Static Data Race Detection for Concurrent Programs with Asynchronous Calls

Vineet Kahlon
NEC Labs America, Princeton, USA.

Nishant Sinha
NEC Labs America Princeton, USA.

Yun Zhang
Department of Computer Science
Princeton University Princeton, USA.

Erik Kruus
NEC Labs America Princeton, USA.

Abstract

A large number of industrial concurrent programs are being designed based on a model which combines threads with event-based communication. These programs consist of several threads which perform computation by dispatching tasks to other threads via *asynchronous function calls*. These asynchronous function calls are implemented using function objects, which are essentially wrappers containing a pointer to the function that should be executed on a particular thread with the corresponding arguments. In many cases, the arguments, in turn, contain function objects which serve as callbacks. Verifying such programs which involves reasoning about complex concurrency constructs comprising function pointers and callback functions is extremely tricky especially in the presence of recursion. In this paper, we present a fast and accurate static data race detection technique for multi-threaded `C` programs with asynchronous function calls and demonstrate its application to real-life software.

1. Introduction

Leveraging parallelism effectively is key to enhancing the performance of software. While there exist many paradigms of concurrent programming, real-life massively concurrent systems are often based on an *event-driven* model wherein threads create tasks and dispatch these tasks to other threads, to be executed asynchronously. The tasks dispatched to a thread are enqueued on its work queue and processed in the order received. This model combines the expressiveness of both thread and event-based models of computation [15], and has the advantage that the thread that creates the task need not wait for it to finish. Threads can delegate computationally intensive tasks to other threads while continuing with more immediate tasks. Several large industrial concurrent systems and Internet services [28] including Ajax-based scripts, routers, and web servers use this model of concurrent computation. A number of libraries, e.g., Intel's threading building blocks [19] as well as languages [3] are available to support the design of concurrent systems with tasks.

Although providing both superior run-time performance as well as flexibility to the system designer, multi-threaded programs with asynchronous events are extremely hard to debug and verify due to the non-sequential flow of control inherent in such systems. First, there is a loose correlation between the asynchronous call and the value returned, making it difficult to track the flow of data across threads. More significantly, these calls are often made using function pointers, and the arguments to these calls may, in turn, contain function pointers to *callback* functions, which are executed, for example, upon completion of the call. Such indirect mechanisms for communication among threads makes analysis of these programs extremely challenging.

In this paper, we present a method to perform static data race detection for concurrent `C` programs which use asynchronous indirect function calls for communication. Given a multi-threaded program with asynchronous calls, our method first builds a precise context-sensitive concurrent control flow graph (CCFG) based on a flow and context-sensitive (FSCS) points-to analysis. Using this CCFG, we perform a staged data race detection, that involves (i) identifying the shared variables and lock pointers, (ii) computing an initial database of race warnings, and finally, (iii) pruning away the spurious warnings using a may-happen-in-parallel (MHP) analysis based on computing lock sets and performing thread order analysis. The main drawback of static analysis is that a large number of bogus data race warnings can often be generated which do not correspond to true bugs. Thus a key challenge in static data race detection lies in satisfying the conflicting goals of scalability and accuracy, i.e., keeping the bogus warning rate low.

Our computation model comprises a thread pool, where each thread iteratively processes tasks from its work queue. In order to send a task request to another thread t_2, thread t_1 issues an asynchronous function call with the task (e.g., a function to be executed) as an argument. An asynchronous function call may or may not be followed by an asynchronous function return (join). Note that the standard fork-join model for concurrent programs can be viewed as a particular case of the thread pool model (with unbounded number of threads, each having a zero-length queue), where fork corresponds to an asynchronous function call (AFC) to an unnamed thread while the join corresponds to the matching asynchronous return. When the thread pool is of finite size, each thread must have a non-zero length queue to store multiple incoming job requests. A key challenge in analyzing such programs is that given an asynchronous call, the function to be executed during the call, the arguments to that function, the callback functions passed to the call and the thread on which the call is to be executed can all be passed indirectly via pointers. Moreover, the values of each of these pointers can be set upstream in the code far away from the function call location. Thus without a precise (flow and context-sensitive) points-to analysis it is hard to even determine control flow in the given program. Recursion further complicates the problem.

The first and most crucial step in static data race detection, therefore, is to construct the CCFG of the given program. In the presence of function pointers and recursion, however, constructing the CCFG of the given program that satisfies the three key criteria of (i) **Soundness:** preserving every real data race, (ii) **Accuracy:** keeping the bogus warning rate low, and (iii) **Scalability**, is a non-trivial problem.

The main problem posed by the presence of function pointers is that it is not possible to compute the CCFG from the syntactic program description. In order to resolve these function pointers while keeping the bogus warning rate low, we have to carry out a flow and context sensitive (FSCS) function pointer alias analysis.

For FSCS pointer alias analysis we, in turn, need to compute the CCFG thereby creating a cyclic dependency between CCFG construction and function pointer resolution. This cyclic dependency can be broken via a context-sensitive construction of the CCFG wherein the contexts of the given program are enumerated starting at the entry function. As soon as a function pointer is encountered in a given context, its points-to set is computed which allows us to continue constructing the CCFG. A context-sensitive CCFG construction presents no problems in the absence of recursion. If recursion is present, however, the number of calling contexts for a recursive function can, in principle, be infinite. The may result in the size of the context sensitive CCFG being infinite. In [6], the problem is handled by introducing back edges for recursive functions. Thus if during the construction of the CCFG, the same function is encountered again in a given context, then instead of introducing a new copy of f a back edge is introduced to the existing copy of f. However, because of the introduction of back edges the construction no longer remains truly context-sensitive thereby leading to an over-approximation of the set of behaviors of the given program. This, in turns, leads to an increase in bogus data race warnings (see Sec. 6) rendering this technique unsuitable for our purposes.

In this paper, we present a new, *unified* approach for constructing the CCFG of a concurrent program with asynchronous calls, recursion and function pointers, while ensuring the precision of data race detection. The apparently disjoint goals of context-sensitive CCFG construction (using function pointer analysis) and lockset computation are achieved simultaneously by a uniform data flow analysis with a fix-point criteria. First, our analysis dove-tails the context-sensitive construction of the CFG with the (context-sensitive) computation of the points-to sets of function pointers. This allows us to resolve the functions called via pointers as and when they are encountered during the construction process. Secondly, in order to ensure that the size of the resulting CFG is finite, we formulate a fix-point criterion which enables us to enumerate a *representative* finite set of contexts that need be explored while ensuring both soundness and accuracy of static data race detection. The key insight in this work is that for purpose of static data race detection, it suffices to consider only those contexts that may generate *different* locksets at locations where shared variables are accessed. Such contexts are only finitely many in number. Interestingly, our fix-point criteria not only ensures finiteness of CCFG, but also prunes out contexts which cannot generate new data race warnings (even for program without function pointers and/or recursion - see Section 8). A fundamental difference between our technique and [6] is that in (over-)approximating all possible contexts via back edges the analysis in [6] loses precision and no longer remains truly context-sensitive. This directly impacts the accuracy of computing the context sensitive locksets leading to bogus warnings. We, on the other hand, use a fix-point criterion to explicitly enumerate a finite subset of contexts that need be explored while preserving soundness, i.e., no data race is missed, and carry out a precise context-sensitive lockset analysis individually for each of these contexts.

An additional challenge that needs to be addressed when computing FSCS points-to sets of lock and function/thread pointers is scalability. Indeed, whole program FSCS alias analysis is expensive. In order to ensure scalability, we leverage bootstrapping [11]. Since, for our application, we are interested only in lock and function/thread pointers we need to consider only those program statements which may affect aliases of these pointers. These statements can be isolated via bootstrapping. Since the statements that may affect aliases of lock or function pointers are very few in number it guarantees scalability of the FSCS analysis for lock and function/thread pointers. To further enhance scalability, we leverage the use of summarization. Since lock and function pointer aliases are typically updated by only a few functions, summaries for FSCS alias analysis (as formulated in [11]) for these pointers need be computed for only a small number of functions. Thus exploiting the synergy between bootstrapping and summarization enables us to efficiently compute the FSCS aliases of lock and function pointers during the CCFG construction.

Our approach has been implemented in the CoBE framework [12] for analyzing multi-threaded C programs. Section 2 presents a motivating example program which combines thread and indirect asynchronous call models. Section 3 describes the background concepts and notations used in the paper. We present the algorithm for constructing concurrent control flow graphs in Section 6. Based on the CCFG created, we present our technique for computing and refining the set of data race warnings in Section 7. Finally we describe the related work and conclude in Section 9.

2. Motivating Example

Figure 1 shows a slice of a concurrent `C` program which illustrates some of the complexities arising from a combination of indirect function calls and thread creation.

We use `C`-like syntax together with a special *thread* construct to denote thread identifiers and *named* fork (see Section 3) and join calls to denote thread creation and termination. More precisely, the call `fork (t1, f, arg1, arg2, ..)` denotes the creation of a thread with identifier `t1` which executes the function `f` with the actual arguments `arg1`, `arg2`, etc. . Similarly the call `join (t1)` blocks the caller until the thread `t1` finishes execution.

The example program starts from the `main` function, which creates threads `t1` or `t2` (lines `l0`, `l1` respectively). Both the threads execute the function `f`; however, depending on the arguments passed to `f`, the callback function `h1` or `h2` is invoked. The function `f` obtains the return value from the callback function and then writes to a shared variable `*z`. Since the shared variable may be written in either thread `*t1` or `*t2`, there is a potential race condition at the locations `l2` and `l3` in function `f`. However, as we show later, if we build the concurrent call graph in a context-sensitive manner and take into account the thread ordering imposed by thread creation and join, we can prove that such a race cannot happen. Also, note that since the functions `h1` and `h2` are called using function pointers, we will not be able to capture the exact asynchronous behavior of the program without a precise function pointer analysis.

3. Preliminaries

In this paper, we focus on static data race detection of multi-threaded `C` programs. A popular paradigm for writing multi-threaded `C` programs is the `POSIX`-style fork/join model. Here the program is comprised of a main thread which may *fork* one or more new threads providing them with a function pointer argument pointing to the function that the new thread must execute. In case, a thread executing a fork call needs to wait for the call to finish a matching join is executed.

While such a model is adequate for small programs, for efficiency reasons or due to resource limitations, large industrial-strength applications often go beyond this simple thread creation model by adopting a more flexible *thread pool* model [28], consisting of a number of threads each having its own work queue for processing tasks. In many cases, a set of threads are created at initialization and each thread is *named*, i.e., it has an unique identifier. Since the threads are created once at initialization, the caller thread simply creates an *execution job* and dispatches it to the callee thread. In order to send a task request to another thread t_2, thread t_1 issues an *asynchronous function call (AFC)* with the task (e.g., a function to be executed) as an argument. Note that the standard

```
int h1 (int x) {   int h2 (int y) {
  return x * x;       return y + y;
}                  }

struct funcType {
  int (*func) (int);
}

void f (int x, funcType *g, int *z) {
l2:  if ( x > 0) { *z = *(g->func) (x); }
l3:  else { *z = *(g->func) (-x); }
}

int main () {
  struct funcType ft;
  int a, b, p1, p2, z;
  struct thread t1, t2;
  ....
  if (p1) {
    ft.func = &h1;
l0: fork (t1, f, a, ft, &z);
    join (t1);
  }
  ...
  if (p2) {
    ft.func = &h2;
l1: fork (t2, f, b, ft, &z);
    join (t2);
  }
}
```

Figure 1. Motivating example concurrent program.

fork-join model for concurrent programs can be viewed as a particular case of the thread pool model (with unbounded number of threads, each having a zero-length queue), where fork corresponds to an AFC to an unnamed thread while the join corresponds to the matching asynchronous return. When the thread pool is of finite size, each thread must have a non-zero length queue to store multiple incoming job requests.

Program Model: Concurrent Control Flow Graph (CCFG) We consider concurrent imperative programs comprising threads that communicate using shared variables and synchronize with each other using standard primitives such as locks. Each thread $T_i : (F_i, \mathtt{e}_i, G_i, L_i)$ consists of procedures F_i, entry procedure $\mathtt{e}_i \in F_i$, a set of global variables G and thread local variables v. Each procedure $p \in F$, is associated with a tuple of formal arguments $\mathsf{args}(p)$, a return type t_p, local variables $L(p)$, and a control flow graph (CFG). Each procedural CFG $(N(p), E(p), \text{action})$ consists of a set of nodes $N(p)$ and a set of edges $E(p)$ between nodes in $N(p)$. A node in $N(p)$ is designated the *entry* node of the procedure and represents the statement wherein control flow enters the procedure. Similarly a subset of nodes of $N(p)$ are designated the *exit* nodes and represent statements where control flow may leave the procedure. Each edge $m \rightarrow n \in E(p)$ is associated with an *action* that is an assignment, a call to another procedure, a return statement, a conditional guard, a synchronization statement, a *named fork* statement or a *named join* statement. The actions in the CFG for a procedure p may refer to variables in the set $G \cup \mathsf{args}(p) \cup L(p)$. The named fork edge provides a means to model, in a unified fashion, the fork operation in the fork/join model and AFCs in the thread pool model. A named fork edge occurs from the program location where the fork or AFC is made to the entry node of the procedure to be executed. If the thread on which the procedure is to be executed is specified as in the thread pool model, the named fork is labeled with the corresponding thread-id. If the thread-id is unknown or irrelevant as in the fork/join model it is simply labeled with '?'. A named join matching a named fork that executes procedure g occurs from an exit node of g to the node representing the join location.

Complexities of AFCs. While the AFC construct provides a very flexible and powerful tool in the hands of programmers, it also introduces many challenges when it comes to analyzing code. A common mechanism to implements AFCs is via the use of bound function objects, e.g., in the Boost library [23]. For example, the function `makeAFC` creates an `AFCTask bf` in the following way.

```
struct AFCTask *bf = makeAFC(thread_t,bind(&g,args));
```

The `bf` object contains the thread pointer $thread\_t$, i.e., the thread on which the task is to be executed, the function pointer $\&g$ pointing to the function g to be executed, and $args$ are the arguments to g. The arguments $args$ are bound to the corresponding function g using the `bind` construct. The actual asynchronous function call takes place with the help of the `enqueue` function: on executing $enqueue(\mathtt{bf})$ in the caller thread, the `bf` object task is dispatched to the working queue of the thread pointed-to by $thread\_t$, which is then executed in a first-in-first-out manner from the queue. Note that the location of creation and dispatch of an AFC may be far removed from each other in the code. Liberal use of AFCs can lead to code that is extremely hard to understand and debug. Consider, for example, the following nested AFC object `afc`, where the function object created by `bind (&bar, barArg)` is itself an argument to another function object:

```
struct AFCTask *afc = makeAFCTask (t,
bind (&g, gArg1, gArg2, bind (&bar, barArg)) );
```

On dispatching the above object, the task with function g (and arguments $gArg1$, $gArg2$ and $bind(\&bar, barArg)$) will be executed on thread t. Upon completion, g may make another AFC with function bar and argument $barArg$. The target of the second AFC might be the same thread t or another thread whose identifier is available to g at the time of execution. Note that this target was kept ambiguous at the time of creation of the `afc` object, and the intent of the programmer at the creation point might be completely forgotten while implementing the function g. This gives rise to subtle bugs which produce unexpected execution results and are extremely difficult to trace due to the indirect call sequences. Since the instantiations to the various entities in an AFC, i.e., thread pointers, function pointers, arguments, etc., may be physically spread out in the source code, it is easy to see that one of the key challenges in analyzing multi-threaded `C/C++` code with AFCs lies in constructing the CCFG from source code.

4. Bootstrapping based Pointer Analysis

We review some useful facts about the well-known Steensgaard's points-to analysis [24] and how to leverage it in improving the scalability of flow and context-sensitive (FSCS) alias analyses via bootstrapping [11].

Steensgaard's Analysis. In Steensgaard's analysis [24], aliasing information is maintained as a relation over abstract memory locations. Every location l is associated with a set of pointers and holds some content α which is an abstract pointer value. Points-to information between abstract pointers is stored as a points-to graph which is a directed graph whose nodes represent sets of pointers and edges encode the points-to relation between them. Intuitively, an edge $e : v_1 \rightarrow v_2$ from nodes v_1 to v_2 represents the fact that a pointer in the set represented by v_1 may point to some pointer

in the set represented by v_2. The key feature of Steensgaard's analysis that is used in bootstrapping is the well known fact that the points-to sets so generated are equivalence classes (see [11]). Hence these sets define a partitioning of the set of all pointers in the given program into disjoint subsets, called *Steensgaard Partitions*, that *respect the aliasing relation*, i.e., a pointer can only be aliased to pointers within its own partition. For pointer p, let n_p denote the node in the Steensgaard points-to graph representing the Steensgaard partition containing p. A Steensgaard points-to graph defines an ordering on the pointers in P which we refer to as the *Steensgaard points-to hierarchy* (see [11]). For pointers p, q, we say that p is *higher than* q in the Steensgaard points-to hierarchy, denoted by $p > q$, or equivalently by $q < p$, if n_p and n_q are distinct nodes and there is a path from n_p to n_q in the Steensgaard points-to graph. Also, we write $p \sim q$ to mean that p and q both belong to the same Steensgaard partition.

Divide and Conquer via Bootstrapping. Whole program flow sensitive and context sensitive (FSCS) alias analysis is expensive. However, for many applications we require FSCS aliases for only a small set of pointers of interest. For instance, for static data race detection, we need to compute FSCS aliases for only the lock pointers. Thus, if we are interested in computing the aliases of pointers in a given set S, we want to leverage divide and conquer by restricting our analysis only to those statements of the given program that may affect aliases of pointers in S. Towards that end, we leverage *bootstrapping* [11] which exploits the fact that the aliases of a pointer in a Steensgaard partition P can be affected only by assignments to either a pointer in P or a pointer q higher in the Steensgaard points-to hierarchy than some pointer in P. Assume now that our goal is to compute FSCS aliases of a pointer $p \in P$. Then it suffices to restrict our analysis only to statements that directly modify values of pointers in the set $P_{\geq}$ comprised of all pointers q such that either $q > p$ or $q \sim p$. It follows from the above observation that if we are interested in the FSCS aliases of pointers in a set S, then it suffices to restrict the analysis to pointers in the *Steensgaard-closure* of S defined as follows:

Definition (Steensgaard Closure). *The Steensgaard closure of a set S of pointers, denoted by $Cl(S)$, in the minimal set with the property that $S \subseteq Cl(S)$ and for each $p \in Cl(S)$ if either $q \sim p$ or $q > p$ then $q \in Cl(S)$.*

4.1 Complete Update Sequences

In resolving FSCS points-to sets of function/thread pointers during CCFG construction, we exploit summarization in which the notion of a complete update sequence plays a crucial role. A pointer p is said to be *semantically equivalent* to q at location l if p and q have the same value at l (even if they are syntactically different).

Definition (Complete Update Sequence) [11]. *Let $\lambda : l_0, ..., l_m$ be a sequence of successive program locations and let π be the sequence $l_{i_1} : p_1 = a_0$, $l_{i_2} : p_2 = a_1$,..., $l_{i_k} : p_k = a_{k-1}$, of pointer assignments occurring along λ. Then π is called a complete update sequence from p to q leading from locations l_0 to l_m iff*

- *a_0 and p_k are semantically equivalent to p and q at locations l_0 and l_m, respectively.*
- *for each j, a_j is semantically equivalent to p_j at l_{i_j},*
- *for each j, there does not exist any (semantic) assignment to pointer a_j between locations l_{i_j} and $l_{i_{j+1}}$; to a_0 between l_0 and l_{i_1}; and to p_k between l_{i_k} and l_m along λ.*

A related concept is that of maximally complete update sequences.

Definition 4 (Maximally Complete Update Sequence). *Given a sequence $\lambda : l_0, ..., l_m$ of successive control locations starting at the entry control location l_0 of the given program, the maximally complete update sequence for pointer q leading from locations l_0 to l_m along λ is the complete update sequence π of maximum length, over all pointers p, from p to q (leading from locations l_0 to l_m) occurring along λ. If π is an update sequence from p to q leading from locations l_0 to l_m, we also call it a maximally complete update sequence from p to q leading from locations l_0 to l_m.*

Maximally complete update sequences can be used to characterize aliasing.

Theorem [11] (Aliasing Theorem) *Pointers p and q are aliased at control location l iff there exists a sequence λ of successive control locations starting at the entry location l_0 of the given program and ending at l such that there exists a pointer a with the property that there exist maximally complete update sequences from a to both p and q (leading from l_0 to l) along λ.*

Advantages of using Update Sequences. A key advantage of using update sequences to characterize aliasing is that update sequences can be summarized in a compact manner. Additionally, bootstrapping allows us to exploit locality of reference. Indeed, since Steensgaard partitions are typically small, by restricting summary computation to each individual partition ensures that the resulting summaries will also be small. Secondly, the number of statements modifying values of pointers in a given partition also tend to be few and highly localized to a few functions. This in turn, obviates the need for computing summaries for functions that don't modify any pointers in the given partition which typically accounts for majority of the functions. Note that without partitioning it would be hard to ensure viability of the summarization approach. Thus it is the synergy between divide and conquer and summarization that ensures scalability of the FSCS alias analysis.

5. Static Data Race Detection

The classical approach to data race detection involves five steps.

1. Identify shared variables, i.e., variables which can be accessed by two or more threads.
2. Enumerate control locations where shared variables are read or written. These determine potential locations where data races can arise.
3. Determine locksets, i.e., the set of locks held, at locations where shared variables are accessed.
4. Each pair of control locations in two different threads where the same shared variable is accessed and disjoint sets of locks are held is labeled a data race warning.
5. Use causality constraints imposed by fork/join operations as well as synchronization primitives to reduce the set of warnings.

Context-Sensitive Lockset Analysis Since locks are typically accessed via pointers, in order to determine locksets in step 3, a precise points-to analysis must be carried out. Consider, a data race warning $\langle (c_1, L_1), (c_2, L_2) \rangle$, where c_1 and c_2 indicate control locations in two different threads where the same shared variable is accessed with at least one of the accesses being a write operation and L_1 and L_2 are the locksets at c_1 and c_2, respectively, such that $L_1 \cap L_2 = \emptyset$. Note that since in static analysis we typically ignore conditional statements, all syntactic paths leading to location c_i are possible. Thus L_i must capture the set of locks that *must* be held irrespective of the path leading to c_i, i.e., L_i must be the intersection of locks held along all paths leading to c_i, often referred to in the literature as *must-locksets*. For the sake of accuracy it is, therefore, imperative that points-to sets of lock pointers be computed context sensitively. This is because lock pointers typically point-to different locks in different contexts so that a context-insensitive lockset computation would result in the must-locksets being empty at most

```
                          h(){
                          1d: fp = &d;
main(){                   2d: lkptr = &lk2;
if(cond)                  3d: f(fp);
1a: fp = &g;              }
else
2a: fp = &h;              e(){
3a: f(fp);                1e: fp = &e;
}                         2e: lock(lkptr);
                          3e: sh1 = 1;
f(fp1){                   4e: unlock(lkptr);
1b: (*fp1)();             5e: f(fp);
}                         }

g(){                      d(){
1c: fp = &e;              1f: fp = &d;
2c: lkptr = &lk1;         2f: lock(lkptr);
3c: f(fp);                3f: sh2 = 2;
}                         4f: unlock(lkptr);
                          5f: f(fp);
                          }
```

Figure 2. An Example Program

locations where shared variables are accessed. This would lead to an explosion in the number of bogus warnings.

6. Concurrent Control Flow Graph Computation

In order to carry out static data race detection via steps 1-5 enumerated above we need to first construct the concurrent control flow graph of the given program. In the absence of function pointers and asynchronous calls this is straightforward. However, the presence of recursion and function/thread pointers give rise to several challenges discussed below.

Function and Thread Pointers. Realistic programs often make use of (indirect) function calls via function pointers and asynchronous calls involving function and thread pointers (see Sec. 3). As a result, it is not possible to compute the call graph of a program from its syntactic program description. In order to resolve these function/thread pointers we have to carry out a function/thread pointer alias analysis. Additionally, for must-lockset computation (see sec. 5), we need to compute the points-to sets of lock pointers context-sensitively for which the contexts of the given program have to be enumerated precisely. In order to enumerate contexts, i.e., construct the CCFG, the points-to sets of the function/thread pointers need to be computed context sensitively which, in turn, requires us to first compute the CCFG. This creates a cyclic dependency between CCFG construction and resolution of function/thread pointers.

Callback Functions. Callback functions are often passed as arguments to function calls, e.g., for exception handling. These callback functions, which are usually passed via function pointers, may be initialized much before the function call executes. Therefore, we must track the values of function pointer arguments for each function call in order to build the CCFG. An additional challenge in that the values of function pointers, in many cases, depend on the actual calling context of the called function. Thus the function pointer points-to analysis for resolving callback function pointers needs to be both flow and context-sensitive.

Recursion. As noted above, in order to resolve points-to sets of function pointers, it is important to carry out a context-sensitive points-to analysis. Because of the cyclic dependency between CCFG construction and resolution of points-to sets of function pointers, we need to construct the CCFG *context-sensitively*. Towards that end, an obvious approach is to enumerate all the contexts of the given program starting from the entry function. As soon as a function pointer is encountered in a given context, its points-to set is computed which allows us to continue constructing the CCFG. Note that since we resolve the points-to sets of function pointers in a given context, we can usually resolve the function being called uniquely. A context-sensitive CCFG construction presents no problems in the absence of recursion. In the presence of recursion, however, the number of calling contexts for a recursive function f can, in principle, be infinite. This may result in the size of the context sensitive CCFG being infinite. In [6], the problem is handled by introducing back edges for recursive functions. Thus if during the construction of the CCFG, the same function is encountered again in a given context, then instead of introducing a new copy of *f* a back edge in introduced to an existing copy of f. However, the introduction of back edges over-approximates the behaviors of the given program as a result of which the construction no longer remains context-sensitive (see below for an example). This reduces the accuracy of the must-lockset computation which, as discussed above, needs to be context-sensitive, thereby leading to bogus data race warnings.

Scalability. Finally, since we need to carry out a FSCS points-to analysis to compute the points-to sets of the function, thread and lock pointers, we need to contend with scalability issues.

6.1 CCFG Construction

We start by presenting an algorithm for constructing the concurrent control flow graph (CCFG) that handles the issues discussed above while ensuring both scalability as well as accuracy of the overall static data race detection framework. As discussed above, the cyclic dependency between CCFG construction and resolution of the points-to sets of function pointers can be broken by constructing the CCFG context-sensitively and resolving the points-to sets of the function pointers on-the-fly. The main challenge in following this approach lies in handling recursion. If there exist recursive procedures in the given program then it may, in principle, result in infinitely many contexts. Indeed, a self-recursive function f with a function call *fcall* to itself will generate the infinitely many contexts $fcall^i$, where $i \geq 1$.

In order to resolve this problem, in [6] if while constructing the CCFG a function *f* is encountered again in a given context, then a *back edge* to an existing copy of *f* is introduced. This approach, however, is not suitable for static data race detection as it leads to bogus warnings as we now illustrate.

Example. Consider a concurrent program comprised of two threads each running the code shown in Fig. 2 with *main* as the entry function and sh_1 and sh_2 being the shared variables. The algorithm for CFG construction formulated in [6] builds an *invocation graph* by unrolling the contexts, i.e., sequence of function calls of the given program. For our example, the procedure starts with the entry function *main*. Since *main* calls *f*, an edge is added from *main* to *f* in the invocation graph. In *f*, the function pointer *fp1* could point to either g or h, depending upon whether *cond* evaluates to *true* or *false* in *main*. Thus both g and h are added as successors to f. Function g, in turn, calls *f*. Since *f* already exists in the current context, i.e., *main* $\rightarrow f \rightarrow g$, a back edge to f is added in the invocation graph. Similarly a back-edge is added from h to f. Finally, f may call e and d both of which, in turn, call f. Thus e and d are both added as successors to f along with back edges from e and d to f resulting in the invocation graph shown in fig 3(a).

Because of the structure of the resulting invocation graph, we see that the must-locksets of *lkptr* at locations 2e and 2f in functions e and d, respectively, are empty sets. This is because *lkptr* is set as pointing-to lk_1 and lk_2 in functions g and h, respectively.

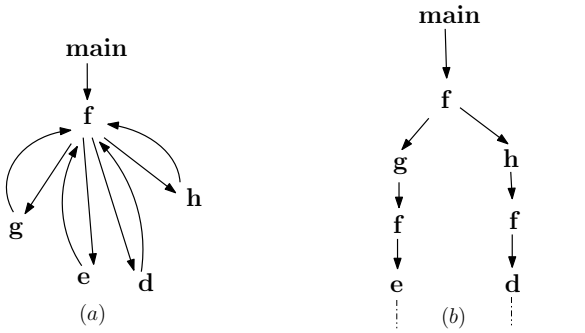

Figure 3. Call graph for the example in Fig. 2

In computing the must-aliases of *lkptr* at 2e, we must, by definition, take the intersection of aliases of *lkptr* along all paths in the invocation graph leading to 2e. However, due to the existence of back-edges from both g and h to f, we see that the paths *main, f, g, f, e* and *main, f, h, f, e* contribute different aliases, i.e., $\&lk_1$ and $\&lk_2$, respectively. Thus the set of must-aliases of *lkptr* at 2*e* is the empty set. As a result, the must-locksets at 3e and 3f are empty sets. In a concurrent program with two threads executing the code shown in Fig. 2, this would result in the pair (3e, 3e) being labeled a data race warning. Similarly, (3f, 3f) would also constitute a data race warning.

If, however, we unroll the contexts without introducing back edges we get an infinite call graph, part of which is shown in fig. 3(b). Here we see that starting at the entry function *main*, all paths leading to any given instance of e pass only through g without passing through h. Thus we can uniquely resolve the points-to set of *lkptr* at 2e as lk_1 as a result of which the must-lockset at 3e is $\{lk_1\}$. Similarly, the lockset at 3f is $\{lk_2\}$. Since the same lock is held at 3e in both the threads, (3e, 3e) no longer constitutes a data race warning. Similarly, (3f, 3f) is also not a data race warning.

To sum up, introducing back-edges in the invocation graph over-approximates the set of behaviors of the given program leading to bogus warnings. Thus finitizing the CCFG via the introduction of back edges is not a viable technique for static data race detection. We now present a new technique that instead of (over-)approximating all possible contexts via back-edges, enumerates a finite subset of contexts that need be explored without losing precision of the analysis at hand. Once these contexts have been enumerated, the must-locksets can be computed in each individual context thereby preserving context-sensitivity in lockset computation that is lost in the back-edge approach.

6.2 Lockset-based Fix-Point Procedure

In the presence of recursion, the number of contexts in a program is infinite. However, for our application, e.g., lockset-based static data race detection, by exploiting the fact that not all these contexts may generate different data race warnings, we show that it suffices to explore only finitely many contexts. Broadly speaking, our goal is to consider only those contexts that may generate different locksets at locations where shared variables are accessed.

CCFG Finitization: Our core strategy for finitizing the CCFG is to explore only those contexts in which the points-to sets of lock, function or thread pointers, or locksets are different. Note that lock pointers need to be tracked for lockset computation for data race detection whereas function and thread pointers need to be tracked for constructing the CCFG.

In general, tracking points-to sets of all pointers in the given program in a flow and context-sensitive (FSCS) fashion would be intractable. For our application, however, since we need to track points-to sets of only the set S of lock, function and thread pointers, we can leverage bootstrapping [11]. Towards that end, we note that the Steensgaard closure $Cl(S)$ of S (see Sec. 4) would simply be the set of lock, function and thread pointers, pointers to lock, function or thread pointers, pointers to pointers to lock, function or thread pointers, etc. Thus when computing FSCS aliases of the set S, we can slice away all statements of the given program that do not affect the aliases of pointers in S, i.e., those that are not assignments to any pointer in $Cl(S)$. Since the number of statements affecting lock, function or thread pointers is typically small, bootstrapping results in a highly precise as well as scalable FSCS analysis for such pointers.

In computing the CCFG context sensitively, if we encounter a function call again, we need to decide whether to explore it or not. Let $con_1 = con_1'.fcall_1$ and $con_2 = con_2'.fcall_2$ be two contexts such that $fcall_1$ and $fcall_2$ are instances of the same function call. Suppose that in constructing the CCFG we have already explored the context con_1 and then encounter the context con_2. Our criterion for continuing to explore $fcall_2$ is whether doing so could lead to the discovery of new aliases for lock, function or thread pointers or new locksets. Towards that end, we show that if in the two contexts con_1' and con_2' the aliases of all pointers in $Cl(S)$ are the same then exploring $fcall_2$ further cannot lead to the discovery of new alias sets at any given location for any pointer in $Cl(S)$. This follows from the following result (see [26] for the proof).

Theorem 1 (Finitization). *Let R be a set of pointers and let $Con_1 = con_1.fc_1$ and $Con_2 = con_2.fc_2$ be contexts such that fc_1 and fc_2 are instances of the same function call fc. Suppose that (i) for each $p \in Cl(R)$, the aliases of p, and (ii) the must-locksets, are the same at the call location of fc in both the contexts con_1 and con_2. Then for any sequence seq of function calls leading to function h, if $Con_1' = Con_1.seq$ and $Con_2' = Con_2.seq$ are valid contexts then the aliases of each pointer in $Cl(R)$ and the must-locksets are the same at each location of h in Con_1' and Con_2'.*

Proof. Let *loc* be a location of h and let $p \in Cl(R)$. Let q be aliased to p at *loc* in context Con_1'. We show that q is also aliased to p at *loc* in context Con_2'. Since p and q are aliased to each other in Con_1', by the aliasing theorem (see pg. 4), there exist maximal update sequences $seq_p : l_{i_1} : p_1 = a_0, l_{i_2} : p_2 = a_1$,..., $l_{i_k} : p_k = a_{k-1}$ and $seq_q : l_{j_1} : q_1 = b_0, l_{j_2} : q_2 = b_1$,..., $l_{j_m} : q_m = b_{m-1}$ starting from the entry location of the given program and leading to *loc* along Con_1' such that (i) $a_0 = b_0$, (ii) p_k and q_m are semantically[1] equal to p and q at locations l_{i_k} and l_{j_m}, respectively, (iii) for each k', $p_{k'}$ is semantically equal to $a_{k'}$ at $l_{i_{k'+1}}$ and $p_{k'}$ is not modified (semantically) along the sequence $l_{k'}+1, ..., l_{k'+1}$ of program locations, and (iv) for each l', $q_{l'}$ is semantically equal to $b_{l'}$ at $l_{j_{l'+1}}$ and $q_{l'}$ is not modified (semantically) along the sequence $l_{l'}+1, ..., l_{l'+1}$ of program locations. Suppose that along the context Con_1', the location l_{call_1} from where fc_1 is called occurs between the locations $l_{i_{c_1}}$ and $l_{i_{c_1}+1}$, i.e., $call_1$ lies in the open interval $(i_{c_1}, i_{c_1}+1)$ (we assume that there is no assignment at the location of a function call). Similarly, we assume that l_{call_1} occurs between the locations $l_{j_{c_2}}$ and $l_{j_{c_2}+1}$ along Con_1', i.e., $call_1 \in (j_{c_2}, j_{c_2}+1)$. Note that the location l_{call_1} occurs along the context con_1.

Consider now the maximal update sub-sequences $seq_q' : l_{i_1} : p_1 = a_0, l_{i_2} : p_2 = a_1$,..., $l_{i_{c_1}} : p_{c_1} = a_{c_1-1}$ and $seq_q' : q_1 = b_0, l_{j_2} : q_2 = b_1$,..., $l_{j_{c_2}} : q_{c_2} = b_{c_2-1}$. We observe that there cannot occur a statement that (semantically) modifies p_{c_1} between locations $l_{i_{c_1}}$ and l_{call_1} along con_1, else seq_p won't be a complete update sequence. Thus seq_p' is, in fact, a maximally complete

[1] We use the term *semantically equal* to mean that p_k and q_m could be accessed directly or indirectly, i.e., via dereferencing of pointer variables.

update sequence from the entry location of the given program to the location l_{call_1}. Similarly, seq'_q is a maximally complete sequence from the entry location of the given program to the location l_{call_1} along con_1. Then since $a_0 = b_0$, we have by the aliasing theorem, that p_{c_1} and q_{c_2} are aliased to each other at location l_{call_1} in con_1. Then by the hypothesis of our theorem, p_{c_1} and q_{c_2} are also aliased to each other at the location l_{call_2} from where fc_2 is called in con_2. From the aliasing theorem, we have that there exist complete maximal update sequences $\mathsf{seq}'_p : l_{i'_1} : r_1 = c_0$, $l_{i'_2} : r_2 = c_1$,..., $l_{i'_u} : r_u = c_{u-1}$ and $\mathsf{seq}'_q : l_{j'_1} : s_1 = d_0$, $l_{j'_2} : s_2 = d_1$,..., $l_{j'_v} : s_v = d_{v-1}$ starting from the entry location of the given program and leading to l_{call_2} such that $c_0 = d_0$; for each k', $r_{k'}$ is semantically equal to $c_{k'}$; for each l', $s_{l'}$ is semantically equal to $d_{l'}$; and r_u and s_v are semantically equal to p_{c_1} and q_{c_2} at l_{call_2} respectively. Consider now the sequences, $\mathsf{seq}_p : l_{i'_1} : r_1 = c_0$, $l_{i'_2} : r_2 = c_1$,..., $l_{i'_u} : r_u = c_{u-1}$, $l_{i_{c_1}+1} : p_{c_1+1} = a_{i_{c_1}}$, ..., $l_{i_k} : p_k = a_{k-1}$ and $\mathsf{seq}_q : l_{j'_1} : s_1 = d_0$, $l_{j'_2} : s_2 = d_1$,..., $l_{j_v} : s_v = d_{v-1}$, $l_{j_{c_2}+1} : q_{c_2+1} = b_{c_2}$, ..., $l_{j_m} : q_m = b_{m-1}$. Note that r_u is semantically equal to p_{c_1} which, in turn, is semantically equal to a_{c_1}. Thus r_u is semantically equal to a_{c_1}. Similarly, s_v is semantically equal to b_{c_2}. From this it follows that seq_p and seq_q are maximally complete update sequences starting from the entry location of the given program to location *loc* in h along Con'_2. Since $c_0 = d_0$ and p_k and q_m are semantically equal to p and q respectively, we have, by the aliasing theorem, that p and q are aliased to each other at location *loc* in context Con'_2. Similarly, we may show that if pointers p' and q' are aliased in Con'_2, then they are also aliased in Con'_1. Thus the set of aliases of each pointer in $Cl(R)$ are the same in Con'_1 and Con'_2.

Since the set R contains all the lock pointers in the given program, we have that the (must-)aliases of all lock pointers are the same in the two contexts Con'_1 and Con'_2. Also, by our hypothesis, the must-locksets are the same at locations l_{call_1} and l_{call_2} in contexts con_1 and con_2, respectively. Furthermore in extending the contexts con_1 and con_2 to form Con'_1 and Con'_2, respectively, the same locks will be acquired and released during the calls fc_1 and fc_2. Thus by combining the above facts, we have that the must-locksets at each location *loc* of h will be the same in both the contexts Con'_1 and Con'_2. **QED**.

Let P be the Steensgaard closed set $Cl(S)$, where S is the set of lock, thread and function pointers. The above result implies that if during the construction of the CCFG a function call *fcall* is encountered again in a context *con*, then it suffices to explore *fcall* only if either (i) the set of aliases for some pointer $p \in P$, or (ii) the lockset, is different from the instances of *fcall* encountered before.

Our procedure is a worklist-based algorithm shown as Algorithm 1. Starting from the entry location of the given concurrent program, we build a graph over tuples of the form $(con,\ func,\ loc,\ \mathcal{DP})$, where *loc* is the current program location of function *func* in context *con*. Here *con* is defined by means of a call string, i.e., a sequence of function calls leading to *func*. The alias sets and locksets are tracked via the dataflow tuple $\mathcal{DP} = (\mathcal{A}, L)$, where (i) the aliasing relation $\mathcal{A} \subseteq P \times 2^P$ assign to each pointer $p \in P$ the set $\mathcal{A}(p)$ of aliases of p at location *loc* of function *func* in context *con*, and (ii) L is the must-lockset at location *loc* of function *func* in context *con*.

To start with the alias set for each pointer p is set to $\{p\}$ and the lockset is set to the empty set. In each iteration, we delete a tuple $tup = (con, func, loc, \mathcal{DF})$ from worklist W, where $\mathcal{DF} = (\mathcal{A}, L)$. If *loc* has been visited before with the lockset L and aliasing relation $\mathcal{A}$ then, by Thm. 1, exploring the successors of *loc* in *con* with the dataflow tuple $\mathcal{DF}$ cannot lead to the discovery of a new shared variable access with a different lockset. Thus tup needs to be processed only if *loc* hasn't been visited before with the current set of dataflow facts (step 5 of Alg. 1).

In processing *tup* we consider three cases. If loc is the location of a function call *fcall* to function g, say, then in computing the successor of tup, we simply update the current location, function and context while leaving the dataflow facts unchanged (steps 7-9). If, on the other hand, the statement at loc modifies a pointer in P, the aliasing relation needs to be re-computed (steps 11-13). In order to avoid re-computing the aliasing relation from scratch every time we visit loc we leverage the use of summarization (see below). Similarly, if loc is the site of a locking/unlocking statement, then the must-lockset L needs to be updated (steps 14-16).

Termination. In Alg. 1, each function is explored the number of times its entry location is visited with a different dataflow tuple $\mathcal{DF} = (\mathcal{A}, L)$. In the worst case, the number of possible values of $\mathcal{A}$ are $|P|2^{|P|}$, where $|P|$ is the cardinality of the Steensgaard-closed set P that is input to Alg. 1. This is because from the fact that P is Steensgaard-closed it follows that all aliases of a pointer $p \in P$ also belong to P. Thus there are at most $2^{|P|}$ different alias sets possible for p. Also, the number of possible locksets is at most $2^{|\mathcal{L}|}$, where $\mathcal{L}$ is the set of locks in the given program. If $|F|$ is the total number of functions, then the size of resulting CCFG is $O(|F||P|2^{|P|}2^{|\mathcal{L}|})$. In practice, however, the actual locksets as well as possible aliases of thread, function and lock pointers at a given location are few in number.

A Note on Summarization. In step 12 of Alg. 1, we are required to compute the aliases of all pointers in P at location *loc* in context *con*. The number of times we need to carry out this computation equals the number of different contexts in which *loc* is visited. In order to avoid computing the aliases from scratch each time we visit *loc*, we makes use of summarization as formulated in [11]. Here the summary of a function f tracks the start and end points of complete update sequences between pairs of locations of f of the form (m, n), where (i) m is either the entry location of f or the location of a function call in f, (ii) n is either the exit location of f or the location of a function call in f, and (iii) there exists a path in the (local) flow graph of f from m to n that does not involve an intermediate function call location. When building the summaries for f, we do not propagate update sequences to or from functions called in f because in the presence of function pointers we cannot always resolve the functions being called. Then given a context *con*, we compose the local update sequences along all function calls in *con* to compute the entire update sequences starting at the entry location of the given program to *loc* along *con*. This gives us the required aliases. It is worth noting that using bootstrapping we need to build summaries only for pointers in $Cl(S)$, where S is the set of lock, function and thread pointers. Since these pointers are few in number and are updated in a small number of functions, most of the function summaries will in fact be empty. Thus bootstrapping further enhances the effectiveness of summarization in computing FSCS pointer aliases efficiently.

Implicit contexts. Although Alg. 1 enumerates contexts explicitly for the sake of exposition, a practical implementation may not do so for efficiency reasons. In practice, we will number the contexts and all the data flow facts including pointer aliases and locksets will be indexed by the context number they correspond to, at each relevant program location. A context manager can be used to provide the main context-related operations, i.e., (i) context numbering, (ii) mapping contexts to data flow facts, (iii) checking equivalence of contexts, and (iv) adding new contexts for a function.

Preservation of Data Races. If the input P to Alg. 1 is $Cl(S)$, where S is the set of lock, function and thread pointers, then we need to show that the reduced set of contexts that are generated by

Algorithm 1 Finitary Context-Sensitive Call Graph Construction

Input: A Steensgaard closed set P
Initialize $Processed$ to the empty set and worklist W to $\{(\epsilon, start, entry_{start}, \mathcal{DF}_0)\}$, where $entry_{start}$ is the entry location of the entry function *start* of the given concurrent program; ϵ denotes the empty call-string; and $\mathcal{DF}_0 = (\mathcal{A}_0, \emptyset)$ with $\mathcal{A}_0$ being the initial aliasing relation that assigns to each pointer $p \in P$ the set $\{p\}$.
while W is not empty **do**
  Delete tuple $tup = (con, func, loc, \mathcal{DF})$, where $\mathcal{DF} = (\mathcal{A}, L)$, from W
  if a tuple of the form $tup_{match} = (con', func, loc, \mathcal{DF})$ does not belong to *Processed* for any *con'* **then**
    Add $(con, func, loc, \mathcal{DF})$ to *Processed*
    if loc is the site of a function call *fcall* to function g, say, **then**
      $Succ = \{(con.fcall, g, entry_g, \mathcal{DF}) |$ $entry_g$ *is the entry location of* g $\}$
    else
      Set $\mathcal{A}' = \mathcal{A}$ and $L' = L$
      if the program statement at loc modifies a pointer in P **then**
        compute a new aliasing relation $\mathcal{A}'$ by composing summaries for update sequences (see [11])
      end if
      if the statement at loc is a locking/unlocking operation **then**
        construct the new must-lockset L' by updating L
      end if
      Succ $= \{(con, func, loc', (\mathcal{A}', L')) |$ loc' is a successor of loc in $func\}$
    end if
    for each tup' in *Succ* **do**
      Add tup' as a successor of tup in CCFG
      if $tup' \notin Processed \cup W$ then add tup' to W
    end for
  else
    Merge nodes for tup and tup_{match} in CCFG
  end if
end while

Alg. 1 does not cause us to miss any data race. Towards that end we show the following (see [26] for the proof).

Theorem 2 (Soundness). *Let L be the must-lockset at location loc in a valid context con of the given program. Then there exists a valid context con' in the CCFG constructed via Alg. 1 such that the must-lockset at loc in con' is L.*

Proof. Let L be the must-lockset at location *loc* along a path x of the given program in context *con*. We show that there exists a path y in the CCFG leading to *loc* along which the must-lockset at *loc* is L. The proof is by induction on the length of x.

For the base case where the length of x is one, x is comprised only of the initial state of the given concurrent program and so the result holds trivially. For the induction step, suppose that the result holds for all paths of length less than or equal to k and let $x = x_0x_1...x_k$ be a path of length $k+1$. Let loc' be the program location and L' the must-lockset at x_{k-1} along x. Then by the induction hypothesis, there exists a path $y' = y_0...y_l$ of the CCFG leading to *loc'* such that the must-lockset at *loc'* along y' is L' and the aliases of all pointers in the Steensgaard closed set P (input to Alg. 1) at *loc'* are the same along $x' = x_0...x_{k-1}$ and y'. We now consider two cases.

First, we assume that *loc'* is not a function call site, i.e., a location where a function is called. In this case we show that $y = y'.x_k$ is the desired path. We consider two sub-cases. First, assume that *loc* is not a locking/unlocking site. Then the must-locksets at *loc'* and *loc* along x are the same, i.e., L'. Moreover, the must-lockset at x_k along y is the same as the must-lockset at y_l along y', which by the induction hypothesis is L'. If *loc* is a locking/unlocking statement then since the aliases of all pointers in P (including the lock pointers) are the same at *loc'* along both x' and y' we see that the lock that is released or acquired at *loc* along x is the same as the one that is released or acquired at *loc* along y. Thus in both cases the must-lockset at *loc* is the same along both x and y.

Next, we consider the aliases of pointers in P. If the statement at *loc* does not modify any pointer in P, then since P is a Steensgaard closed set, the set of aliases of each pointer in P is the same at *loc* and *loc'*. If, on the other hand, the statement at *loc'* modifies the value of some pointer in P, then we need to show that the set of aliases of each pointer in P are the same at *loc* along both x and y. Towards that end, let p and q be aliased to each other at *loc* along x. By the aliasing theorem (pg. 4), there exist maximal update sequences $seq_p : l_{i_1} : p_1 = a_0, l_{i_2} : p_2 = a_1$,..., $l_{i_k} : p_k = a_{k-1}$ and $seq_q : l_{j_1} : q_1 = b_0, l_{j_2} : q_2 = b_1$,..., $l_{j_m} : q_m = b_{m-1}$ starting from the entry location of the given program and leading to *loc* along x such that $a_0 = b_0$ and p_k and q_m are semantically equal to p and q at locations l_{i_k} and l_{j_m}, respectively. We first consider the case where none of the locations l_{i_k} and l_{j_m} is *loc*. In that case seq_p and seq_q are also maximal update sequences starting from the entry location of the given program and leading to *loc'* along x. Thus p and q are aliased at *loc'* along x and hence, by the induction hypothesis, at *loc'* along y. Since the statement at *loc* cannot extend the update sequences that cause p and q to be aliased at *loc'* along y', p and q are also aliased at *loc* along y. Now consider the case where at least one of l_{i_k} or l_{j_m} is *loc*. Note that both l_{i_k} and l_{j_m} cannot be *loc* as only one variable can be modified at a given location. Assume for definiteness that $l_{i_k} = loc$. Then consider the maximally complete update sub-sequence $seq'_p : l_{i_1} : p_1 = a_0$, $l_{i_2} : p_2 = a_1$,..., $l_{i_{k-1}} : p_{k-1} = a_{k-2}$ of seq_p. Since both $l_{i_{k-1}}$ and l_{j_m} occur before *loc* along x we see that p_{k-1} and q are aliased at *loc'* along x. Then by the induction hypothesis they are aliased at *loc'* along y. Applying the aliasing result again, we have that there exist complete maximal update sequences $\mathsf{seq}'_p : l_{i'_1} : r_1 = c_0$, $l_{i'_2} : r_2 = c_1$,..., $l_{i'_u} : r_u = c_{u-1}$ and $\mathsf{seq}'_q : l_{j'_1} : s_1 = d_0$, $l_{j'_2} : s_2 = d_1$,..., $l_{j'_v} : s_v = d_{v-1}$ starting from the entry location of the given program and leading to *loc'* along y' such that r_u and s_v are semantically equal to p_{k-1} and q_m (which is semantically equivalent to q) at $l_{i'_u}$ and $l_{j'_v}$, respectively. Then the sequence $\mathsf{seq}_p : r_1 = c_0, l_{i'_2} : r_2 = c_1$,..., $l_{i'_u} : r_u = c_{u-1}, l_{i_k} : p_k = a_{k-1}$ is a complete update sequence from the entry location of the given program to *loc* along y. Also, seq_q is a complete update sequence from the entry location of the given program to *loc* along y. Since p_k and q_m are semantically equal to p and q, respectively, we have that p and q are aliased at *loc'* along y.

Finally, we consider the case when *loc'* is a function call site. By the induction hypothesis, the path y' occurs in the CCFG. If the path $y'.x_k$ also occurs in the CCFG then we are done. Else, the only reason why $y'.x_k$ will not be explored is if there exists an alternative path z in CCFG leading to x_k such that the must-lockset at x_k along z is L and the aliases of all pointers in P are the same at *loc*. In that case z is the desired path. This completes the induction step and proves the desired result. **QED**.

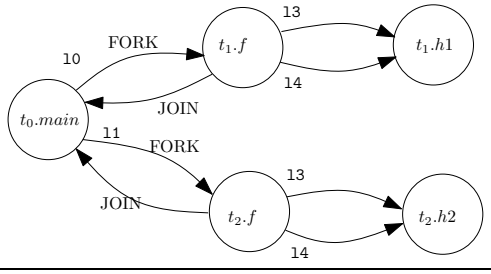

Figure 4. Concurrent call graph for the example in Fig. 1

Since, by Theorem 2, Alg. 1 preserves all possible locksets at locations where shared variables are accessed, it follows immediately that we will not miss any data race.

Example. The concurrent call graph for the example program in Figure 1 is shown in Figure 4. Note that each node in labeled by both the thread identifier and the function name. Multiple calls to other functions from a single function is labeled by the corresponding calling locations in order to distinguish the contexts. The fork and join edges are also explicitly marked.

7. Data Race Analysis

As mentioned in Section 5, the key steps in static data race detection are (i) computing the set of shared variables, (ii) determining the set of must-locksets at each shared variable access in a context-sensitive manner, and (iii) pruning spurious data race warnings by taking into account locksets and program causality constraints. In this section, we will discuss steps (i) and (ii) in some detail and then present a pruning strategy that relies on a may-happen-in-parallel (MHP) analysis.

Shared Variable Detection. In real-life concurrent `C` programs shared variables are typically accessed via pointers. The set of shared variables consists of all variables that are either global variables of threads, aliases of global variables or escaped variables. Note that local pointer variables may also alias global variables and result in a data race violation. However, if we label all local variables aliasing global variables as shared, we will end up with a large number of spurious warnings. Therefore, we are only interested in the set of local pointers that are used to actually update the values of global variables. Towards that end, using the CCFG computed in the previous section, we use a data flow analysis on pointer variables [13] to compute the set of shared variable accesses. The analysis essentially propagates the assignments in complete update sequences from variables p to q (where p is global). If the sequence is followed a modification of some scalar variable via q, then q is marked as shared.

Initial Data-race warnings. After we have computed the set of shared variables, we can conservatively enumerate data race warnings by considering all pairs of accesses to the same shared variable in two different threads, where at least one of the accesses is a write operation. For industrial size programs, this conservative enumeration may lead to thousands of warnings, most of which are bogus. We now describe a staged MHP analysis, which prunes away the spurious warnings by first taking into account lock acquisition patterns and then using thread ordering constraints imposed by *fork* and *join* operations in the CCFG.

7.1 MHP analysis

Lockset analysis. Computing context-sensitive locksets is trivial after computing the context-sensitive CCFG for the program (cf. Algorithm 1): the locksets can be extracted from the data flow facts $\mathcal{DF}$ computed at the shared variable access locations. By combining lockset computation with the call graph computation, we avoid a separate lockset computation phase, required in previous approaches [13]. Then two accesses in different thread may happen in parallel only if the locksets held at their respective locations are disjoint.

Thread order analysis. The order of program statements in the CCFG prohibits some concurrent accesses. For example, consider a concurrent program comprised of two threads: t_0 and t_1, where t_0 creates t_1 and later waits for t_1 to *join*. An access to a shared variable x in thread t_1 cannot happen concurrently with an access to x in t_0 if the access in t_0 follows the join instruction corresponding to t_1. The thread order analysis is designed to reveal the causalities (of above form) in a program that arise due to fork/join constraints as well as (sequential) control flow in threads.

Fork-join model. We start by observing that *fork* and *join* operations enforce the following causality constraints:

1. All statements in the parent thread between matching fork and join points may happen in parallel with all statements in the child thread (and all threads forked thereof)
2. No other statements can happen in parallel.

The thread order analysis exploits the above causality constraints in the fork-join model to compute an MHP set.

Thread Pool model. Recall that in the thread pool model, tasks are dispatched for execution to previously created threads. These calls implicitly correspond to a fork operation. However, these fork operations do not have corresponding join operations. In order to conservatively estimate the effect of these calls, we introduce matching join operations at the exit locations of the start function of the given program.

Furthermore, in contrast to the fork-join model, the thread pool model (by virtue of the work queue) allows multiple function tasks to be executed on the same thread. Since only one task can be executing on a thread at any given time, we add the constraint that no two functions (and any pair of locations in those functions) which always execute on the same thread may happen in parallel. Note that the information about the set of functions that always execute on the same thread is readily available from the CCFG.

Given a thread t, let $children(t)$ denote the set of threads created by t. For each thread t, we denote the first and last program locations in t by $t.start$ and $t.end$ respectively. In general, $t.start$ and $t.end$ are not unique since multiple fork calls can be dispatched to the same thread in the thread pool model. However, given a fork call instruction to thread t, $t.start$ and $t.end$ are uniquely defined. For program locations l and m, we write $l \rightarrow_{fj} m$ to mean that there exists a path in the CCFG of the given program from l to m. Similarly, $l \rightarrow m$ means that there exists a path in the CCFG from l to m along which there exists no fork or join statements.

Algorithm 2 shows the details of the analysis. Intuitively, Loc_{parent} consists of all the locations in the parent thread that are forward reachable from the fork call and backward reachable from the join instruction without any intermediate fork or join edges. The set Loc_{child} consists of all locations in the child thread as well as threads created thereof. The analysis produces the set MHPlocs of location pairs that may happen in parallel.

Based on the thread order analysis, we can now prune the warnings by removing any warning wherein the program location pairs that do not occur in the MHPlocs set.

Example. Recall the example program in Figure 1. As mentioned earlier, the program has two potential data race conditions in the function f on the shared variable pointed to by the argument z. More precisely, the races involve location pairs ($t1.f.l2$, $t2.f.l2$) and ($t1.f.l3$, $t2.f.l3$) since function f is executed by both threads $t1$ and $t2$. However, if we perform thread order analysis on the context-sensitive CCFG obtained from the program, we find that

Algorithm 2 Thread order analysis

MHPlocs := {}
for all Fork-join call pair $fj = (l_{fk}, l_{jn})$ in CCFG **do**
  Loc_{parent} := $\{l \mid l_{fk} \rightarrow l \wedge l \rightarrow l_{jn}\}$
  Loc_{child} := $\{l \mid child.start \rightarrow_{fj} l \wedge l \rightarrow_{fj} child.end\}$
  MHPlocs := MHPlocs $\cup$ $(Loc_{parent} \times Loc_{child})$
end for
Remove all locations in MHPlocs that belong to functions executing on the same thread

the above location pairs are not in the set MHPlocs, since the thread $t2$ can only execute after the thread $t1$ finishes executing.

8. Experiments

The proposed data race detection technique has been implemented in the CoBE framework [12] for analysis of concurrent C programs being developed at NEC. All the experiments were performed on an Intel Core Duo 1.86Ghz machine with 1GB memory, running Linux. In order to evaluate the usefulness of the proposed approach, we applied it to a wide variety of software: open source benchmarks including Linux drivers, `bzip2smp` software, and in-house products including a parallel MPEG decoder **D** and a commercial software system, **S**. Table 1 shows the experimental results. The system **S** was the prime inspiration behind this work since it is based on the thread pool model and employs AFCs as well as locks for synchronization, and also contains recursive functions. The other three benchmarks are chosen to show the effectiveness of the proposed CCFG construction method over the previous method [13], even on programs without function pointers or recursion.

Linux Drivers: In previous work [13], we reported data race detection statistics on a suite of Linux drivers with known data races downloaded from `kernel.org`. The goal of this exercise was to test whether CoBE could detect these known data races. We showed that CoBE could in fact discover all the known data races efficiently while keeping the bogus warnings rate low. The procedure formulated in [13] carried out data race detection by enumerating all contexts in the given program and computing locksets individually for each context, without checking for context equivalence with respect to locksets. The time taken for static warning generation via exhaustive context enumeration is given in col. Ex-Time of Table 1. Using our new CCFG construction that exploits the fix-point criterion to drastically cut down on the contexts that need be explored, we see that the time taken (shown in LFC-Time col.) for data race detection is reduced many-fold. Thus, although the fix-point criterion was proposed for finitizing the CCFG in the presence of function pointers and recursion, it also enhances the scalability of existing approaches by considering only a representative set of contexts necessary for data-race detection. Cols. #War and #Aft. Red. refer, respectively, to the initial number of warnings generated via the lockset method and the number of warnings left after applying warning reduction which combines thread order analysis (Sec. 7.1) with existing reduction schemes [13].

bzip2smp: The `bzip2smp` program parallelizes the `bzip` compression algorithm to work on SMP machines. It uses the `pthread` library as opposed to AFCs. Thus our new framework can handle both the fork/join and the thread pool models in a unified manner.

Parallel Decoder D: D is an in-house parallel implementation of an MPEG-4 decoder. Notice the stark difference between the time taken in exploring all the contexts exhaustively as in [13] and in exploring the contexts enumerated via our new CCFG construction. The dramatic difference is because for efficiency reasons the implementation does not use any locks; all synchronization is via the use of barriers. Thus each function f is explored only once during our CCFG construction (corresponding to a single lockset data fact, see Algorithm 1) as opposed to [13] where it is explored for each different context in which f appears.

NEC Product S: a large in-house concurrent software system, denoted by **S**. The **S** system consists of about 400K lines of C++ code using Boost libraries [23] and is based on a thread pool model. Much of the code deals with asynchronous stream processing, which makes execution flow non-evident. Callback objects are passed as first-class data items. The final execution context for any particular callback object is not evident and often is the source of subtle errors. Changes made in one small routine have often led to bugs occurring much later in code, at a location seemingly unrelated to the original bug location. These bugs are mostly irreproducible due to the complex nature of asynchronous calls and remain unexpressed over long periods of time.

We evaluated our approach by checking for data races in a central module M of **S** (about 10K of C++ code). To this goal, we first built a faithful C model of M. This primarily involved recreating the thread pool model based on the pthread library and building the infrastructure for programming asynchronous function calls (including modeling the *bind* functionality of the Boost library). Based on this infrastructure, we were able to obtain a C model of the module M that closely resembled the actual C++ implementation. We performed data-race detection on this model: we observed that the initial number of data race warnings with the context-sensitive CCFG construction and the lockset based reduction were 524. Exploiting the thread order analysis drastically reduced the number of warnings to 161. Note due to bootstrapping the time taken to analyze this code was only 29 sec. This would not have been possible if we had carried out a whole program FSCS points-to analysis. Thus bootstrapping was key in making our analysis scalable.

9. Related Work and Conclusions

Data race detection being a problem of fundamental interest has been the subject of extensive research. Many techniques have been proposed in order to attack the problem including dynamic run-time detection, static analysis and model checking.

Early work on dynamic data race detection includes the Eraser data race detector [22] which is based on computing locksets. There has been much work that improves upon the basic Eraser methodology. One such approach is [9] which leverages the use of static analysis to reduce the number of data race warnings that need to be validated via a run-time analysis. Other run-time detection tools based on Lamport's happened before model restrict the number of interleavings that need be explored [5, 16]. The advantage of run-time techniques is the absence of false warnings. On the other hand, the disadvantages are the extra cost incurred in instrumenting the code and poor coverage both of which become worse as the size of code increases especially in the context of concurrent programs. Additionally, run time detection techniques presume that the give code can be executed which may not be an option for applications like device drivers.

Recently, there has been a spurt of activity in applying static analysis techniques for data race detection [4, 14, 25, 2, 18, 17, 7, 20, 9]. An advantage of such techniques is that they can be made to scale to large programs. The key disadvantage is that since static analysis works on heavily abstracted versions of the original program, they are not refined enough and can produce a large number of false warnings. Concurrency or MHP analysis has received wide attention [21] for object-oriented programs (see, e.g., [27, 1]). These analyses rely on the fact that at thread creation/join locations, the target thread object/code can be syntactically inferred. In

Driver	KLOC	# *ShVars*	#War	#Aft.Red.	Ex-Time (secs)	LFC-Time (secs)
`hugetlb`	1.2	5	4	1	3.2	1
`ipoib_multicast`	26.1	10	33228	6	7	1.4
`plip`	13.7	17	94	51	5	3
`sock`	0.9	6	32	13	2	1.2
`ctrace_comb`	1.4	19	985	58	6.7	1.1
`autofs_expire`	8.3	7	20	3	6	1.9
`ptrace`	15.4	3	9	1	15	2.4
`tty_io`	17.8	1	6	3	4	0.9
`raid`	17.2	6	23	13	1.5	1.1
`pci_gart`	0.6	1	3	1	1	0.8
`bzip2smp`	6.4	25	15	12	-	23
D (NEC)	2.9	4	256	25	22 min	8 sec
S (NEC)	1.3	12	524	161	-	29 sec

Table 1. Experimental comparison of data-race detection on benchmarks. The column Ex.Time reports the time taken (in seconds) via the exhaustive context enumeration algorithm of [13] whereas the LFC-Time column denotes the time taken for exploring the reduced set of contexts enumerated by our new CCFG construction. The '-' entries in the table indicate that the previous technique [13] could not handle named forks as a result of which experiments could not be carried out on these examples.

contrast, this paper analyzes programs where thread dispatch takes place via indirect calls through function pointers. Moreover, since the exact target depends on the particular calling context, performing race detection on a context-sensitive control flow graph reduces the number of spurious data races significantly.

Verification of particular forms of asynchronous programs has received some attention recently [10, 8]. In both cases, however, asynchronous tasks are executed sequentially and there is no concurrent interaction between the tasks. To the best of our knowledge, no previous method has investigated data race detection for general asynchronous multi-threaded programs with indirect calls.

Emami et al. [6] present an algorithm for context-sensitive interprocedural points-to analysis for recursive sequential programs with calls made via function pointers. However, handling recursion via the introduction of back-edges results in an over-approximation of the set of behaviors of the given program. As a result, the points-to analysis no longer remains fully context-sensitive - a key requirement for static data race detection - leading to bogus warnings. Our key insight, on the other hand, is that even in the presence of recursion and concurrency, it is possible to perform a precise and efficient data race detection by considering a finite representative subset of contexts (instead of over-approximating the set of contexts) for each function to precisely identify the shared variables as well as compute context-sensitive locksets and points-to sets of function/thread pointers for CFG construction.

In this paper, we have presented a method to perform static data race detection for concurrent C programs that use asynchronous indirect function calls for communication - a problem of great practical importance but which has received little attention in the literature. One of our key contributions has been a new technique for context-sensitive CFG construction that guarantees termination even in the presence of recursion and without losing precision of the analysis at hand. This enables us to build a framework for fast and accurate data race detection that can handle concurrent programs with complex programming constructs thereby making such an analysis practical for a larger class of realistic programs. Experiments on real-life open source benchmarks have demonstrated the efficacy of our new technique.

References

[1] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent java programs. In *LCPC*, pages 152–169, 2005.

[2] M. Burrows and K. Leino. Finding stale-value errors in concurrent programs. In *Compaq Systems Research Center SRC-TR-2002-004*, 2002.

[3] P. Chandrasekaran, C. L. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with clarity. In *FSE*, 2007.

[4] D. Detlefs, K. Leino, G. Nelson, and J. Saxe. Extended static checking. In *TR SRC-159 Compaq SRC*, 1998.

[5] A. Dinning and E. Schonberg. An empirical comparision of monitoring algorithms for access anomaly detection. In *PPoPP*, 1990.

[6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. pages 242–256, 1994.

[7] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.

[8] P. Ganty, R. Majumdar, and A. Rybalchenko. Verifying liveness for asynchronous programs. In *POPL*, pages 102–113, 2009.

[9] C. J, K. Lee, A. Loginov, R.O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.

[10] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *POPL*, 2007.

[11] V. Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI*, 2008.

[12] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Semantic reduction of thread interleavings for concurrent programs. In *TACAS*, 2009.

[13] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, 2007.

[14] R. Leino, G. Neslon, and J. Saxe. Esc/java users' manual. In *Technical Note 2000-002, Compaq Systems Research Center*, 2001.

[15] P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI*, 2007.

[16] J. Mellor-Crummey. One-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 Supercomputer Debugging Workshop*, 1991.

[17] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.

[18] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.

[19] C. Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23(4):298–298, 2008.

[20] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-

Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.

[21] M. C. Rinard. Analysis of multithreaded programs. In *SAS*, 2001.

[22] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programming. In *ACM TCS*, volume 15(4), 1997.

[23] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual (With CD-ROM)*. Addison-Wesley Professional, December 2001.

[24] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[25] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, 1993.

[26] F. Version. `www.cs.utexas.edu/users/kahlon/async`.

[27] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. *SIGPLAN Not.*, 38(5):115–128, 2003.

[28] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.