information in the graph to distinguish between inter & intra thread edges. Our aim is to build an automata-like representation of thread switches, actually critical section executions. Before doing that we would need to perform forward/backward analysis in each thread and generate the intra-thread execution sequence. This will be sequential. Using this information, an inter thread automata would be built, whose current state represents a program execution upto

that point. finally we would generate access graphs for each path in the inter-thread automata.

The following is a description of the procedure to generate the inter-thread automata. First generate an automata of critical sections for each thread. (This is the intra-thread step). In addition to this, identify for all critical sections if they are executed only once or any number of times. For each critical section, connect all the critical sections of all other threads. This is the inter-thread edge. This needs to be triggered upon a condition. In addition to this, for each critical section a counter storing the number of times the critical section is visited/executed is stored. The condition for an intra-thread edge entering a critical section C_i is based on the counter for C_i being strictly less than 1 if C_i can only be executed once. Otherwise, there is no need for a condition on inter-thread edges.

The above construction would lead to formation of a transition system. See figure 6.7 for an example 6.2. C_1 and C_2 are critical sections of thread 1 and 2 respectively. Since both of them can only be executed once, the incoming edge on C_i is triggered by the condition that count-ci <1.

Note that conditions on edges will only be of this form. We may have conjunction and disjunction over multiple conditions for an edge. Let us take the same example of figure 6.6. Since all the edge predicates/conditions are of the type var_i <1, we can try to convert this transition system to an automata. This basically some states of the automata need to be visited only once. This can be ensured by removing that state from the automata, and generating 2 copies of the remaining xautomata. All incoming edges to state S_i are connected from states in $copy_1$ and all the outgoing edges from S_i are connected to nodes in $copy_2$. the starting state is the starting state of $copy_1$. All other non-reachable states in $copy_2$ should be removed. Note that this results in duplication of some states. Say we have S_j and S'_j and we have removed S_i from the automata. Suppose in the original automata the condition for incoming edges into S_j was C_j <1. Upon duplication we would need to ensure the condition on S'_j would be C_j <1 && C'_j <1. Figure 6.7 shows the schematic of the step we have just described. Also note that edges outgoing from S to S'_j having the condition C'_j <1 will be ANDed with the condition of S_j, where S'_j is the duplicate of S_j in the right copy of A minus {S}. Alos note that the left copy satisfies the invariant $C_S = 0$ and the right side satisfies the invariant C_S ¿= 1. These predicates can be used to convert dotted edges to either solid edges (without conditions) or no edges (if the invariant makes the predicate false). Sometimes, it can simply set an

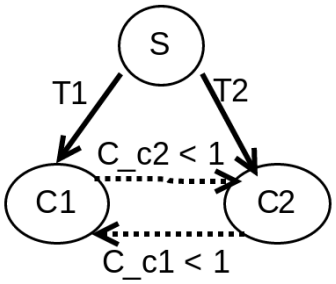

Figure 6.6: An example transition system

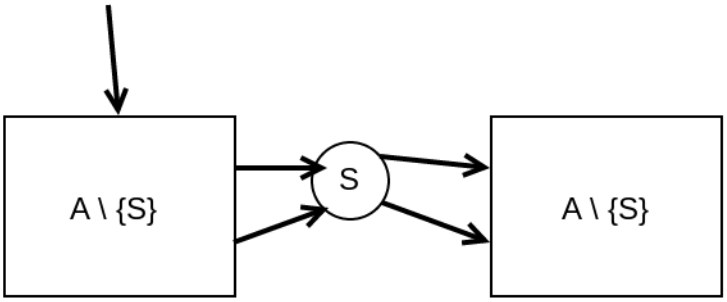

Figure 6.7: Ensuring S gets visited only once for a valid path

argument of a conjunction condition to true and simplify the condition. See figure 6.8 for the conversion to automata of figure 6.6. we will obtain the execution sequences. And then we will perform analysis for along all these thread contexts. And also note that we will not merge across inter-thread edges. Similarly we can convert other examples in chapter 5 to this form, by this method.

Note that this method of propagation of data flow values along valid thread contexts can be applied to any analysis, not just heap liveness analysis.

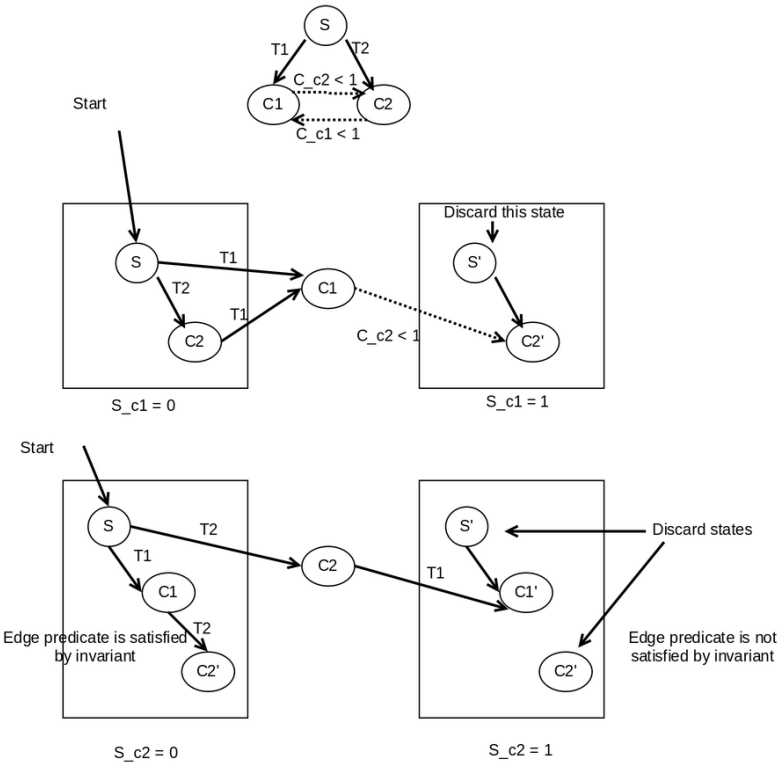

Figure 6.8: Converting the condtion based transition system to automata like structure.

Chapter 7

Tools for Design of the Analysis

For programs with procedure call, context-sensitive analysis is required to improve the precision of data flow facts. It is needed to extend the concurrent analysis technique to inter-procedural level. In the paper, an extension based on the call strings approach was proposed. However we wish to perform inter-procedural analysis using value contexts, for better way to handle recursive programs.

A thread now consists of a number of procedure calls, each with their own CFGs. Each thread has an entry procedure with the same name as the thread. Execution of a thread starts with the execution of the start node of the entry procedure. Due to the inter-procedural nature we would need to handle call and return statements by adding edges from the call statement to the root node of the called procedure CFG and edges from the return statement of the called procedure CFG to the statement immediately following the call statement.

As an input for the concurrent inter-procedural analysis, we will supply programs with multiple threads with procedure calls in the main thread functions which can be recursive.

7.1 Call Graph Generation

Call graph construction can be done using Soot[7]. The call graph is available only in whole program mode of soot (-w option). It can be accessed through the *getCallGraph* method. Alternatively, call graph can also be constructed using the VASCO using the class *vasco.callgraph.CallGraphTest* in the *vasco.callgraph package*. VASCO generates a much precise call graph

using flow and context sensitive points to analysis over the program. The arguments that need to be given along to run is the classpath containing the soot jar file, the output directory, maximum depth of call chains and the main class indicating the entry point of the program.

The command is:
java [-cp CLASSPATH] vasco.callgraph.CallGraphTest [-out DIR] [-k DEPTH] MAIN_CLASS.

For example
java -cp bin:jars/ vasco.callgraph.Test -out "vasco-output/" -k 9 tests.test*
is executed from the project root directory. The class file to be given as input is int the package tests.

For the given input program

```
package tests;
public class test {
    static class A {
        void foo() { bar(); }
        void bar() { }
    }

    public static void main(String[] args) {
        A a1 = new A();
        a1.foo();

        A a2 = new A();
        a2.foo();

        a2.bar();
    }
}
```

We get the calls in the main procedure as
PCG Method : main → foo
PCG Method : main → bar
PCG Method : main → foo
PCG Method : main → <init>
PCG Method : main → <init>

7.2 Combined Unit Graph

We will be using the *CombinedUnitGraph API* to create a sync-CFG required to perform the analysis. This is currently implemented to handle 2 threads and perform intra-procedural analysis. This needs to be improved to handle more than 2 threads and construct inter-procedural CFG as discussed.

Creating an intra-procedural CFG : We first create an object of the *CombinedUnitGraph* class by passing the body of the 1st CFG (for the first thread) to be combined to its constructor. Then, we call the *addLockUnlockUnitsForThread1* method to add the lock and unlock nodes in the 1st CFG and store them in respective lists. Now, we can add the 2nd CFG to this object by calling the *addGraph* method with the 2nd graph as an argument. This is to be followed by a call to the *addLockUnlockUnitsForThread2* method and finally the *addUnlockToLockEdges* method call in order to complete the construction of the sync-CFG by joining unlock to lock edges for the same lock object. Exact API details are provided in the thesis on Concurrent Analysis of programs.[8]

Consider this input program for intra-procedural live variable analysis:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LiveVariablesInput {
private Lock lock = new ReentrantLock();

private class Thread1 extends Thread {
    public void run() {
        int a=0,b=0;
        lock.lock();
        a = 0;
        lock.unlock();

        lock.lock();
        b = a;
        lock.unlock();
    }
}

private class Thread2 extends Thread {
```

```
    public void run() {
        int a=0,b=0;
        lock.lock();
        a = 0;
        lock.unlock();

        lock.lock();
        a = 0;
        lock.unlock();
    }
}

public static void main(String[] args) {
    LiveVariablesInput ip = new LiveVariablesInput();

    Thread t1 = new Thread(ip.new Thread1());
    t1.start();

    Thread t2 = new Thread(ip.new Thread2());
    t2.start();
    }
}
```

The combined unit graph creates a sync-CFG in Figure 5.1 for the statements in the jimple representation of the program. The class file is converted to Jimple using Soot.

7.3 VASCO framework

We have already seen the use of VASCO framework to generate precise call graph for inter-procedural programs. VASCO framework is also used to design inter-procedural analysis. To perform and analysis, it is needed to override the methods *normalFlowFunction, callEntryFlowFunction, callExitFlowFunction, callLocalFlowFunction, boundaryValue, copy, meet, topValue, programRepresentation* from the *ForwardInterProceduralAnalysis* (or *BackwardInterProceduralAnalysis* depending on the nature of the analysis).

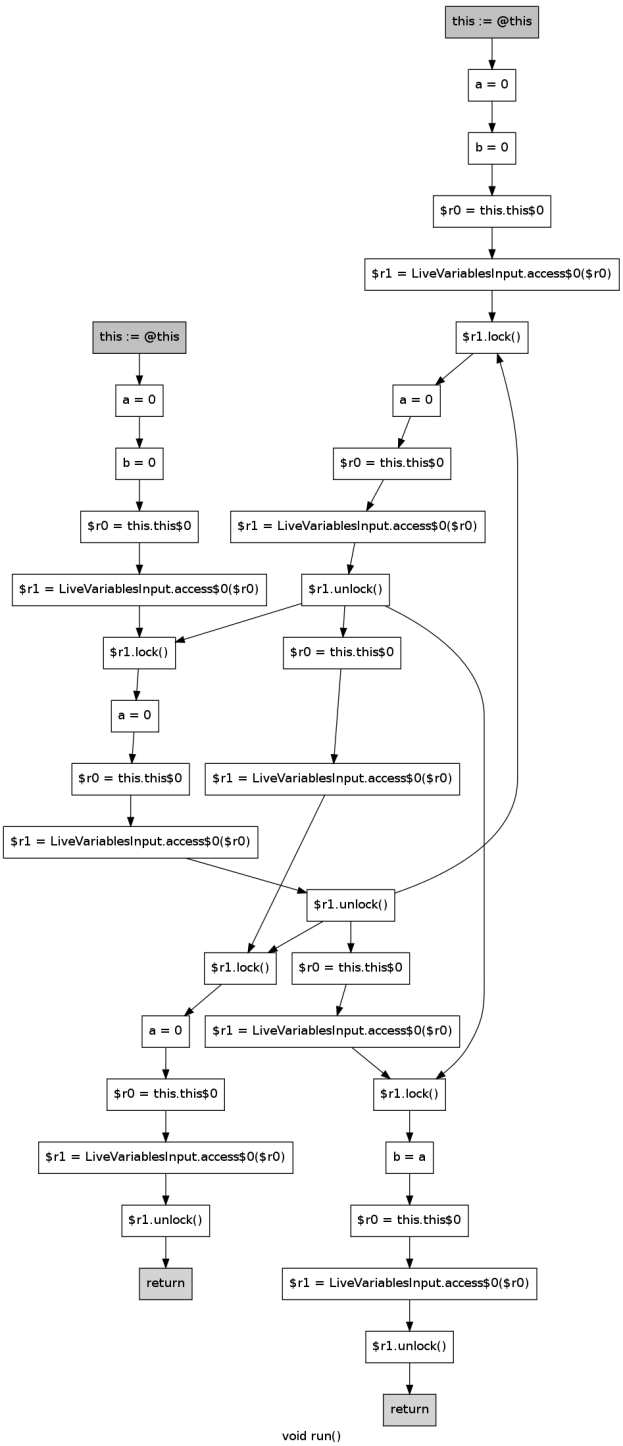

Figure 7.1: Sync-cfg for intra-procedural live variable analysis

Chapter 8

Conclusion and Future Work

In this report the topics covered are :

- The mechanism of Heap reference analysis using points-to and live variable analysis on access paths. It is necessary to store the access path values at a statement as an access graph to keep the data fact bounded.
- The method of value context used to carry out precise inter-procedural analysis with an example case on handling recursion.
- The technique of constructing sync-cfg for performing intra-procedural data flow analysis of concurrent programs.
- The extension of concurrent analysis to an inter-procedural level, by using VASCO for generating call-graph and designing inter-procedural analysis.
- Problems with the analysis on the sync-cfg/program graph. The reasons as to why this is an imprecise analysis are mentioned.
- Some improvements to the basic sync-cfg concurrent analysis suggested. The improvements distinguish between inter and intra-thread edges. The execution semantics of multi-thread programs is used to come up with a technique performs analysis taking into account valid executions of the program.
- The notion of thread context is introduced in a vague way. Analysis should only be performed over the thread switches which can actually be feasible. This notion is not just heap liveness analysis, but for any general analysis of a multi-threaded concurrent program.

- The automata like representation of critical section switching can be used to form a exploded-thread-cfg graph and we cn directly perform analysis over the statements in all the valid paths.

The future work to be done is:

- Comparing the running times of this analysis with the sync-cfg simple analysis.
- Implementation of the transition diagram to automata converter routine. Generation and implementation of exploded-thread-cfg from the automata structure of critical section execution.
- Figuring out if the analysis based on thread context/execution semantics can be used to extend to handle function calls in threads.

References

[1] Amitabha Sanyal Uday P. Khedker and Amey Karkar. *Heap Reference Analysis Using Access Graphs*. ACM Transactions on Programming Languages and Systems. 2007.

[2] Alefiya Lightwala. *Interprocedural Liveness Based Heap Points-to Analysis for Java*. MTP Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.

[3] Alan Mycroft Uday P. Khedker and Prashant Singh Rawat. *Liveness-Based pointer analysis*. SAS'12 Proceedings of the 19th international conference on Static Analysis. 2012.

[4] Uday P. Khedker. *Slides on Interprocedural Data Flow Analysis*. Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.

[5] Rohan Padhye and Uday P. Khedker. *Interprocedural Data Flow Analysis in Soot using Value Contexts*. In the Proceedings of the 2nd ACM SIGPLAN. 2013.

[6] Deepak DSouza Arnab De and Rupesh Nasre. *Dataflow Analysis for Datarace-Free Programs*. 2010.

[7] Arni Einarsson and Janus Dam Nielsen. "A Survivor's Guide to Java Program Analysis with Soot". Notes for the Soot framework from McGill University. 2008.

[8] Aditya Bhandari. *Data Flow Analysis for Concurrent Programs*. BTP Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.