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 Flow-Sensitive pointer Analysis that achieves its scalability for large Multithreaded 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 \( *q = r \) 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...
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 Flow-Sensitive pointer analysis for handling large Multithreaded C programs (using Pthreads). We address the aforementioned 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 an-
alyzed 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 state-
ment pairs with common value flows to produce so-called
aliased pairs. Our lock analysis analyzes lock/unlock opera-
tions flow- and context-sensitively to identify those interfering
aliased pairs based on the happen-before relations estab-
lished 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 pro-
gram 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 inter-
  ference 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
  NONPARSE, 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, NONPARSE 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),
\( p = s q \) (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).

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 analy-
thesis 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 sequen-
tial C programs, sparse analysis [10] is proposed by stag-
ing 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., \( s_1 \xrightarrow{p} s_2 \) 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 \( \phi \) functions inserted at confluence
points as is standard). A def-use \( s_1 \xrightarrow{p} s_2 \), 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
$\alpha \in \mathcal{A}$ may be pointed to by $r$ to represent a potential use of $a$ at the load. Similarly, a store, e.g., $*x = \gamma$ is annotated with a function $\alpha = \chi(a)$ 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 $a_{0}$ (version $v$) at a store to its uses at a store or a load, resulting in two indirect def-use edges of $a$ i.e. $s_1 \rightarrow a \rightarrow s_3$ and $s_3 \rightarrow a \rightarrow s_4$ (Figure 4(c)). Any def function introduced for an address-taken variable $a$ during the SSA conversion will be ignored as $a$ is not versioned.

Every call site is also annotated with $\mu$ and $\chi$ 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 pre-compute 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, $f_k$, 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, CFG$_t$, 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 $s \rightarrow s'$ from a call node $s$ to the entry node $s'$ of a callee at call site $i$, and (3) an interprocedural return edge $s \rightarrow s'$ from an exit node $s$ of a callee to the return node $s'$ at call site $i$.

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 $t \xrightarrow{(c,f_k)} t'$ to represent the spawning relation that a spawner thread $t$ creates a spawnee thread $t'$ at a context-sensitive fork site $(c, f_k)$, where $c$ is a context stack represented by a sequence of callsites, $[c_{s_0}, \ldots, c_{s_n}]$, from the entry of the main function to the fork site $f_k$. 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 $(c, f_k)$, we write $S_t$ for its fork site and ignore the rest in the program. The joining relation $t \xrightarrow{(c,jn_i)} t'$ indicates that a spawner thread is joined by its spawner at a join site $(c, jn_i)$ as our pre-analysis is
Given all program paths or partially along some but not all paths. Note that the joining
gram to be excluded from
threads spawned at
fork statement
in
in the pro-
control flow, we need to make the modification side effects
an exception thrown at the end of
fun
implies, as discussed in Section 3.1, that
t
is a unique run-
time thread to be joined. Let
fun
be the start procedure of
t.

In Step 3, we deal with every direct join operation han-
dled by our static thread model ((T-JOIN)). Let
join(t’)
be a candidate join site executed in the spawner thread
which, implies, as discussed in Section 3.1, that
t’
is a unique run-
time 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 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,
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
s1 \rightarrow s2
being 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
t’s
ICFG,
our analysis approximates which threads
may run in parallel with
t
when
s
is executed, denoted as

\begin{align*}
\text{void main()} \{ \\
\quad \ldots \\
\quad s_1: \quad \ast p = \ldots \\
\quad f_{k1}: \quad \text{fork}(t_1, \text{foo}); \\
\quad s_2: \quad \ast p = \ldots \\
\quad j_{n1}: \quad \text{join}(t_1); \\
\quad s_3: \quad \ldots = \ast p; \\
\quad s_4: \quad \ast q = \ldots \\
\quad \}
\end{align*}

\begin{align*}
\text{(a) Program } P \\
\text{(b) Def-use for } P_{seq} \\
\text{(c) Fork-related def-use} \\
\text{(d) Join-related def-use}
\end{align*}

Figure 6: Thread-oblivious def-use edges (where
p
and
q
are found to point to
o
during the pre-analysis).
For example, if our interleaving analysis is formulated as a forward data-flow problem \( (V, \cap, F) \) (Figure 7). Here, \( V \) represents the set of all thread interleaving facts, \( \cap \) is the meet operator \((\cup)\), and \( F : V \rightarrow V \) represents the set of transfer functions associated with each node in an ICFG.

\[ \begin{align*}
\text{[I-Descendant]} & \quad t \xrightarrow{(c,fk_1)} t' \quad (t, c, f k_1) \rightarrow (t, c, s) \quad (c', s') = \text{Entry}(S_{t'}) \\
& \quad \{ t' \} \subseteq \mathcal{I}(t, c, s) \quad \{ t \} \subseteq \mathcal{I}(t, c', s') \\
\text{[I-Sibling]} & \quad t \bowtie t' \quad (c, s) = \text{Entry}(S_t) \quad (c', s') = \text{Entry}(S_{t'}) \\
& \quad \{ t \} \subseteq \mathcal{I}(t', c', s') \quad \{ t' \} \subseteq \mathcal{I}(t, c, s) \\
\text{[I-Join]} & \quad t \xrightarrow{(c,jn_1)} t' \\
& \quad \mathcal{I}(t, c, jn_1) = \mathcal{I}(t, c, jn_1) \setminus \{ t' \} \\
\text{[I-Call]} & \quad (t, c, s) \xrightarrow{\text{call}, i} (t', c', s') \\
& \quad (t, c, s) \subseteq \mathcal{I}(t, c, s) \\
\text{[I-Return]} & \quad (t, c, s) \xrightarrow{\text{return}, i} (t', c', s') \\
& \quad (t, c, s) \subseteq \mathcal{I}(t, c, s)
\end{align*} \]

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).

main() {
  foo1() {
    \( f k_3 : \text{fork}(t_3, \text{bar}); \)
  }
  \( f k_1 : \text{fork}(t_1, \text{foo1}); \)
  \( j n_3 : \text{join}(t_3); \)
  \( s_1; \)
  \( j n_1 : \text{join}(t_1); \)
  \( s_2; \)
  \( f k_2 : \text{fork}(t_2, \text{foo2}); \)
  \( s_4; \)
  \( j n_2 : \text{join}(t_2); \)
  \( s_5; \)
}

(a) Program

Fork:
\[ \begin{align*}
& t_0 \xrightarrow{\text{fork}} t_1 \\
& t_0 \xrightarrow{\text{fork}} t_2 \\
& t_0 \xrightarrow{\text{fork}} t_3
\end{align*} \]

Join:
\[ \begin{align*}
& t_0 \xleftarrow{\text{join}} t_1 \\
& t_0 \xleftarrow{\text{join}} t_2 \\
& t_0 \xleftarrow{\text{join}} t_3
\end{align*} \]

HB:
\[ \begin{align*}
& t_1 \bowtie t_2 \\
& t_1 > t_2 \\
& t_3 > t_2
\end{align*} \]

(b) Thread relations

(c) Thread interleavings

(d) MHP pairs

Given \( t_2 \in \mathcal{I}(t_1, c_1, s_1) \wedge t_1 \in \mathcal{I}(t_2, c_2, s_2) \) if \( t_1 \neq t_2 \), \( t_1 \in \mathcal{M} \) otherwise.

Given \( (c, fk_1) \) (spawning relation), \( (c, jn_1) \) (joining relation), \( \bowtie \) (thread sibling) and \( > \) (HB from Definition 2), our interleaving analysis is formulated as a forward data-flow problem \( (V, \cap, F) \) (Figure 7). Here, \( V \) represents the set of all thread interleaving facts, \( \cap \) 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 \( t \xrightarrow{(c,fk_1)} t' \) at a fork site \((c, fk_1)\). The statement \((c, s)\) that appears immediately after \((c, fk_1)\) in ICFG, 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 \( t > t' \) nor \( t' > t \) ([**I-Sibling**]).

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

For a thread \( t \), [**I-Call**] and [**I-Return**] ([**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 \( f k_1 \) and \( f k_2 \), respectively. In its start procedure \( \text{foo1}, t_1 \) spawns another thread \( t_3 \) and fully joins it later at \( j n_3 \). 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 \( j n_1 \), which joins \( t_2 \) 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 \( \text{bar()} \) 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
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, \((t_0, [], s_3) \parallel \langle t_2, [2], s_4 \rangle\) but \((t_0, [], s_2) \parallel \langle t_2, [2], s_4 \rangle\).

### 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 \(T\) are kept in registers and thus thread-local. However, the address-taken variables in \(A\) can be accessed by concurrent threads via loads and stores. It is only necessary to consider inter-thread value-flow s for MHP store-load and store-store pairs \(\langle t, c, s \rangle, \langle t', c, s' \rangle\), where \(s\) is a store \(\ast p = \ldots\) and \(s'\) is a load \(\ast q = \ldots\). Hence, \([\text{THREAD-VF}]\) comes into play, where \(AS(\ast p, \ast q)\) is the set of objects in \(V\) pointed to by both \(p\) and \(q\) (due to pre-analysis).

\[
\begin{align*}
\text{[THREAD-VF]} & \\
\frac{s : \ast p = \ldots \quad s' : \ast q = \ldots}{(t, c, s) \parallel \langle t', c, s' \rangle \in AS(\ast p, \ast q)}
\end{align*}
\]

**Example 2.** For the program in Figure 6(a), we apply \([\text{THREAD-VF}]\) to add all the missing thread-aware def-use chains on top of Figure 6(d). According to pre-analysis, \(AS(\ast p, \ast q) = \{o\}\). As \((t_0, [], s_2) \parallel \langle t_1, [1], s_4 \rangle\), \(s_2 \stackrel{\alpha}{\rightarrow} s_4\) is added. As \((t_0, [], s_2) \parallel \langle t_1, [1], s_3 \rangle\), \(s_2 \stackrel{\alpha}{\rightarrow} s_3\) is added. While \((t_1, [1], s_4) \parallel \langle t_0, [], s_2 \rangle\), \(s_4 \stackrel{\alpha}{\rightarrow} s_2\) 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 \([\text{THREAD-VF}]\) 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 \(sp_1\) 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\(_1\) 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 \(l = l'\).

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 \((t_2, [3], s_4)\) called from \(cs_4\) is inside the lock-release span \(sp_{12}\).

**Definition 4 (Span Head).** For an object \(o \in A\), \(HD(sp_1, o)\) represents a set of context-sensitive loads or stores that may access \(o\) at the head of the span \(sp_1\): \(HD(sp_1, o) = \{(t, c, s) \in sp_1 \mid \exists (t', c', s') \in sp_1 : s' \stackrel{\alpha}{\rightarrow} o\}\).

**Definition 5 (Span Tail).** For an object \(o \in A\), \(TL(sp_1, o)\) represents a set of context-sensitive stores that may access \(o\) at the tail of the span \(sp_1\): \(TL(sp_1, o) = \{(t, c, s) \in sp_1 \mid \exists s, (t', c', s') \in sp_1 : (s' is a store \wedge s \stackrel{\alpha}{\rightarrow} o)\}\).

**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., \(\exists l, l' \in (t, c, s) \in sp_1 \wedge (t', c', s') \in sp_1, l = l'\). We say that the pair is a non-interference lock pair if \((t, c, s) \notin TL(sp_1, o) \wedge (t', c', s') \notin TL(sp_1, o)\).

By refining \([\text{THREAD-VF}]\) with Definition 6 being taken into account, some spurious value-flows are filtered out.

**Example 3.** In Figure 9, two lock-release spans \(sp_{11}\) and \(sp_{12}\) are protected by a common lock, since \(*l1\) and \(*l2\) are found to be must aliases. By applying \([\text{THREAD-VF}]\) alone, all the three def-use edges in red will be added. By Definition 6, however, \(s_2\) inside \(sp_{11}\) cannot interleave with \(s_4\) inside \(sp_{12}\). So \(s_2 \stackrel{\alpha}{\rightarrow} s_4\) 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
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 \( \rightarrow \). The last enables a weak or strong update at a store, whichever is appropriate, where \( \text{singlets} \) is the set of objects in \( A \) 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 NONS\textsc{parse}, 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 (\texttt{word\_count} and \texttt{kmeans}) from Phoenix-2.0, the five largest (\texttt{radiosity}, \texttt{ferret}, \texttt{bodytrack}, \texttt{raytrace} and \texttt{x264}) from Parsec-3.0, and three open-source applications (\texttt{automount}, \texttt{mt\_daapd} and \texttt{httpd\_server}), as shown in Table 1. All our experiments were conducted on a platform consisting of a 2.7GHz 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.

```c
// word\_count-pthread.c
140 for(i=0; i<num\_procs; i++){
166  pthread\_create(&tid[i], &attr, 
167                  word\_count\_map, (void*)out) != 0);
167 }
170 for (i = 0; i < num\_procs; i++){
173  pthread\_join(tid[i], 
175     (void**)(&ret\_val) != 0);
175 }
...
```

Figure 11: A multi-forked example in \texttt{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 \texttt{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 \texttt{word\_count\_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.

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...
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


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))