Sparse Flow-Sensitive Pointer Analysis for Multithreaded Programs

Yulei Sui, Peng Di, and Jingling Xue
UNSW Australia

Abstract

For C programs, flow-sensitivity is important to enable pointer analysis to achieve highly usable precision. Despite significant recent advances in scaling flow-sensitive pointer analysis sparsely for sequential C programs, relatively little progress has been made for multithreaded C programs.

In this paper, we present FSAM, a new **F**low-**S**ensitive pointer **A**nalysis that achieves its scalability for large **M**ultithreaded C programs by performing sparse analysis on top of a series of thread interference analysis phases. We evaluate FSAM with 10 multithreaded C programs (with more than 100K lines of code for the largest) from `Phoenix-2.0`, `Parsec-3.0` and open-source applications. For two programs, `raytrace` and `x264`, the traditional data-flow-based flow-sensitive pointer analysis is unscalable (under two hours) but our analysis spends just under 5 minutes on `raytrace` and 9 minutes on `x264`. For the rest, our analysis is 12x faster and uses 28x less memory.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages, Performance

Keywords Pointer Analysis, Sparse Analysis, Flow-Sensitivity

1. Introduction

C, together with its OO incarnation C++, is the de facto standard for implementing system software (e.g., operating systems and language runtimes), server and client applications. A substantial number of these applications are multithreaded in order to better utilize multicore computing resources. However, multithreading poses a major challenge for pointer analysis, since shared memory locations can be accessed non-deterministically by concurrent threads.

Pointer analysis is a fundamental static analysis, on which many other analyses/optimizations are built. The more precisely a pointer is resolved, the more effective the pointer analysis will likely be. By improving its precision and scalability for multithreaded C programs, we can directly improve the effectiveness of many clients, including data race detection [24], deadlock detection [30], compiler optimization reuse [14], control-flow integrity enforcement [8], memory safety verification [20], and memory leak detection [28].

For such client applications operating on C programs, pointer analysis needs to be flow-sensitive (by respecting control flow) in order to achieve highly usable precision. There have been significant recent advances in applying sparse analysis to scale flow-sensitive pointer analysis for sequential C programs [10, 11, 21, 29, 32, 33]. However, applying them directly to their multithreaded C programs using Pthreads will lead to unsound (imprecise) results if thread interference on shared memory locations is ignored (grossly over-approximated). In the case of pointer analysis for OO languages like Java, context-sensitivity instead of flow-sensitivity is generally regarded as essential in improving precision [19, 27, 31]. So far, relatively little progress has been made in improving the scalability of flow-sensitive pointer analysis for multithreaded C programs. Below we describe some challenges and insights for tackling this problem and introduce a sparse approach for solving it efficiently.

1.1 Challenges and Insights

One challenge lies in dealing with an unbounded number of thread interleavings. Two threads interfere with each other when one writes into a memory location that may be accessed by the other. In Figure 1(a), $c = *p$ can load the points-to values from x that are stored into by $*p = r$ in the same (main) thread or $*p = q$ in a parallel thread t. As a result, the points-to set of c is $pt(c) = \{y, z\}$.

In addition, computing sound (i.e., over-approximate) points-to sets flow-sensitively relies on a so-called may-happen-in-parallel (MHP) analysis to discover parallel code regions. Unlike structured languages such as Cilk [25] and X10 [1], which provide high-level concurrency constructs

CGO'16, March 12–18, 2016, Barcelona, Spain
© 2016 ACM. 978-1-4503-3778-6/16/03...$15.00
http://dx.doi.org/10.1145/2854038.2854043

`p = &x; q = &y;`
`r = &z;`
`void main(){`
`  fork(t,foo);`
`  *p = r;`
`  c = *p;`
`}`
 	
`void foo(){`
`  *p = q;`
`}`
	`p = &x; q = &y;`
	`r = &z;`
	`void main(){`
	`  fork(t1,foo);`
	`  join(t1);`
	`  *p = r;`
	`}`
	`void foo(){`
	`  fork(t2,bar);`
	`}`
	`void bar(){`
	`  *p = q;`
	`  c = *p;`
	`}`
		`p = &x; q = &y;`
		`r = &z;`
		`void main(){`
		`  *p = r;`
		`  fork(t,foo);`
		`  join(t);`
		`  c = *p;`
		`}`
		
		`void foo(){`
		`  *p = q;`
		`}`
			`p = &x; q = &y;`
			`r = &z; x = &a;`
			`void main(){`
			`  fork(t,foo);`
			`  c = *p;`
			`}`
			
			`void foo(){`
			`  *p = q;`
			`  *x = r;`
			`}`
				`p = &x;  q = &y;`
				`r = &z;  u = &v;`
				`void main(){`
				`  *p = r;`
				`  fork(t,foo);`
				`  lock(l1); c = *p; unlock(l1);`
				`}`
				`void foo(){`
				`  lock(l2);`
				`  *p = u; *p = q;`
				`  unlock(l2);`
				`}`
				`//l1 and l2 point to same lock`

$pt(c) = \{y, z\}$	$pt(c) = \{y, z\}$	$pt(c) = \{y\}$	$pt(c) = \{y\}$	$pt(c) = \{y, z\}$
(a) Interleaving	(b) Soundness	(c) Precision	(d) Data-flow	(e) Sparsity

Figure 1: Examples for illustrating some challenges faced by flow-sensitive pointer analysis for multithreaded C programs (with irrelevant code elided). For brevity, fork() and join() represent pthread_create() and pthread_join() in the Pthreads API.

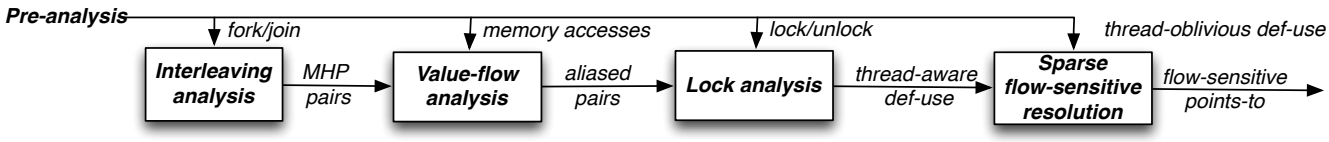

Figure 2: FSAM: a sparse flow-sensitive pointer analysis framework for multithreaded C programs.

with restricted parallelism patterns, unstructured and low-level constructs in the Pthreads API allow programmers to express richer parallelism patterns. However, such flexible non-lexically-scoped parallelism significantly complicates MHP analysis. For example, a thread may outlive its spawning thread or can be joined partially along some program paths or indirectly in one of its child threads. In Figure 1(b), thread $t2$ executes independently of its spawning thread $t1$ and will stay alive even after $t1$ has been joined by the main thread. Thus, $*p = r$ executed in the main thread may interleave with the two statements $*p = q$ and $c = *p$ in `bar()` executed by $t2$. A sound points-to set for c is $pt(c) = \{y, z\}$.

How to maintain precision can also be challenging. Synchronization statements (e.g., fork/join and lock/unlock) must be well tracked to reduce spurious interleavings among non-parallel statements. In Figure 1(c), $*p = r$, $*p = q$, and $c = *p$ are always executed serially in that order. By performing a strong update at $*p = q$ with respect to thread ordering, we can discover that c points to y stored in x by $*p = q$ (not z stored in x at $*p = r$, since x has been strongly updated with $\&y$, killing $\&z$). Thus, $pt(c) = \{y\}$.

How do we scale flow-sensitive pointer analysis for large multithreaded C programs? One option is to adopt a data-flow analysis to propagate iteratively the points-to facts generated at a statement s to every other statement s' that is either reachable along the control flow or may-happen-in-parallel with s, without knowing whether the facts are needed at s' or not. This traditional approach computes and maintains a separate points-to graph at each program point in order to accommodate the side-effects of all parallel threads. Blindly propagating the points-to information this way under all thread interleavings is inefficient in both time and space. In Figure 1(d), $c = *p$ in the main thread can interleave with $*p = q$ and $*x = r$ in thread t. However, propagating the points-to information generated at $*x = r$ to $c = *p$ is not necessary, since $*p$ and $*x$ are not aliases. So $pt(c) = \{y\}$.

Finally, how do we improve scalability by propagating points-to facts along only a set of pre-computed def-use chains sparsely? It turns out that this pre-computation is much more challenging in the multithreaded setting than the sequential setting [10]. Imprecise handling of synchronization statements (e.g., fork/join and lock/unlock) may lead to spurious def-use chains, reducing both the scalability and precision of the subsequent sparse analysis. In Figure 1(e), $pt(c) = \{y, z\}$, if $l1$ and $l2$ are must aliases pointing to the same lock. However, if a pre-computed def-use edge is added from $*u = v$ to $c = *p$, then following this spurious edge makes the analysis not only less efficient but also less precise by concluding that $pt(c) = \{y, z, v\}$ is possible.

1.2 Our Solution

In this paper, we present FSAM, a new **F**low-**S**ensitive pointer **A**nalysis for handling large **M**ultithreaded C programs (using Pthreads). We address the afore-mentioned challenges by performing sparse analysis along the def-use chains precomputed by a pre-analysis and a series of thread interference analysis phases, as illustrated in Figure 2. To bootstrap the sparse analysis, a *pre-analysis* (by applying Andersen's pointer analysis algorithm [2]) is first

performed flow- and context-insensitively to discover over-approximately the points-to information in the program.

Based on the pre-analysis, some *thread-oblivious* def-use edges are identified. Then thread interleavings are analyzed to discover all the missing *thread-sensitive* def-use edges. Our *interleaving analysis* reasons about fork and join operations flow- and context-sensitively to discover may-happen-in-parallel (MHP) statement pairs. Our *value-flow analysis* adds the thread-aware def-use edges for MHP statement pairs with common value flows to produce so-called aliased pairs. Our *lock analysis* analyzes lock/unlock operations flow- and context-sensitively to identify those interfering aliased pairs based on the happen-before relations established among their corresponding mutex regions.

Finally, a *sparse* flow-sensitive pointer analysis algorithm is applied by propagating the points-to facts sparsely along the pre-computed def-use chains, rather than along all program points with respect to the program's control flow.

This paper makes the following contributions:

- We present the first sparse flow-sensitive pointer analysis for unstructured multithreaded C programs.
- We describe several techniques (including thread interference analyses) for pre-computing def-use information so that it is sufficiently accurate in bootstrapping sparse flow-sensitive analysis for multithreaded C programs.
- We show that FSAM (implemented in LLVM (3.5.0)) is superior over the traditional data-flow analysis, denoted NONSPARSE, in terms of scalability on 10 multithreaded C programs from `Phoenix-2.0`, `Parsec-3.0` and open-source applications. For two programs, `raytrace` and `x264`, NONSPARSE is unscalable (under two hours) but FSAM spends just under 5 minutes on `raytrace` and 9 minutes on `x264`. For the remaining programs, FSAM is 12x faster and uses 28x less memory.

2. Background

We introduce the partial SSA form used for representing a C program and sparse pointer analysis in the sequential setting.

2.1 Partial SSA Form

A program is represented by putting it into LLVM's partial SSA form, following [10, 17, 18, 32]. The set of all program variables $\mathcal{V}$ are separated into two subsets: $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer and $\mathcal{T}$ containing all *top-level variables*, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$.

After the SSA conversion, a program is represented by five types of statements: $p = \&a$ (ADDROF), $p = q$ (COPY), $p = *q$ (LOAD), $*p = q$ (STORE), and $p = \phi(q, r)$ (PHI), where $p, q, r \in \mathcal{T}$ and $a \in \mathcal{A}$. Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via LOAD or STORE. For an ADDROF statement $p = \&a$, known as an *allocation site*, a is a stack or global variable with its address taken or a dynamically created abstract heap object (at, e.g., a `malloc()` site).

	p = &a;
p = &a;	***t1 = &b;***
a = &b;	****p = t1;***
q = &c;	***q = &c;***
****p = *q;***	***t2 = *q;***
	****p = t2;***
(a) C code	(b) Partial SSA

Figure 3: A C code fragment and its partial SSA form.

Figure 3 shows a code fragment and its corresponding partial SSA form, where $p, q, t1, t2 \in \mathcal{T}$ and $a, b, c \in \mathcal{A}$. Note that a is indirectly accessed at a store $*p = t1$ by introducing a top-level pointer $t1$ in the partial SSA form. The complex statements like $*p = *q$ are decomposed into the basic ones by introducing a top-level pointer $t2$.

2.2 Sparse Flow-Sensitive Pointer Analysis For Sequential C Programs

The traditional data-flow-based flow-sensitive pointer analysis computes and maintains points-to information at every program point with respect to the program's control flow. This is costly as it propagates points-to information blindly from each node in the CFG of the program to its successors without knowing if the information will be used there or not.

To address the scalability issue in analyzing large sequential C programs, sparse analysis [10] is proposed by staging the pointer analysis: the def-use chains in a program are first approximated by applying a fast but imprecise pre-analysis (e.g., Andersen's analysis) and the precise flow-sensitive analysis is conducted next by propagating points-to facts only along the pre-computed def-use chains *sparsely*.

The core representation of sparse analysis is a *def-use graph*, where a node represents a statement and an edge between two nodes e.g., $s_1 \overset{v}{\hookrightarrow} s_2$ represents a def-use relation for a variable $v \in \mathcal{V}$, with its def at statement s_1 and its use at statement s_2. This representation is sparse since the intermediate program points between s_1 and s_2 are omitted.

In partial SSA form, the uses of any top-level pointer have a unique definition (with ϕ functions inserted at confluence points as is standard). A def-use $s_1 \overset{t}{\hookrightarrow} s_2$, where $t \in \mathcal{T}$, can be found easily without requiring pointer analysis.

As address-taken variables are not (yet) in SSA form, their indirect uses at loads may be defined indirectly at multiple stores. Their def-use chains are built in several steps following [10], as illustrated in Figure 4. We go through a sequence of steps needed in building the def-use chains for $a \in \mathcal{A}$. The def-use chains for $b \in \mathcal{A}$ are built similarly.

First, indirect defs and uses (i.e., may-defs and may-uses) are exposed at loads and stores, based on the points-to information obtained during the pre-analysis (Figure 4(a)). A load, e.g., $s = *r$ is annotated with a function $\mu(a)$, where

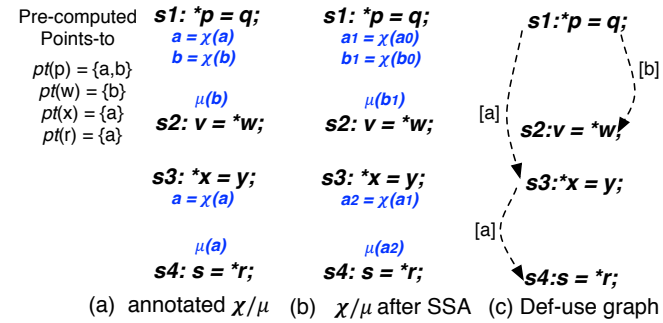

Figure 4: A sparse def-use graph.

$$[\text{T-FORK}]\quad \frac{t \xRightarrow{(c,fk_i)} t' \quad t' \xRightarrow{(c',fk_{i'})} t''}{t \xRightarrow{(c,fk_i)} t''}$$

$$[\text{T-JOIN}]\quad \frac{t \xLeftarrow{(c,jn_i)} t' \quad t' \xLeftarrow[\text{full}]{(c',jn_{i'})} t''}{t \xLeftarrow{(c,jn_i)} t''}$$

$$[\text{T-SIBLING}]\quad \frac{t'' \xLeftarrow{(c,fk_i)} t \quad t'' \xLeftarrow{(c',fk_{i'})} t' \quad (i \neq i' \vee c \neq c')}{t \bowtie t'}$$

Figure 5: Static modeling of fork and join operations.

$a \in \mathcal{A}$ may be pointed to by r to represent a potential use of a at the load. Similarly, a store, e.g., $*x = y$ is annotated with a function $a = \chi(a)$ to represent a potential def and use of a at the store. If a can be strongly updated, then a receives whatever y points to and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a weak update to a. Third, each address-taken variable, e.g., a is converted into SSA form (Figure 4(b)), with each $\mu(a)$ treated as a use of a. and each $a = \chi(a)$ as both a def and use of a. Finally, an indirect def-use chain of a is added from a definition of a identified as a_n (version n) at a store to its uses at a store or a load, resulting in two indirect def-use edges of a i.e. $s_1 \xhookrightarrow{a} s_3$ and $s_3 \xhookrightarrow{a} s_4$ (Figure 4(c)). Any ϕ function introduced for an address-taken variable a during the SSA conversion will be ignored as a is not versioned.

Every callsite is also annotated with μ and χ functions to expose its indirect uses and defs. As is standard, passing arguments into and returning results from functions are modeled by copies. So the def-use chains across the procedural boundaries are added similarly. For details, we refer to [10].

Once the def-use chains are in place for the program, flow-sensitive pointer analysis can be performed sparsely, i.e., by propagating points-to information only along these pre-computed def-use edges. For example, the points-to sets of a computed at s_1 are propagated to s_3 with s_2 bypassed, resulting in significant savings both time and memory.

3. The FSAM Approach

We first describe a static thread model used for handling fork and join operations (Section 3.1). We then introduce our FSAM framework (Figure 2), focusing on how to precompute def-use chains (Sections 3.2 and 3.3) and discussing thereafter on how to perform the subsequent sparse analysis for multithreaded C programs (Section 3.4).

3.1 Static Thread Model

Abstract Threads A program starts its execution from its *main* function in the *main (root) thread*. An *abstract thread* t refers to a call of `pthread_create()` at a context-sensitive fork site during the analysis. Thus, a thread t always refers to a context-sensitive fork site, i.e., a unique runtime thread unless t is multi-forked, in which case, t may represent more than one runtime thread.

Definition 1 (Multi-Forked Threads)**.** *A thread $t \in \mathcal{M}$ is a* multi-forked *thread if its fork site, say, fk_i resides in a loop, a recursion cycle, or its spawner thread $t' \in \mathcal{M}$.*

Intra-Thread CFG For an abstract thread t, its intra-thread control flow graph, ICFG_t, is constructed as in [15], where a node s represents a program statement and an edge from s_1 to s_2 signifies a possible transfer of control from s_1 to s_2. For convenience, a call site is split into a call node and a return node. Three kinds of edges are distinguished: (1) an intra-procedural control flow edge $s \rightarrow s'$ from node s to its successor s', (2) an interprocedural call edge $s \xrightarrow{call_i} s'$ from a call node s to the entry node s' of a callee at callsite i, and (3) an interprocedural return edge $s \xrightarrow{ret_i} s'$ from an exit node s of a callee to the return node s' at callsite i.

There are no outgoing edges for a fork or join site. Function pointers are resolved by pre-analysis.

Modeling Thread Forks and Joins Figure 5 gives three rules for modeling fork and join operations statically. We write $t \xRightarrow{(c,fk_i)} t'$ to represent the spawning relation that a *spawner thread* t creates a *spawnee thread* t' at a context-sensitive fork site (c, fk_i), where c is a context stack represented by a sequence of callsites, $[cs_0, \cdots, cs_n]$, from the entry of the main function to the fork site fk_i. Note that the callsites inside each strongly-connected cycle in the call graph of the program are analyzed context-insensitively.

For a thread t forked at (c, fk_i), we write $\mathcal{S}_t$ to stand for its start procedure, where the execution of t begins. $Entry(\mathcal{S}_t) = (c', s)$ maps $\mathcal{S}_t$ to its first statement (c', s), where $c' = c.push(i)$, context-sensitively.

Consider the three rules in Figure 5. The spawning relation $t \xRightarrow{(c,fk_i)} t'$ is transitive, representing the fact that t can create t' directly or indirectly at a fork site fk_i (`[T-FORK]`).

We will handle only the join operations identified by `[T-JOIN]` and ignore the rest in the program. The joining relation $t \xLeftarrow{(c,jn_i)} t'$ indicates that a spawnee t' is joined by its spawner t at a join site (c, jn_i) As our pre-analysis is

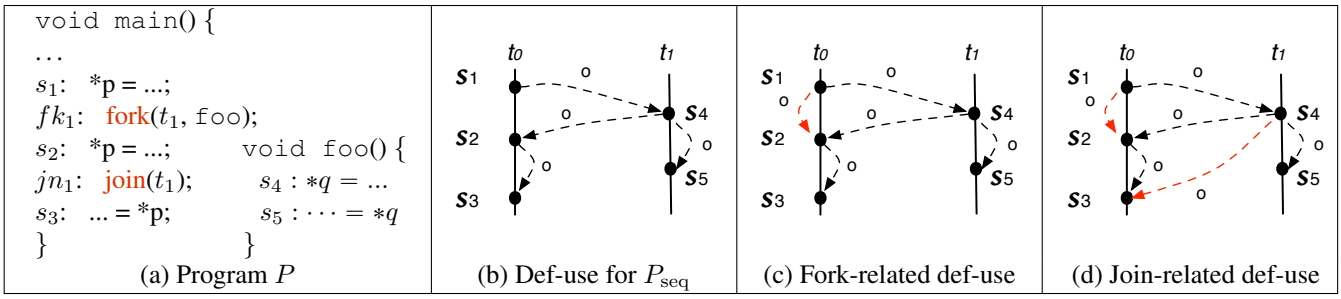

Figure 6: Thread-oblivious def-use edges (where p and q are found to point to o during the pre-analysis).

flow- and context-insensitive, we achieve soundness by requiring t' joined at a join site `pthread_join()` in the program to be excluded from $\mathcal{M}$, so that t' represents a unique runtime thread (under all contexts). Note that the joining relation is not transitive in the same sense as the spawning relation. In Pthreads programs, a thread can be joined fully along all program paths or partially along some but not all paths. Given $t \xleftarrow{(c,jn_i)} t'$ and $t' \xleftarrow{(c',jn_{i'})} t''$, $t \xleftarrow{(c,jn_i)} t''$ holds when $t' \xleftarrow{(c',jn_{i'})} t''$ is a full join, denoted $t' \xleftarrow[\text{full}]{(c',jn_{i'})} t''$.

If neither $t \xRightarrow{(c,fk_i)} t'$ nor $t' \xRightarrow{(c',fk_{i'})} t$ holds, then t and t' are *siblings*, denoted $t \bowtie t'$ (`[T-SIBLING]`). In this case, t and t', where $t \neq t'$, share a common ancestor thread t''. Furthermore, t and t' do not happen-in-parallel if one happens before the other (as defined below).

Definition 2 (Happens-Before (HB) Relation for Sibling Threads)**.** *Given two sibling threads t and t', t happens before t', denoted $t \succ t'$, if the fork site of t' is backward reachable to a join site of t along every program path.*

Presently, FSAM does not model other synchronization constructs such as barriers and signal/wait, resulting in sound, i.e., over-approximate results.

3.2 Computing Thread-Oblivious Def-Use Chains

Given a multithreaded C program P, we transform P into a sequential version P_{seq}, representing one possible thread interleaving of P. We then derive the def-use chains from P_{seq} as the thread-oblivious def-use chains for P, based on the points-to information obtained during the pre-analysis.

There are three steps, illustrated in Figure 6.

In Step 1, we transform P into P_{seq} by replacing every fork statement fk in P by calls to the start procedures of all the threads spawned at fk. Let S_{fk} be the set of these start procedures. We keep the join operations that we can handle by `[T-JOIN]` and ignore the rest. We then follow [10] to compute its def-use chains for P_{seq}, as discussed in Section 2.2. Given P in Figure 6(a), where $S_{fk1} = \{\texttt{foo}\}$, we obtain its P_{seq} by replacing `fork(t,foo)` with a call to `foo()`. The def-use chains for P_{seq} are given in Figure 6(b).

In Step 2, we add the fork-related missing def-use edges at a fork site fk by assuming $S_{fk} = \varnothing$, since every start procedure, say, *fun*, in S_{fk} may be executed nondeterministically later. This means that any value flow added in Step 1 that go through fun can also bypass it altogether. For every such a def-use chain that starts at a statement s before the callsite for fun and ends at a statement s' after, we add a def-use edge from s to s'. (Technically, a weak update is performed for every $a = \chi(a)$ function associated with the callsite for fun, so that the old contents of a prior to the call are preserved.) In our example Figure 6(b) becomes Figure 6(c) with the fork-related def-use edge $s1 \overset{o}{\hookrightarrow} s2$ being added.

In Step 3, we deal with every direct join operation handled by our static thread model (`[T-JOIN]`). Let join(t') be a candidate join site executed in the spawner thread t, which implies, as discussed in Section 3.1, that t' is a unique runtime thread to be joined. Let *fun*$'$ be the start procedure of t'. In one possible thread interleaving, this join statement plays a similar role as an exception-catching statement for an exception thrown at the end of fun'. Given this implicit control flow, we need to make the modification side effects of *fun*$'$ visible at the join site. Let $a \in \mathcal{A}$ be an address-taken variable defined at the exit of fun'. For the first use of a reachable from the join site along every program path in ICFG$_t$, we add a def-use edge from that definition to the use. In our example, Figure 6(c) becomes Figure 6(d) with the join-related def-use edge $s4 \overset{o}{\hookrightarrow} s3$ added.

3.3 Computing Thread-Aware Def-Use Chains

For a program P, we must also discover the def-use chains formed by all the other thread interleavings except P_{seq}. Such def-use chains are *thread-aware* and computed with the three thread interference analyses incorporated in FSAM.

3.3.1 Interleaving Analysis

As shown in Figure 2, FSAM invokes this as the first of the three interference analyses to compute thread-aware def-use chains. The objective here is to reason about fork and join operations to identify all MHP statements in the program.

Our interleaving analysis operates flow- and context-sensitively on the ICFGs of all the threads (but uses points-to information from the pre-analysis). For a statement s in thread t's ICFG$_t$, our analysis approximates which threads may run in parallel with t when s is executed, denoted as

$$\texttt{[I-DESCENDANT]}\quad \frac{t \xRightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \rightarrow (t,c,s) \quad (c',s') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,s) \quad \{t\} \subseteq \mathcal{I}(t',c',s')}$$

$$\texttt{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c,s) = Entry(\mathcal{S}_t) \quad (c',s') = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',s') \quad \{t'\} \subseteq \mathcal{I}(t,c,s)}$$

$$\texttt{[I-JOIN]}\quad \frac{t \xLeftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}} \qquad \texttt{[I-CALL]}\quad \frac{(t,c,s) \xrightarrow{call_i} (t,c',s') \quad c' = c.push(i)}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c',s')}$$

$$\texttt{[I-INTRA]}\quad \frac{(t,c,s) \rightarrow (t,c,s')}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c,s')} \qquad \texttt{[I-RET]}\quad \frac{(t,c,s) \xrightarrow{ret_i} (t,c',s') \quad i = c.peek() \quad c' = c.pop()}{\mathcal{I}(t,c,s) \subseteq \mathcal{I}(t,c',s')}$$

Figure 7: Interleaving analysis (where $\rightarrow$ denotes a control flow edge in a thread's ICFG introduced Section 3.1).

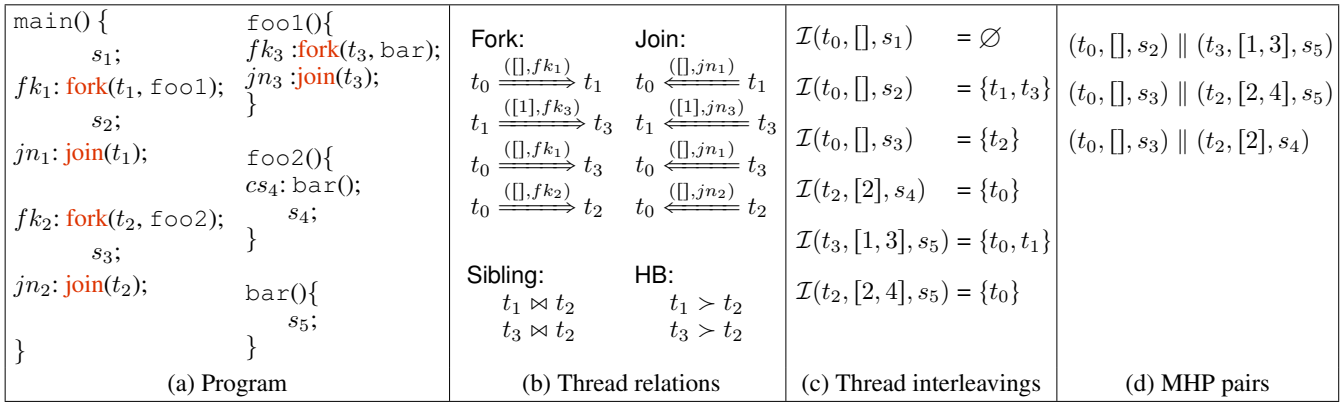

Figure 8: An illustrating example for interleaving analysis (with t_0 denoting the main thread).

$\mathcal{I}(t,c,s)$, where c is a calling context to capture one instance of s when its enclosing method is invoked under c. For example, if $\mathcal{I}(t_1,c,s) = \{t_2,t_3\}$, then threads t_2 and t_3 may be alive when s_1 is executed under context c in t_1.

Statement s_1 in thread t_1 may happen in parallel with statement s_2 in thread t_2, denoted as $(t_1,c_1,s_1) \parallel (t_2,c_2,s_2)$, if the following holds (with $\mathcal{M}$ from Definition 1):

$$\begin{cases} t_2 \in \mathcal{I}(t_1,c_1,s_1) \wedge t_1 \in \mathcal{I}(t_2,c_2,s_2) & \text{if } t_1 \neq t_2 \\ t_1 \in \mathcal{M} & \text{otherwise} \end{cases}$$

Given $\xRightarrow{(c,fk_i)}$ (spawning relation), $\xLeftarrow{(c,jn_i)}$ (joining relation), $\bowtie$ (thread sibling) and $\succ$ (HB from Definition 2), our interleaving analysis is formulated as a forward data-flow problem $(V,\sqcap,F)$ (Figure 7). Here, V represents the set of all thread interleaving facts, $\sqcap$ is the meet operator ($\cup$), and $F : V \rightarrow V$ represents the set of transfer functions associated with each node in an ICFG.

[I-DESCENDANT] handles thread creation $t \xRightarrow{(c,fk_i)} t'$ at a fork site (c,fk_i). The statement (c,s) that appears immediately after (c,fk_i) in ICFG$_t$ may-happen-in-parallel with the entry statement (c',s') of the start procedure of thread t'.

Given two sibling threads t and t', the entry statements (c,s) and (c',s') of their start procedures may interleave with each other if neither $t \succ t'$ nor $t' \succ t$ ([I-SIBLING]).

[I-JOIN] represents the fact that a descendent thread will no longer be alive after it has been joined at a join site.

For a thread t, [I-CALL] and [I-RET] ([I-INTRA]) propagate data-flow facts interprocedurally by matching calls and returns context-sensitively (intraprocedurally).

Example 1. *We illustrate our interleaving analysis with a program in Figure 8. As shown in Figure 8(a), the main thread t_0 creates two threads t_1 and t_2 at fork sites fk_1 and fk_2, respectively. In its start procedure* foo1*, t_1 spawns another thread t_3 and fully joins it later at jn_3. Figure 8(b) shows all the thread relations. Note that t_2 continues to execute after its two sibling threads t_1 and t_3 have terminated due to jn_1, which joins t_1 directly and t_3 indirectly.*

The results of applying the rules in Figure 7 are listed in Figure 8(c). Due to context-sensitivity, our analysis has identified precisely the three MHP relations given in Figure 8(d). As bar() *is called under two contexts, s_5 has two different instances $(t_3,[1,3],s_5)$ and $(t_2,[2,4],s_5)$. The former*

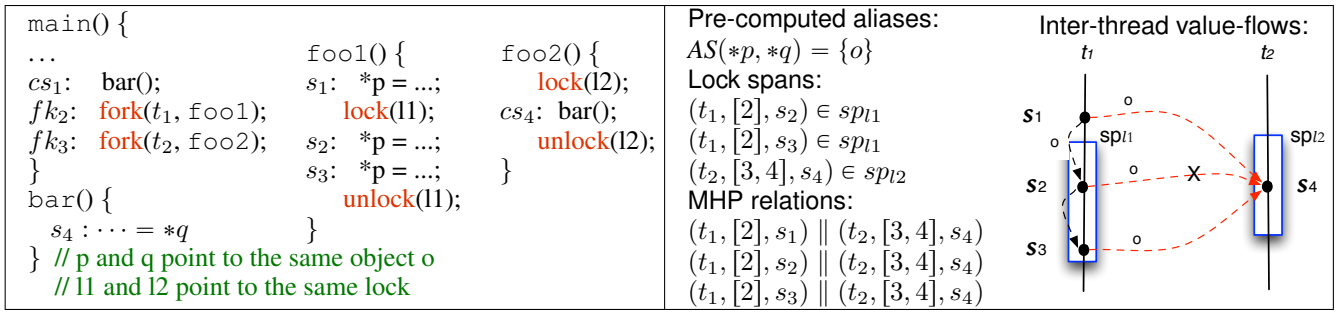

Figure 9: A lock analysis example (with irrelevant code elided), avoiding $s2 \overset{o}{\hookrightarrow} s4$ that would be added by [THREAD-VF].

one may-happen-in-parallel with $(t_0, [], s_2)$ *and the later one with* $(t_0, [], s_3)$. *As our analysis is context-sensitive,* $(t_0, [], s_3) \parallel (t_2, [2], s_4)$ *but* $(t_0, [], s_2) \nparallel (t_2, [2], s_4)$.

3.3.2 Value-Flow Analysis

Given a pair of MHP statements, we make use of the points-to information discovered during the pre-analysis to add the potential (thread-aware) def-use edges in between. In partial SSA form, the top-level pointers in $\mathcal{T}$ are kept in registers and thus thread-local. However, the address-taken variables in $\mathcal{A}$ can be accessed by concurrent threads via loads and stores. It is only necessary to consider inter-thread value-flows for MHP store-load and store-store pairs $[(t, c, s), (t', c', s')$, where s is a store $*p = \ldots$ and s' is a load $\cdots = *q$ or a store $*q = \cdots$. Hence, [THREAD-VF] comes into play, where $AS(*p, *q)$ is the set of objects in $\mathcal{V}$ pointed to by both p and q (due to pre-analysis).

$$\text{[THREAD-VF]}\quad \frac{s : *p = \_ \quad s' : \_ = *q \text{ or } *q = \_ \quad (t, c, s) \parallel (t', c', s') \quad o \in AS(*p, *q)}{s \overset{o}{\hookrightarrow} s'}$$

Example 2. *For the program in Figure 6(a), we apply* [THREAD-VF] *to add all the missing thread-aware def-use chains on top of Figure 6(d). According to pre-analysis,* $\text{AS}(*p, *q) = \{o\}$. *As* $(t_0, [], s_2) \parallel (t_1, [1], s_4)$, $s_2 \overset{o}{\hookrightarrow} s_4$ *is added. As* $(t_0, [], s_2) \parallel (t_1, [1], s_5)$, $s_2 \overset{o}{\hookrightarrow} s_5$ *is added. While* $(t_1, [1], s_4) \parallel (t_0, [], s_2)$, $s_4 \overset{o}{\hookrightarrow} s_2$ *has been added earlier as a thread-oblivious def-use edge (Section 3.2).*

3.3.3 Lock Analysis

Statements from different mutex regions are interference-free if these regions are protected by a common lock. By capturing lock correlations, we can avoid some spurious def-use edges introduced by [THREAD-VF] in the two lock-release spans defined below. We do this by performing a flow- and context-sensitive analysis for lock/unlock operations (based on the points-to information from pre-analysis)

Definition 3 (Lock-Release Spans)**.** *A* lock-release span sp_l *at a context-sensitive lock site* $(t, c, lock(l))$ *consists of the statements starting from* $(c, lock(l))$ *to the corresponding release site* $(c', unlock(l'))$ *in* $ICFG_t$ *obtained with a forward reachability analysis with calls and returns being matched context-sensitively, where* l *and* l' *points to the same singleton (i.e., runtime) lock object, denoted as* $l \equiv l'$.

Just in the case of MHP analysis for fork/join operations, context-sensitivity ensures that lock analysis can distinguish different calling contexts under which a statement appears inside a lock-release span. In Figure 9, bar() is called twice, but only the instance of statement $(t_2, [3, 4], s_4)$ called from cs_4 is inside the lock-release span sp_{l2}.

Definition 4 (Span Head)**.** *For an object* $o \in \mathcal{A}$, $HD(sp_l, o)$ *represents a set of context-sensitive loads or stores that may access* o *at the head of the span* sp_l*:* $HD(sp_l, o) = \{(t, c, s) \in sp_l \mid \nexists\, (t', c', s') \in sp_l : s' \overset{o}{\hookrightarrow} s\}$.

Definition 5 (Span Tail)**.** *For an object* $o \in \mathcal{A}$, $TL(sp_l, o)$ *represents a set of context-sensitive stores that may access* o *at the tail of the span* sp_l*:* $TL(sp_l, o) = \{(t, c, s) \in sp_l \mid s \text{ is a store}, \nexists\, (t', c', s') \in sp_l : (s' \text{ is a store} \wedge s \overset{o}{\hookrightarrow} s')\}$.

Definition 6 (Non-Interference Lock Pairs)**.** *Let* $(t, c, s) \parallel (t', c', s')$ *be a MHP statement pair, where* s *is a store, such that both statements are protected by at least one common lock, i.e.,* $\exists\, l, l' : (t, c, s) \in sp_l \wedge (t', c', s') \in sp_{l'} \implies l \equiv l'$. *We say that the pair is a* non-interference lock pair *if* $(t, c, s) \notin TL(sp_l, o) \vee (t', c', s') \notin HD(sp_{l'}, o)$ *holds.*

By refining [THREAD-VF] with Definition 6 being taken into account, some spurious value-flows are filtered out.

Example 3. *In Figure 9, two lock-release spans* sp_{l1} *and* sp_{l2} *are protected by a common lock, since* $*l1$ *and* $*l2$ *are found to be must aliases. By applying* [THREAD-VF] *alone, all the three def-use edges in red will be added. By Definition 6, however,* s_2 *inside* sp_{l1} *cannot interleave with* s_4 *inside* sp_{l2}. *So* $s_2 \overset{o}{\hookrightarrow} s_4$ *is spurious and can be ignored.*

3.4 Sparse Analysis

Once all the def-use chains have been built, the sparse flow-sensitive pointer analysis algorithm developed for sequential C programs [10], given in Figure 10, can be reused in the

$$\text{[P-ADDR]}\ \frac{s : p = \&o}{\{o\} \subseteq pt(s,p)} \qquad \text{[P-COPY]}\ \frac{s : p = q \quad s' \overset{q}{\hookrightarrow} s}{pt(s',q) \subseteq pt(s,p)}$$

$$\text{[P-PHI]}\ \frac{s : p = \phi(q,r) \quad s' \overset{q}{\hookrightarrow} s \quad s'' \overset{r}{\hookrightarrow} s}{pt(s',q) \subseteq pt(s,p) \quad pt(s'',r) \subseteq pt(s,p)}$$

$$\text{[P-LOAD]}\ \frac{s : p = *q \quad s'' \overset{q}{\hookrightarrow} s \quad o \in pt(s'',q) \quad s' \overset{o}{\hookrightarrow} s}{pt(s',o) \subseteq pt(s,p)}$$

$$\text{[P-STORE]}\ \frac{s : *p = q \quad s'' \overset{p}{\hookrightarrow} s \quad o \in pt(s'',p) \quad s' \overset{q}{\hookrightarrow} s}{pt(s',q) \subseteq pt(s,o)}$$

$$\text{[P-SU/WU]}\ \frac{s : *p = \_ \qquad s' \overset{o}{\hookrightarrow} s \qquad o \in \mathcal{A} \backslash kill(s,p)}{pt(s',o) \subseteq pt(s,o)}$$

$$kill(s,p) = \begin{cases} \{o'\} & \text{if } pt(s,p) = \{o'\} \wedge (o' \in \textit{singletons}) \\ \mathcal{A} & \text{else if } pt(s,p) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Figure 10: Sparse flow-sensitive pointer analysis.

multithreaded setting. For a variable v, $pt(s,v)$ denotes its points-to set computed immediately after statement s.

The first five rules deal with the five types of statements introduced in Section 2.1, by following the pre-computed def-use chains $\hookrightarrow$. The last enables a strong or weak update at a store, whichever is appropriate, where *singletons* [17] is the set of objects in $\mathcal{A}$ representing unique locations by excluding heap, array, and local variables in recursion.

FSAM is sound since (1) its pre-analysis is sound, (2) the def-use chains constructed for the program (as described in Sections 3.2 and 3.3) are over-approximate, and (3) the sparse analysis given in Figure 10 is as precise as the traditional iterative data-flow analysis [10].

4. Evaluation

The objective is to show that our sparse flow-sensitive pointer analysis, FSAM, is significantly faster than while consuming less memory than the traditional data-flow-based flow-sensitive pointer analysis, denoted NONSPARSE, in analyzing large multithreaded C programs using Pthreads.

4.1 Experimental Setup

We have selected a set of 10 multithreaded C programs, including the two largest (`word_count` and `kmeans`) from `Phoenix-2.0`, the five largest (`radiosity`, `ferret`, `bodytrack`, `raytrace` and `x264`) from `Parsec-3.0`, and three open-source applications (`automount`, `mt_daapd` and `httpd-server`), as shown in Table 1. All our experiments were conducted on a platform consisting of a

Table 1: Program statistics.

Benchmark	Description	LOC
word_count	Word counter based on map-reduce	6330
kmeans	Iterative clustering of 3-D points	6008
radiosity	Graphics	12781
automount	Manage autofs mount points	13170
ferret	Content similarity search server	15735
bodytrack	Body tracking of a person	19063
httpd_server	Http server	52616
mt_daapd	Multi-threaded DAAP Daemon	57102
raytrace	Real-time raytracing	84373
x264	Media processing	113481
Total		380659

2.70GHz Intel Xeon Quad Core CPU with 64 GB memory, running Ubuntu Linux (kernel version 3.11.0).

The source code of each program is compiled into bit code files using clang and then merged together using LLVM Gold Plugin at link time stage (LTO) to produce a whole-program bc file. In addition, the compiler option mem2reg is turned on to promote memory into registers.

```
// word_count-pthread.c
140 for(i=0; i<num_procs; i++){
166   pthread_create(&tid[i], &attr,
                wordcount_map, (void*)out) != 0);
167 }
170 for (i = 0; i < num_procs; i++){
173   pthread_join(tid[i],
                (void **)(void*)&ret_val) != 0);
175 }
    ...
```

Figure 11: A multi-forked example in `word_count`.

4.2 Implementation

We have implemented FSAM in LLVM (version 3.5.0). Andersen's analysis (using the constraint resolution techniques from [23]) is used to perform its pre-analysis indicated in Figure 2. In order to distinguish the concrete runtime threads represented by an abstract multi-forked thread (Definition 1) inside a loop, we use LLVM's SCEV alias analysis to correlate a fork-join pair. Figure 11 shows a code snippet from `word_count`, where a fixed number of threads are forked and joined in two "symmetric" loops. FSAM can recognize that any statement in a slave thread (with its start routine `wordcount_map`) does not happen in parallel with the statements after its join executed in the main thread.

FSAM is field-sensitive. Each field of a struct is treated as a separate object, but arrays are considered monolithic. Positive weight cycles (PWCs) that arise from processing fields are detected and collapsed [22]. The call graph of a program is constructed on-the-fly. Distinct allocation sites are modeled by distinct abstract objects [10, 32].

4.3 Methodology

We are not aware of any flow-sensitive pointer analysis for multithreaded C programs with Pthreads in the literature or any publicly available implementation. RR [25] is closest; it performs an iterative flow-sensitive data-flow-based pointer analysis on structured parallel code regions in Clik programs. However, C programs with Pthreads are unstructured, requiring MHP analysis to discover their parallel code regions. PCG [14] is a recent MHP analysis for Pthreads that distinguishes whether two procedures may execute concurrently. We have implemented RR also in LLVM (3.5.0) for multithreaded C programs with their parallel regions discovered by PCG, denoted NONSPARSE, as the base line.

To understand FSAM better, we also analyze the impact of each of its phases on the performance of sparse flow-sensitive points-to resolution. To do this, we measure the slowdown of FSAM with each phase turned off individually: (1) No-Interleaving: with our interleaving analysis turned off but the results from PCG used instead, (2) No-Value-Flow: with our value-flow analysis turned off (i.e., $o \in AS(*p, *q)$ in `[THREAD-VF]` disregarded), and (3) No-Lock: with our lock analysis turned off.

Note that some spurious def-use edges may be avoided by more than one phase. Despite this, these three configurations allow us to measure their relative performance impact.

4.4 Results and Analysis

Table 2 gives the analysis times and memory usage of FSAM against NONSPARSE. FSAM spends less than 22 minutes altogether in analyzing all the 10 programs (totaling 380KLOC). For the two largest programs, `raytrace` and `x264`, FSAM spends just under 5 and 9 minutes, respectively, while NONSPARSE fails to finish analyzing each under two hours. For the remaining 8 programs analyzable by both, FSAM is 12x faster and uses 28x less memory than NONSPARSE, on average. For the two programs with over 50KLOC, `httpd_server` and `mt_daapd`, FSAM is 11x faster and uses 117x less memory for `httpd_server` and 29x faster and uses 89x less memory for `mt_daapd`.

For small programs, such as `word_count` and `kmeans`, FSAM yields little performance benefits over NONSPARSE due to relatively few statements and simple thread synchronizations used. For larger ones, which contain more pointers, loads/stores and complex thread synchronization primitives, FSAM has a more distinct advantage, with the best speedup 39x observed at `bodytrack` and the best memory usage reduction at `httpd_server`. FSAM has achieved these better results by propagating and maintaining significantly less points-to information than NONSPARSE.

Figure 12 shows the relative impact of each of FSAM's three thread interference analysis phases on its analysis efficiency for the three configurations defined in Section 4.3. The performance impact of each phase varies considerably across the programs evaluated. On average, value-flow analysis is more beneficial than the other two in reducing spurious def-use edges passed to the final sparse analysis.

Table 2: Analysis time and memory usage.

Program	Time (Secs)		Memory (MB)	
	FSAM	NONSPARSE	FSAM	NONSPARSE
word_count	3.04	17.40	13.79	53.76
kmeans	2.50	18.19	18.27	53.19
radiosity	6.77	29.29	38.65	95.00
automount	8.66	83.82	27.56	364.67
ferret	13.49	87.10	52.14	934.57
bodytrack	128.80	2809.89	313.66	12410.16
httpd_server	191.22	2079.43	55.78	6578.46
mt_daapd	90.67	2667.55	37.92	3403.26
raytrace	284.61	OOT	135.06	OOT
x264	531.55	OOT	129.58	OOT

Interleaving analysis is very useful for `kmeans`, `httpd-server` and `mt_daapd` in avoiding spurious MHP pairs. These programs adopt the master-slave pattern so that the slave threads perform their tasks in their start procedures while the master thread handles some post-processing task after having joined all the slave threads. Precise handling of join operations is critical in avoiding spurious MHP relations between the statements in the slave threads and those after their join sites in the master thread.

Value-flow analysis is effective in reducing redundant def-use edges among concurrent threads in most of the programs evaluated. For `automount`, `ferret` and `mt_daapd`, value-flow analysis has avoided adding over 80% (spurious) def-use edges. In these programs, the concurrent threads manipulate not only global variables but also their local variables frequently. Thus, value-flow-analysis can prevent the subsequent sparse analysis from propagating blindly a lot of points-to information for non-shared memory locations.

Lock analysis is beneficial for programs such as `automount` and `radiosity` that have extensive usage of locks (with hundreds of lock-release spans) to protect their critical code sections. In these program, some lock-release spans can cover many statements accessing globally shared objects. Figure 13 gives a pair of lock-release spans with a common lock accessing the shared `global`'s `task_queue` in two threads. The spurious def-use chains from the write at line 457 in `dequeue_task` to all the statements accessing the shared `task_queue` object in `enqueue_task` are avoided by our analysis.

5. Related Work

We discuss the related work on sparse flow-sensitive pointer analysis and pointer analysis for multithreaded programs.

Sparse Flow-Sensitive Pointer Analysis Sparse analysis, a recent improvement over the classic iterative data-flow approach, can achieve flow-sensitivity more efficiently by

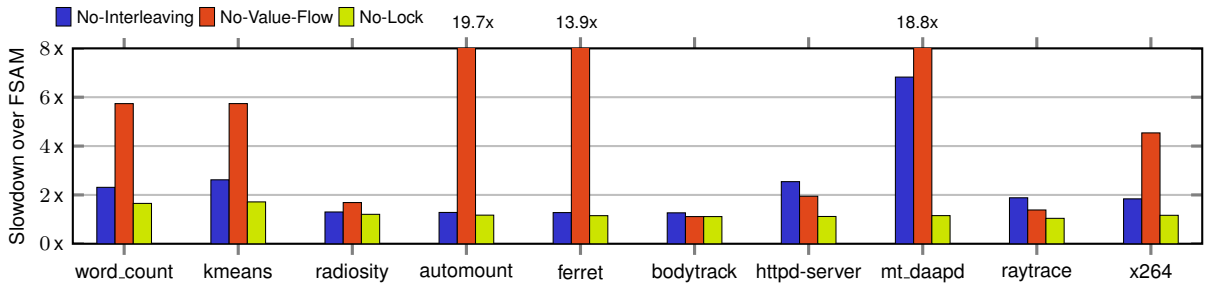

Figure 12: Impact of FSAM's three thread interference analysis phases on its analysis efficiency.

```
// taskman.C
377 void enqueue_task(long qid, Task *task,
                long  mode) {
382     tq = &global->task_queue[ qid ] ;
385     LOCK(tq->q_lock);
387     if( tq->tail == 0 )     // read
390         tq->tail = task ;  // write
              ......
412     UNLOCK(tq->q_lock);
413 }
418 Task *dequeue_task(long qid, long max_visit,
                 long process_id){
443     tq = &global->task_queue[ qid ] ;
449     LOCK(tq->q_lock);
              .....
457     tq->tail = NULL ;       // write
470     tq->tail = prev ;       // write
              .....
475     UNLOCK(tq->q_lock);
494 }
```

Figure 13: Effectiveness of lock analysis for `radiosity`.

propagating points-to facts sparsely across pre-computed def-use chains [10, 11, 21]. Initially, sparsity was experimented with in [12, 13] on a Sparse Evaluation Graph [4], a refined CFG with irrelevant nodes removed. On various SSA form representations (e.g., factored SSA [5], HSSA [6] and partial SSA [16]), further progress was made later. The def-use chains for top-level pointers, once put in SSA form, can be explicitly and precisely identified, giving rise to a semi-sparse flow-sensitive analysis [11]. Recently, the idea of staged analysis [9, 10] that uses pre-computed points-to information to bootstrap a later more precise analysis has been leveraged to make pointer analysis full-sparse for both top-level and address-taken variables [10, 21, 29, 33].

Pointer Analysis for Multithreaded Programs This has been an area that is not well studied and understood due to the challenges discussed in Section 1.1. Earlier, Rugina and Rinard [25] introduced a pointer analysis for Clik programs with structured parallelism. They solved a standard data-flow problem to propagate points-to information iteratively along the control flow and evaluated their analysis with benchmarks with up to 4500 lines of code.

However, unstructured multithreaded C or Java programs are more challenging to analyze due to the use of non-lexically-scoped synchronization statements (e.g., fork/join and lock/unlock). For Java programs, a compositional approach [26] analyzes pointer and escape information of variables in a method that may be escaped and accessed by other threads. The approach performs a flow-sensitive lock-free analysis to analyze each method modularly but iteratively without considering strong updates. The proposed approach was evaluated on six small benchmarks (with up 18K lines of bytecode). To maintain scalability for large Java programs, modern pointer analysis tools for Java embrace context-sensitivity instead of flow-sensitivity [27, 31].

However, flow-sensitivity is important to achieve precision required for C programs. To the best of our knowledge, this paper presents the first sparse flow-sensitive pointer analysis for C programs using Pthreads. The prior analyses on handling thread synchronizations are conservative, by ignoring locks [26] or joins [14] or dealing with only partial and/or nested joins [3]. In contrast, FSAM models such synchronization operations more accurately, by building on our recent work on MHP analysis [7], to produce the first multithreaded flow-sensitive points-to analysis that scales successfully to programs up to 100K lines of code.

6. Conclusion

We have designed and implemented FSAM, a new sparse flow-sensitive pointer analysis for multithreaded C programs and demonstrated its scalability over the traditional data-flow approach. Some further details can be found in its artifact. In future work, we plan to evaluate the effectiveness of FSAM in helping some bug-detection tools in detecting concurrency bugs such as data races and deadlocks in multithreaded C programs. We also plan to combine FSAM with some dynamic analysis tools such as Google's ThreadSanitizer to reduce their instrumentation overhead.

Acknowledgments

We thank all the reviewers for their constructive comments on an earlier version of this paper. This research is supported by ARC grants DP130101970 and DP150102109.

References

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP '07*, pages 183–193.

[2] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, 1994.

[3] R. Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC '05*, pages 152–169.

[4] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91*, pages 55–66.

[5] J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, 1994.

[6] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267.

[7] P. Di, Y. Sui, D. Ye, and J. Xue. Region-based may-happen-in-parallel analysis for c programs. In *ICPP '15*, pages 889–898.

[8] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS '15*, pages 901–913.

[9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008.

[10] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298.

[11] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238.

[12] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[13] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98*, pages 57–81.

[14] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL'11*, pages 623–636.

[15] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural aliasing. In *PLDI '92*, pages 235–248.

[16] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86.

[17] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16.

[18] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353.

[19] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP' 14*, pages 27–53.

[20] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *SNAPL '15*, pages 190–208.

[21] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI '12*, pages 229–238.

[22] D. Pearce, P. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.

[23] F. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135.

[24] P. Pratikakis, J. S. Foster, and M. W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI' 06*, pages 320–331.

[25] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *PLDI '99*, pages 77–90.

[26] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *PPOPP '01*, 36(7):12–23.

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. POPL '11, pages 17–30.

[28] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264.

[29] Y. Sui, S. Ye, J. Xue, and P.-C. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse ssa. In *APLAS '11*, pages 155–171.

[30] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI' 08*, pages 281–294.

[31] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI '04, pages 131–144.

[32] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336.

[33] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229.

A. Artifact Description

Summary: The artifact includes full implementation of FSAM and NONSPARSE analyses, benchmarks and scripts to reproduce the data in this paper.

Description: You may find the artifact package and all the instructions on how to use FSAM via the following link: `http://www.cse.unsw.edu.au/~corg/fsam`

A brief checklist is as follows:

- `index.html`: the detailed instructions for reproducing the experimental results in the paper.
- `FSAM.ova`: virtual image file (4.6G) containing installed Ubuntu OS and FSAM project.
- Full source code of FSAM developed on top of the SVF framework `http://unsw-corg.github.io/SVF`.
- Scripts used to reproduce the data in the paper including `./table2.sh` and `./figure12.sh`.
- Micro-benchmarks to validate pointer analysis results.

Platform: All the results related to analysis times and memory usage in our paper are obtained on a 2.70GHz Intel Xeon Quad Core CPU running Ubuntu Linux with 64GB memory. For the VM image, we recommend you to allocate at least 16GB memory to the virtual machine. The OS in the virtual machine image is Ubuntu 12.04. A VirtualBox with version 4.1.12 or newer is required to run the image.

License: LLVM Release License (The University of Illinois/NCSA Open Source License (NCSA))