Parallel Pointer Analysis with CFL-Reachability

Yu Su  Ding Ye  Jingling Xue  
School of Computer Science and Engineering  
UNSW Australia  
{ysu, dye, jingling}@cse.unsw.edu.au  

Abstract—This paper presents the first parallel implementation of pointer analysis with Context-Free Language (CFL) reachability, an important foundation for supporting demand queries in compiler optimisation and software engineering. Formulated as a graph traversal problem (often with context- and field-sensitivity for desired precision) and driven by queries (issued often in batch mode), this analysis is non-trivial to parallelise.

We introduce a parallel solution to the CFL-reachability-based pointer analysis, with context- and field-sensitivity. We exploit its inherent parallelism by avoiding redundant graph traversals with two novel techniques, data sharing and query scheduling. With data sharing, paths discovered in answering a query are recorded as shortcuts so that subsequent queries will take the shortcuts instead of re-traversing its associated paths. With query scheduling, queries are prioritised according to their statically estimated dependences so that more redundant traversals can be further avoided. Evaluated using a set of 20 Java programs, our parallel implementation of CFL-reachability-based pointer analysis achieves an average speedup of 16.2X over a state-of-the-art sequential implementation on 16 CPU cores.

I. INTRODUCTION

Pointer (or alias) analysis, which is an important component of an optimising or parallelising compiler, determines statically what a pointer may point to at runtime. It also plays an important role in debugging, verification and security analysis. Much progress has been made to improve the precision and efficiency of pointer analysis, across several dimensions, including field-sensitivity (to match field accesses), context-sensitivity (to distinguish calling contexts) and flow-sensitivity (to consider control-flow). In the case of object-oriented programs, e.g., Java programs, both context- and field-sensitivity are considered to be essential in achieving a good balance between analysis precision and efficiency [5].

Traditionally, Andersen’s algorithm [2] has been frequently used to discover the points-to information in a program (e.g., in C and Java). Andersen’s analysis is a whole-program analysis that computes the points-to information for all variables in the program. This algorithm has two steps: finding the constraints that represent the pointer-manipulating statements and propagating these constraints until a fixed point is reached.

For many clients, such as debugging [17], [18], [19] and alias disambiguation [21], however, the points-to information is needed on-demand only for some but not all variables in the program. As a result, there has been a recent focus on demand-driven pointer analysis, founded on Context-Free Language (CFL) reachability [15], which only performs the necessary work to compute the points-to information requested in a graph representation of the program [6], [16], [17], [18], [19], [26], [27]. By computing the points-to information of some (instead of all variables as in whole-program Andersen’s analysis), demand-driven analysis can be performed both context- and field-sensitively to achieve better precision more scalably than Andersen’s analysis, especially for Java programs.

However, precise CFL-reachability-based analysis can still be costly when applied to large programs. In the sequential setting, there are efforts for improving its efficiency for Java programs by resorting to refinement [18], [19], summarisation [17], [26], incrementalisation [6], [16], pre-analysis [25] and ad-hoc caching [18], [25]. However, multicore platforms, which are ubiquitous nowadays, have never been exploited for accelerating the CFL-reachability-based analysis.

There have been a few parallel implementations of Andersen’s pointer analysis on multi-core CPUs [3], [8], [9], [14], GPUs [7], [10], and heterogeneous CPU-GPU systems [20], as compared in detail in Table II. As Andersen’s analysis is a whole-program pointer analysis, these earlier solutions cannot be applied to parallelise demand-driven pointer analysis.

Despite the presence of redundant graph traversals both within and across the queries, parallelising CFL-reachability-based pointer analysis poses two challenges. First, parallelism-inhibiting dependences are introduced into its various analysis stages since processing a query involves matching calling contexts and handling heap accesses via aliased base variables (e.g., p and q in x = p.f and q.f = y). Second, the new points-to/alias information discovered in answering some queries during graph traversals is not directly available to other queries on the (read-only) graph representation of the program.

In this paper, we describe the first parallel implementation of this important analysis for Java programs (although the proposed techniques apply equally well to C programs [27]). We expose its inherent parallelism by reducing redundant graph traversals using two novel techniques, data sharing and query scheduling. With data sharing, we recast the original graph traversal problem into a graph rewriting problem. By adding new edges to the graph representation of the program to shortcut the paths traversed in a query, we can avoid re-traversing (redundantly) the same paths by taking their shortcuts instead when handling subsequent queries. With query scheduling, we prioritise the queries to be issued (in batch mode) according to their statically estimated dependences so that more redundant graph traversals can be further reduced.

While this paper focuses on CFL-reachability-based pointer analysis, the proposed techniques for data sharing and scheduling are expected to be useful for parallelising other graph-
reachability-based analyses (e.g., bug detection [22], [23]).

In summary, this paper makes the following contributions:

- The first parallel implementation of context- and field-sensitive pointer analysis with CFL-reachability;
- A data sharing scheme to reduce redundant graph traversals (in all query-processing threads) by graph rewriting;
- A query scheduling scheme to eliminate unnecessary traversals further, by prioritising queries according to their statically estimated dependencies; and
- An evaluation with a set of 20 Java benchmarks, showing that our parallel solution achieves an average speedup of 16.2X over a state-of-the-art sequential implementation (with 16 threads) on 16 CPU cores.

The rest of this paper is organised as follows. Section II introduces the intermediate representation used for analysing Java programs and reviews CFL-reachability-based pointer analysis. Section III describes our parallel solution in terms of our data sharing and query scheduling schemes. Section IV evaluates and analyses our solution. Section V discusses the related work. Section VI concludes the paper.

II. BACKGROUND

Section II-A describes the intermediate representation used for analysing Java programs. Section II-B reviews the standard formulation of pointer analysis in terms of CFL-reachability.

A. Program Representation

We focus on Java although this work applies equally well to C [27]. A Java program is represented as a Pointer Assignment Graph (PAG), as defined in Fig. 1.

A node $n$ represents a variable $v$ or an object $o$ in the program, where $v$ can be local $(l)$ or global $(g)$. An edge $e$ represents a statement in the program oriented in the direction of its value flow. An edge connects only to local variables ($l_1$ and/or $l_2$) unless it represents an assignment involving at least one global variable (assign$^g$). Let us look at the seven types of edges in detail. $l_1 \xrightarrow{\text{assign}^g} l_2$ captures the flow of object $o$ to variable $l_1$, indicating that $l_1$ points directly to $o$. $l_1 \xrightarrow{\text{assign}} l_2$ represents a local assignment ($l_1 = l_2$). A global assignment is similar except that one or both variables at its two sides are static variables in a class (i.e. $g$). $l_1 \xrightarrow{\text{ld}(f)} l_2$ represents a load of the form $l_1 = l_2.f$ and $l_1 \xrightarrow{\text{st}(f)} l_2$ represents a store of the form $l_2.f = l_2$. $l_1 \xrightarrow{\text{param}} l_2$ models parameter passing, where $l_2$ is an actual parameter and $l_1$ is its corresponding formal parameter, at call site $i$. Similarly, $l_1 \xrightarrow{\text{ret}} l_2$ indicates an assignment of the return value in $l_2$ to $l_1$ at call site $i$.

Fig. 2 gives an illustrating example and its PAG representation. Note that $o_i$ denotes the object created at the allocation site in line $i$ and $v_m$ represents variable $v$ declared in method $m$. Loads and stores to array elements are modeled by collapsing all elements into a special field, denoted arr.

B. CFL-Reachability-based Pointer Analysis

CFL-reachability [15] is an extension of traditional graph reachability. Let $G$ be a directed graph with edges labelled by letters over an alphabet $\Sigma$ and $L$ be a CFL over $\Sigma$. Each path $p$ in $G$ is composed of a string $s(p)$ of $\Sigma^*$, formed by concatenating in order the labels of edges along $p$. A path $p$ is an $L$-path if $s(p) \in L$. Node $x$ is $L$-reachable to $y$ iff a path $p$ from $x$ to $y$ exists, such that $s(p) \in L$. For a single-source reachability analysis, the worst-case time complexity is $O(\Gamma^3N^3)$, where $\Gamma$ is the size of a normalised grammar for $L$ and $N$ is the number of nodes in $G$.

1) Field-sensitivity: Let us start with a field-sensitive formulation without context-sensitivity. When calling contexts are ignored, there is no need to distinguish the four types of assignments, assign$^1$, assign$^g$, param, and ret$^i$; they are all identified as assignment edges of type assign.

For a PAG $G$ with only new and assign edges, the CFL $L_{FT}$ (FT for flows to) is a regular language, meaning that a variable $v$ is $L_{FT}$-reachable from an object $o$ in $G$ iff $o$ can flow to $v$. The rule for $L_{FT}$ is defined as:

$$\text{flowsTo} \rightarrow \text{new} \ (\text{assign})^*$$ (1)

with flowsTo as the start symbol. If $o$ flowsTo $v$, then $v$ is $L_{FT}$-reachable from $o$. For example, in Fig. 2(b), since $o_{15} \xrightarrow{\text{new}} v_{l_{\text{main}}}, l_{\text{thisVector}}, o_{15} \leftrightarrow v_{\text{thisVector}}$.

When field accesses are also handled, the CFL $L_{FS}$ (FS for field-sensitivity) is defined as follows:

$$\text{flowsTo} \rightarrow \text{new} \ (\text{assign} \ | \ \text{st}(f) \ | \ \text{ld}(f))^*$$

$$\text{alias} \rightarrow \text{flowsTo} \text{flowsTo}$$

$$\text{flowsTo} \rightarrow (\text{assign} \ | \ \text{ld}(f) \ | \ \text{st}(f))^* \text{new}$$ (2)

The rule for flowsTo matches fields via st$(f)$ alias ld$(f)$, where st$(f)$ and ld$(f)$ are like a pair of parentheses [19]. For a pair of statements $q.f = y$ (st$(f)$) and $x = p.f$ (ld$(f)$), if $p$ and $q$ are aliases, then an object $o$ that flows into $y$ can flow first through this pair of parentheses (i.e. st$(f)$ and ld$(f)$) and then into $x$. For example, in Fig. 2(b), as $o_{15} \xrightarrow{\text{new}} v_{l_{\text{main}}} \xrightarrow{\text{param}} \text{thisVector} \xrightarrow{\text{param}} \text{thisGet} \xrightarrow{\text{param}} \text{thisGet},$ we have $\text{thisGet} \xrightarrow{\text{alias thisGet}}$, Thus $o_{15} \xrightarrow{\text{new}} v_{l_{\text{main}}} \xrightarrow{\text{st}(\text{thisGet})} \xrightarrow{\text{thisGet}}$, indicating that $o_{15}$ flows to $\text{thisGet}$.

To allow the alias relation in the language, flowsTo is introduced as the inverse of the flowsTo relation. For a flowsTo-path $p$, its corresponding flowsTo-path $\overline{p}$ can be obtained using
flowsTo
new
flowsTo
n2
o
not[count++ = e;} // W t.arr
for context-sensitivity) for capturing realisable paths is:
the unrealisable paths in a program [18]. The CFL
matched, also in terms of balanced parentheses, by ruling out
either
i
ysis requires exiting from that method back to call site
or
method via
param
y
its inverse edge is
inverse edges, and vice versa. For each edge \( x \xleftarrow{\text{flowsTo}} y \) in \( p \),
it inverse edge is \( y \xleftarrow{p} x \). Therefore, \( o \) \text{flowsTo} \( x \) iff
\( x \) \text{flowsTo} \( o \), indicating that \( \text{flowsTo} \) actually represents the
points-to relation. To find the points-to set of a variable, we
use the CFL given in (2) with \( \text{flowsTo} \) as the start symbol.

2) Context-sensitivity: When context-sensitivity is consid-
ered, \param\( i \) and \ret\( i \) are treated as \assign\( i \) as before in \( L_{FS} \).
However, \assign\( 1 \) and \assign\( 6 \) are now distinguished.

A context-sensitive CFL requires \param\( i \) and \ret\( i \) to be
matched, also in terms of balanced parentheses, by ruling out
the unrealisable paths in a program [18]. The CFL \( R_{CS} \) (CS
for context-sensitivity) for capturing realisable paths is:

\[
\begin{align*}
\c & \rightarrow \text{entry}_i \c \ \text{exit}_i \mid \c \c \mid \epsilon \\
\text{entry}_i & \rightarrow \param_i \mid \ret_i \\
\text{exit}_i & \rightarrow \ret_i \mid \param_i
\end{align*}
\]

(3)

When traversing along a \text{flowsTo} path, after entering a
method via \param\( i \), from call site \( i \), a context-sensitive analysis
requires exiting from that method back to call site \( i \), via either \ret\( i \), to continue its traversal along the same \text{flowsTo} path or
\param\( i \) to start a new search for a \text{flowsTo} path. Traversing
along a \text{flowsTo} path is similar except that the direction of
traversal is reversed. Consider Fig. 2(b). \( s_{\text{main}} \) is found to
point to \( o_{18} \) as \( o_{18} \) reaches \( s_{\text{main}} \), via \( \param_i \) matches \( s_{\text{main}} \) and then \( \param_{i8} \) and \ret_{18}. However, \( s_{\text{main}} \) does not point to \( o_{20} \) since \( o_{20} \)
cannot reach \( s_{\text{main}} \), along a realisable path.

Let \( L_{PT} \) (PT for points-to) be the CFL for computing the
points-to information of a variable field- and context-
sensitively. Then \( L_{PT} \) is defined in terms of (2) and (3): \( L_{PT} = L_{FS} \cap R_{CS} \), where \text{flowsTo} is the start symbol.

3) Algorithm: With CFL-reachability, a query that requests
for a variable’s points-to information can be answered on-
demand, according to Algorithm 1. This algorithm makes use
of three variables: (1) \( E \) represents the edge set of the PAG
of the program, (2) \( B \) is the (global) budget defined as the
maximum number of steps that can be traversed by any query,
with each node traversal being counted as one step [19], and
(3) \( steps \) is query-local, representing the number of steps that
has been traversed so far by a particular query.

Given a query \((l, c)\), where \( l \) is a local variable and \( c \) is

```java
1 class Vector { 
2 Object[] elems; 
3 int count;
4 Vector() { 
5 count = 0;
6 t = new Object[MAXSIZE];
7 this elems = t; 
8 void add(Object e) 
9 t[count++] = e; } // W t.arr
10 }
11 static void main(String[] args) {
12 Vector v1 = new Vector();
13 String n1 = new String("N1");
14 Object s1 = v1.get(0);
15 Vector v2 = new Vector(); 
16 Integer(1);
17 t.add(n1); 
18 Object s2 = v2.get(0); 
19 }
20 }
```

(a) Java code

Fig. 2: A Java example and its PAG.

Algorithm 1 CFL-reachability-based pointer analysis,
where \text{POINTSTo} computes \text{flowsTo} and \text{flowsTo} is
analogous to its inverse \text{POINTSTo} and thus omitted.

```
Global E; Const B; QueryLocal steps; II initially 0
Procedure POINTSTo(l, c) begin
pts ← 0;
W ← \{c \in c;\}
while W \neq \emptyset do
\begin{align*}
&\langle x, c \rangle \xleftarrow{W.pop();}
&\text{steps} \xleftarrow{\text{steps} + 1};
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{if} \langle x, c \rangle \xleftarrow{B \text{outOfBudget}(0);}
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{if} \langle x, c \rangle \xleftarrow{\text{outOfBudget}(0);}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>);
\end{align*}
return pts;
```

Procedure REACHABLENODES(x, c) begin
rch ← \emptyset;
\begin{align*}
&\langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{if} \langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{alias} \leftarrow \emptyset;
&\text{W.push}(<y, c>); \\
&\text{if} \langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>); \\
&\text{if} \langle x, c \rangle \xleftarrow{\text{W.pop();}}
&\text{W.push}(<y, c>);
\end{align*}
return rch;

Procedure outOfBudget(BDG) begin
```
```
```
a context. POINTSTo computes the points-to set of l under c. It traverses the PAG with a work list W maintained for variables to be explored. pts is initialised with an empty set and W with <l, c> (lines 1 - 2). By default, steps for this query is initialised as 0. Each variable x with its context c, i.e., <x, c> obtained from W is processed as follows: steps is updated, triggering a call to OUTOFBUDGET if the remaining budget is 0 (lines 5 - 6), and the incoming edges of x are traversed according to (2) and (3) (lines 7 - 15).

Field-sensitivity is handled by REACHABLENODES(x, c), which searches for the reachable variables y to x in context c, due to heap accesses by matching the load (x = p.f) with every store (q.f = y), where p and q are aliases (lines 17 – 25). Both POINTSTo and FLOWSTo are called (recursively) to ensure that p and q are aliased base variables.

To handle context-sensitivity, the analysis stays in the same context c for assignl (line 8), clears c for assign2 as global variables are treated context-insensitively (line 9), matches the context (c.top() = i) for parami but allows for partially balanced parentheses when c = Ø since a realizable path may not start and end in the same method (lines 12 – 14), and pushes call site i into context c for reti (line 15).

III. METHODOLOGY

CFL-reachability-based pointer analysis is driven by queries issued by application clients. There are two main approaches to dividing work among threads, based on different levels of parallelism available: intra-query and inter-query.

To exploit intra-query parallelism, we need to partition and distribute the work performed in computing the points-to set of a single query among different threads. Such parallelism is irregular and hard to achieve with the right granularity. In addition, considerable synchronisation overhead that may be incurred would likely offset the performance benefit achieved.

To exploit inter-query parallelism, we assign different queries to different threads, harnessing modern multicore processors. This makes it possible to obtain parallelism without incurring synchronisation overhead unduly. In addition, some clients may issue queries in batch mode for a program. For example, the points-to information may be requested for all variables in a method, a class, a package or even the entire program. This provides a further optimisation opportunity. The focus of this work is on exploiting inter-query parallelism.

A. A Naïve Parallelisation Strategy

A naïve approach to exploiting inter-query parallelism is to maintain a lock-protected shared work list for queries and let each thread fetch queries (to process) from the work list until the work list is empty. While achieving some good speedups (over the sequential setting), this naïve strategy is inefficient due to a large number of redundant graph traversals made. We propose two schemes to reduce such redundancies. Section III-B describes our data sharing scheme, while Section III-C explains our query scheduling scheme.

III. METHODOLOGY

CFL-reachability-based pointer analysis is driven by queries issued by application clients. There are two main approaches to dividing work among threads, based on different levels of parallelism available: intra-query and inter-query.

To exploit intra-query parallelism, we need to partition and distribute the work performed in computing the points-to set of a single query among different threads. Such parallelism is irregular and hard to achieve with the right granularity. In addition, considerable synchronisation overhead that may be incurred would likely offset the performance benefit achieved.

To exploit inter-query parallelism, we assign different queries to different threads, harnessing modern multicore processors. This makes it possible to obtain parallelism without incurring synchronisation overhead unduly. In addition, some clients may issue queries in batch mode for a program. For example, the points-to information may be requested for all variables in a method, a class, a package or even the entire program. This provides a further optimisation opportunity. The focus of this work is on exploiting inter-query parallelism.

A. A Naïve Parallelisation Strategy

A naïve approach to exploiting inter-query parallelism is to maintain a lock-protected shared work list for queries and let each thread fetch queries (to process) from the work list until the work list is empty. While achieving some good speedups (over the sequential setting), this naïve strategy is inefficient due to a large number of redundant graph traversals made. We propose two schemes to reduce such redundancies. Section III-B describes our data sharing scheme, while Section III-C explains our query scheduling scheme.

Fig. 3: Adding jmp edges by graph rewriting, for a single iteration of the loop in line 18 of REACHABLENODES(x, c). In (a), x jmp yk is introduced for each (yk, ck) added to rch in line 24 of REACHABLENODES(x, c) when p and qk are aliases. In (b), a special x jmp O edge is introduced.

B. Data Sharing

Given a program, we are motivated to add edges to its PAG to serve as shortcuts for some paths traversed in a query so that subsequent queries may take the shortcuts instead of re-traversing their associated paths (redundantly). The challenge here is to perform data sharing context- and field-sensitively. Section III-B1 formulates data sharing in terms of graph rewriting. Section III-B2 gives an algorithm for realising data sharing in the CFL-reachability framework.

1) Data Sharing by Graph Rewriting: We choose to share paths involving heap accesses, which tend to be long (time-consuming to traverse) and common (repeatedly traversed across the queries). As illustrated in Fig. 3, we do so by avoiding making redundant alias tests in REACHABLENODES(x, c). For its loop at line 18, each iteration starts with a load x = p.f and then examines all the N matching stores q1.f = y1, . . . , qN.f = yN at line 19. For each qk.f = yk accessed in context ck such that qk is an alias of p, (yk, ck) is inserted into rch, meaning that (x, c) is reachable from (yk, ck) (lines 20 - 24). Note that during this process, mutually recursive calls to POINTSTo(), FLOWSTo() and REACHABLENODES() for discovering other aliases are often made.

There are two cases due to the budget constraint. Fig. 3(a) illustrates the case when an iteration of line 18 is completely analysed in s steps starting from (x, c) within the pre-set budget. A jmp edge, x jmp yk, is added for each qk that is an alias of p. Instead of rediscovering the path from
flowsTo flowsTo

rewriting problem by adding a special node added and if its remaining budget is smaller than the language as the original CFL if all

PAG of a program are ignored, since the CFL given earlier in (2) is modified to:

By definition of jmp, this modified CFL generates the same language as the original CFL if all jmp edges of the type illustrated in Fig. 3(b) (for handling OUTOFBUDGET) in the PAG of a program are ignored, since the jmp edges of the other type illustrated in Fig. 3(a) serve as shortcuts only. Two types of jmp edges are exploited in our parallel implementation to accelerate its performance as described below.

2) Algorithm: With data sharing, REACHABLENODES(x, c) in Algorithm 1 is revised as shown in Algorithm 2. There are three cases, the original one plus the two shown in Fig. 3:

- In the if branch (line 2) for handling the scenario depicted in Fig. 3(b), the analysis makes an early termination by calling REACHABLENODES() if its remaining budget at (x, c), B - steps, is smaller than s. Otherwise, the analysis moves to execute the second else.
- In the first else branch for handling the scenario in Fig. 3(a), the analysis takes the shortcuts identified by the jmp(s) edges instead of re-traversing its associated paths. The same precision is maintained even if we do not check the remaining budget B - steps against s, since the source node of a jmp edge is a variable (not an object). When this variable is explored later, the remaining budget will be checked in line 6 of Algorithm 1 or line 3 of Algorithm 2.
- In the second else branch, we proceed as in REACHABLENODES(x, c) given in Algorithm 1 except that we will need to add the jmp edge(s) as illustrated in

Algorithm 2 REACHABLENODES with data sharing.

Global E; Const B; QueryLocal steps, S;

Procedures REACHABLENODES(x, c)

begin

rch ← ϕ;

if ∃ x × jmp(s) →<c, c'> E ∈ O then

  if B - steps < s then OUTOFBUDGET(s);

else ∃ x × jmp(s) →<c, c'> y ∈ E then

  steps ← steps + s;

  foreach x × jmp(s) →<c, c'> y ∈ E do

    rch ← rch ∪ (y, c');

  return rch;

else

  s' = steps;

  S ← S ∪ (x, c, s');

  foreach x × p ∈ E do

    foreach q × y ∈ E do

      alias ← ϕ;

      foreach o, c' ∈ POINTSTO(p, c) do

        if o × alias ∈ alias ∪ FLOWSTo(o, c');

        else if q' = q then

          rch ← rch ∪ (y, c'');

          E ← E ∪ (x × jmp(steps - s') →<c, c''>);

      S ← S \ (x, c, s');

      return rch;

end

Procedures OUTOFBUDGET(BDG)

begin

foreach x, c, s ∈ S do

  E ← E ∪ (x × jmp(min(B, BDG + steps - s)) →<c, c>);

end;

exit();

either Fig. 3(a) (line 20) or Fig. 3(b) (line 24).

OUTOFBUDGET(BDG) is called from line 6 (by passing 0) in Algorithm 1 or line 3 in Algorithm 2 (by passing s). In both cases, let n be the node visited before the call. With a remaining budget no larger than BDG on encountering n, the analysis will surely run out of budget eventually. For each (x, c, s) ∈ S, the analysis first reaches (x, c) and then n in steps - s steps. Thus, x × jmp(min(B, BDG + steps - s)) →<c, c> is added.

C. Query Scheduling

The order in which queries are processed affects the number of early terminations made, due to B - steps < s tested in line 3 of Algorithm 2, where s appears in a jmp(s) edge that was added in an earlier query and steps is the number of steps already consumed by the current query. In general, if we handle a variable y before those variables x such that x is reachable from y, then more early terminations may result.
To increase early terminations, we organise queries (available in batch mode) in groups and assign a group of queries rather than a single query to a thread at a time to reduce synchronisation overhead on the shared work list for queries. We discuss below how the queries in a group and the groups themselves are scheduled. Section III-C1 describes how to group queries. Section III-C2 discusses how to order queries. Section III-C3 gives an illustrating example.

1) Grouping Queries: A group contains all possible variables such that every member is connected with at least another member in the PAG of the program via the following relation:

\[
direct \rightarrow (\text{assign}^1 | \text{assign}^2 | \text{param}^1 | \text{ret}^1)^* \tag{5}
\]

Both \(l_1 \leftarrow \text{id}(f)\) and \(l_1 \leftarrow \text{ut}(f)\) edges are not included since there is no reachability between \(l_1\) and \(l_2\).

2) Ordering Queries: For the variables in the same group, we use their so-called connection distances (CDs) to determine their issuing order. The CD of a variable in a group is defined as the length of the longest path that contains the variable in the group (modulo recursion). For the variables in a group, the shorter their CDs are, the earlier they are processed.

For different groups, we use their so-called dependence depths (DDs) to determine their scheduling order. For example, computing \(\text{pointsTo}(x, c)\) for \(x\) in Algorithm 1 depends on the points-to set of the base variable \(p\) in load \(x = p.f\) (line 21). Preferably, \(p\) should be processed earlier than \(x\).

To quantify the DD of a group, we estimate the dependences between variables based on their (static) types. We define the level of a type \(t\) (with respect to its containment hierarchy) as:

\[
L(t) = \begin{cases} 
\max_{t_i \in F_T(t)} L(t_i) + 1 & \text{isRef}(t) \\
0 & \text{otherwise}
\end{cases}
\]

where \(F_T(t)\) enumerates the types of all instance fields of \(t\) (modulo recursion) and isRef\((t)\) is true if \(t\) is a reference type. The DD of a variable of type \(t\) is defined to be \(1/L(t)\). Note that the DD of a static variable is also approximated heuristically this way. The DD of a group of variables is defined as the smallest of the DDs of all variables in the group.

During the analysis, groups are issued (sorted) in increasing values of their DDs. Let \(M\) be the average size of these groups. To ensure load balance, groups larger than \(M\) are split and groups smaller than \(M\) are merged with their adjacent groups, so that each resulting group has roughly \(M\) variables.

3) An Example: In Fig. 5, we focus on its three variables \(x, y,\) and \(z\), which are assumed to all run out of budget \(B\). According to (5), as shown in Fig. 5(a), \(x\) and \(y\) (together with \(w\)) are in one group and \(z\) (together with \(p\)) is in another group. The CDs of \(x, y,\) and \(z\) are 100, 200 and 300 steps respectively. As both \(x\) and \(y\) depend on \(z\), the latter group will be scheduled before the former group. As a result, our query scheduling scheme will likely cause \(x, y,\) and \(z\) to be processed sequentially according to \(O_3\) (in some thread interleaving) among the three orders, \(O_1, O_2\) and \(O_3\), listed in Fig. 5(b).

For \(O_1, y\) is processed first, which takes \(B\) steps (i.e., the maximum budget allowed), with the two \(\text{jmp}\) edges added as shown, where \(s_z = B - 500\) and \(s_w = B - 200\). When \(x\) is processed next, neither shortcut will be taken, since \(x\) still has more budget remaining: \(B - 400 > B - 500\) at \(z\) and \(B - 100 > B - 200\) at \(w\). Similarly, the two shortcuts do not benefit \(z\) either. Thus, no early termination occurs.

For \(O_2, x\) is issued first, resulting in also the same two \(\text{jmp}\) edges added, except that \(s_z = B - 400\) and \(s_w = B - 100\). So when \(y\) is handled next, an early termination is made at \(w\), since its budget remaining at \(w\) is \(B - 200\) (<\(s_w = B - 100\)).

According to \(O_3\), the order that is mostly likely induced by our query scheduling scheme, \(z\) is processed first. Only the \(\text{jmp}(s_z)\) edge at \(z\) is added, where \(s_z = B\). When \(x\) is analysed next, \(z\) is reached in 400 steps. Taking \(\text{jmp}(s_z)\) (since \(B - 400 < s_z = B\)), an early termination is made. Meanwhile, the \(\text{jmp}(s_w)\) edge at \(w\) is added, where \(s_w = B\). Finally, \(y\) is issued, causing \(w\) to be visited in 200 steps. Taking \(\text{jmp}(s_w)\) (since \(B - 200 < s_w = B\)), another early termination is made. Of the three orders illustrated in Fig. 5(b), \(O_3\) is likely to cause more early terminations, resulting in fewer traversal steps.

### IV. Evaluation

We demonstrate that our parallel implementation of CFL-reachability-based pointer analysis achieves significant speedups over a state-of-the-art sequential implementation.

#### A. Implementation

The sequential one is coded in Java based on the publicly available source-code implementation of the CFL-reachability-based pointer analysis [18] in Soot 2.5.0 [24], with its non-refinement (general-purpose) configuration used. Note that the refinement-based configuration is not well-suited to certain clients such as null-pointer detection. Our parallel implementation given in Algorithms 1 and 2 are also coded in Java. In both cases, the per-query budget \(B\) is set as 75,000 steps, recursion cycles of the call graph are collapsed, and points-to cycles are eliminated as described in [18].
In our parallel implementation, we use a ConcurrentHashMap to manage jmp edges efficiently. We apply a simple optimisation to further reduce synchronisation incurred and thus achieve better speedups.

If we create jmp edges exhaustively for all the paths discovered in Algorithm 2, the overhead incurred by such operations as search, insertion and synchronisation on the map may outweigh the performance benefit obtained. As an optimisation, we will introduce the jmp(s) edges in Fig. 3(a) only when $s \geq \tau_F$ and the special jmp(s) edge in Fig. 3(b) only when $s \geq \tau_U$, where $\tau_F$ and $\tau_U$ are tunable parameters. In our experiments, we set $\tau_F = 100$ and $\tau_U = 10000$. Their performance impacts are evaluated in Section IV-D2.

For the case in Fig. 3(a), the set of all jmp edges is associated with the key $(x,c)$ when inserted into the map. So no two threads reaching $(x,c)$ simultaneously will insert this set of jmp edges twice into the map. For the case in Fig. 3(b), if one thread tries to insert $(x,c), x \xymap{m_p(s_1)}{<c,\epsilon>} \bigcup \mathcal{O}$ and another thread tries to insert $(x,c), x \xymap{m_p(s_2)}{<c,\epsilon>} \bigcup \mathcal{O}$ into the map, only one of the two will succeed. An attempt that selects the one with the large $s$ value (to increase early terminations) can be cost-ineffective due to the extra work performed.

### B. Experimental Setting

The multi-core system used in our experiments is equipped with two Intel Xeon E5-2650 CPUs with 62GB of RAM. Each CPU has 8 cores, which share a unified 20MB L3 cache. Each CPU core has a frequency of 2.00MHz, with its own L1 cache of 64KB and L2 cache of 256KB. The Java Virtual Machine used is the Java HotSpot 64-Bit Server VM (version 1.7.0_40), running on a 64-bit Ubuntu 12.04 operating system.

### C. Methodology

We evaluate the performance advantages of our parallel implementation over the sequential one by comparing the query-processing times taken in both cases. $\text{SEQ}CFL$ denotes the sequential implementation. In order to assess the effectiveness of our parallel implementation, we consider a number of variations. $\text{PARCFL}^t_m$ represents a particular parallel implementation, where $t$ stands for the number of threads used. Here, $m$ indicates one of the three parallelisation strategies used: (1) the naive solution described in Section III-A when $m = \text{naive}$, (2) our parallel solution with the data sharing scheme described in Section III-B enabled when $m = D$, and (3) the parallel solution (2) with the query scheduling scheme described in Section III-C also enabled when $m = DQ$.

Table I lists a set of 20 Java benchmarks used, consisting of all the 10 SPEC JVM98 benchmarks and 10 additional benchmarks from the DaCapo 2009 benchmark suite. For each benchmark, the analysed code includes both the application code and the related library code, with their class and method counts given in Columns 2 and 3, respectively. The node and edge counts in the original PAG of a benchmark are given in Columns 4 and 5, respectively. For each benchmark, the queries that request points-to information are issued for all the local variables in its application code, collected from Soot 2.5.0 as in prior work [17, [25]. Note that more queries are generated in some DaCapo benchmarks than some JVM98 benchmarks even though the DaCapo benchmarks have smaller PAGs. This is because the JVM98 benchmarks involve more library code. The remaining columns are explained below.

### D. Performance Results

We examine the performance benefits of our parallel pointer analysis and the causes for the speedups obtained.

1) Speedups: Fig. 6 shows the speedups of our parallel implementation over $\text{SEQ}CFL$ (as the baseline), where the analysis times of $\text{SEQ}CFL$ for all the benchmarks are given in Column 7 of Table I. Note that $\text{SEQ}CFL$ is 49% faster than the open-source sequential implementation of [18] in Soot 2.5.0, since we have simplified some of its heuristics and em-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Classes</th>
<th>#Methods</th>
<th>#Nodes</th>
<th>#Edges</th>
<th>#Queries</th>
<th>$T_{\text{Seq}}$ (secs)</th>
<th>$#$Jumps</th>
<th>$#$S ($\times 10^5$)</th>
<th>$R_S$</th>
<th>$S_g$</th>
<th>$#$ET</th>
<th>$R_{ET}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>avoria</td>
<td>7,541</td>
<td>53,154</td>
<td>225,797</td>
<td>429,551</td>
<td>1,101</td>
<td>2.88</td>
<td>428</td>
<td>4.14</td>
<td>25.76</td>
<td>16.7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>batik</td>
<td>7,574</td>
<td>54,591</td>
<td>229,755</td>
<td>429,808</td>
<td>1,328</td>
<td>3.72</td>
<td>1,210</td>
<td>4.21</td>
<td>8.42</td>
<td>4.6</td>
<td>5</td>
<td>1.00</td>
</tr>
<tr>
<td>fop</td>
<td>8,251</td>
<td>57,797</td>
<td>233,676</td>
<td>435,391</td>
<td>6,389</td>
<td>46.52</td>
<td>2,306</td>
<td>86.17</td>
<td>53.33</td>
<td>3.8</td>
<td>53</td>
<td>3.17</td>
</tr>
<tr>
<td>h2</td>
<td>3,381</td>
<td>32,691</td>
<td>115,249</td>
<td>204,516</td>
<td>44,901</td>
<td>9.39</td>
<td>22,094</td>
<td>91.38</td>
<td>12.39</td>
<td>10.3</td>
<td>18</td>
<td>1.17</td>
</tr>
<tr>
<td>luindex</td>
<td>3,160</td>
<td>28,791</td>
<td>108,827</td>
<td>191,126</td>
<td>22,415</td>
<td>23.28</td>
<td>62,457</td>
<td>60.93</td>
<td>8.72</td>
<td>8.2</td>
<td>113</td>
<td>0.71</td>
</tr>
<tr>
<td>lusearch</td>
<td>3,120</td>
<td>28,223</td>
<td>109,439</td>
<td>193,012</td>
<td>17,520</td>
<td>57.78</td>
<td>77,153</td>
<td>66.26</td>
<td>7.90</td>
<td>9.3</td>
<td>75</td>
<td>1.52</td>
</tr>
<tr>
<td>pmr</td>
<td>3,786</td>
<td>33,432</td>
<td>110,588</td>
<td>195,834</td>
<td>56,833</td>
<td>61.05</td>
<td>77,313</td>
<td>69.10</td>
<td>7.93</td>
<td>9.2</td>
<td>84</td>
<td>1.06</td>
</tr>
<tr>
<td>sunflow</td>
<td>6,066</td>
<td>56,673</td>
<td>233,459</td>
<td>447,002</td>
<td>21,339</td>
<td>55.56</td>
<td>20,046</td>
<td>49.04</td>
<td>5.57</td>
<td>7.4</td>
<td>24</td>
<td>2.38</td>
</tr>
<tr>
<td>tomcat</td>
<td>8,458</td>
<td>83,092</td>
<td>265,015</td>
<td>574,236</td>
<td>185,810</td>
<td>202.89</td>
<td>24,601</td>
<td>243.90</td>
<td>23.14</td>
<td>13.1</td>
<td>574</td>
<td>1.33</td>
</tr>
<tr>
<td>xalan</td>
<td>3,716</td>
<td>33,248</td>
<td>109,317</td>
<td>192,441</td>
<td>56,229</td>
<td>54.11</td>
<td>33,459</td>
<td>60.35</td>
<td>7.90</td>
<td>9.4</td>
<td>82</td>
<td>1.43</td>
</tr>
<tr>
<td>Average</td>
<td>5,486</td>
<td>50,972</td>
<td>198,518</td>
<td>383,592</td>
<td>30,624</td>
<td>62.19</td>
<td>22,023</td>
<td>97.62</td>
<td>28.6</td>
<td>10.9</td>
<td>114.0</td>
<td>1.35</td>
</tr>
</tbody>
</table>

Table I: Benchmark information and statistics.
employed different speedups. When the naive parallelisation strategy is used, PARCFL_{naive} (with one single thread) is as efficient as SEQCFL, since the locking overhead incurred for retrieving the queries from the shared work list is negligible. With 16 threads, PARCFL_{naive} attains an average speedup of 7.3X. When our data sharing scheme is used, PARCFL_{16} runs a lot faster, with the average speedup being pushed up further to 13.4X. When our query scheduling scheme is also enabled, PARCFL_{DQ}, which traverses significantly fewer steps than SEQCFL, has finally reached an average speedup of 16.2X. The superlinear speedups are achieved in some benchmarks due to the avoidance of redundant traversals (a form of caching) in all concurrent query-processing threads as analysed below.

2) Effectiveness of Data Sharing: Our data sharing scheme, which enables the traversal information obtained in a query to be shared by subsequent queries via graph rewriting, has succeeded in accelerating the analysis on top of the naive parallelisation strategy (PARCFL_{naive}) for all benchmarks.

To understand its effectiveness, some statistics are given in Columns 8 – 10 in Table I. For a benchmark, #Jumps denotes the number of jmp edges added to its PAG due to data sharing. #S represents the total number of steps traversed by SEQCFL (without data sharing) for all the queries issued from the benchmark, and $R_S$ is the ratio of the number of steps saved by the jmp edges for the benchmark over the number of steps traversed across the original edges (when data sharing is enabled). For the 20 benchmarks used, 22,023 jmp edges have been added on average per benchmark. The number of steps saved by these jmp edges is much larger than that of the original ones, by a factor of 28.6X on average. This implies that a large number of redundant traversals (#S $\times$ $R_S$) for a benchmark) have been eliminated. Thus, PARCFL_{16} exhibits substantial improvements over PARCFL_{naive}, with the superlinear speedups observed in _202_jess, _213_javac, _222_mpegaudio, batik, fop and tomcat.

The optimisation described in Section IV-A for adding jmp edges selectively to reduce synchronisation overhead is also useful for improving the performance. Fig. 7 reveals the histograms of added jmp edges with and without this optimisation. In the absence of such optimisation, many jmp edges representing relatively short paths are also added, causing PARCFL_{DQ} to drop from 16.2X to 12.4X on average.

3) Effectiveness of Query Scheduling: When query scheduling is also enabled, queries are grouped and reordered to increase early terminations made. PARCFL_{DQ} achieves superlinear speedups in two more benchmarks than PARCFL_{16}; avrora and sunflow. PARCFL_{DQ} is faster than PARCFL_{16} as the average speedup goes up from 13.4X to 16.2X.

To understand its effectiveness, some statistics are given in Columns 11 – 13 in Table I. For a benchmark, #ETs gives the average number of queries in a group, #ETs is the number of early terminations found without query scheduling, and $R_{ET}$ is the ratio of #ETs obtained with query scheduling over #ETs obtained without query scheduling. On average, our
query scheduling scheme leads to 35% more early terminations, resulting in more redundant traversals being eliminated.

4) Scalability: To see the scalability of our parallel implementation, Fig. 8 plots its speedups with a few thread counts over the baseline. \( \text{PARCFL}_{DQ}^{1} \) achieves an average speedup of 8.1X, due to data sharing and query scheduling. Our parallel solution scales well to 8 threads for most benchmarks. When moving from 8 to 16 threads, \( \text{PARCFL}_{DQ}^{16} \) suffers some performance drops over \( \text{PARCFL}_{DQ}^{8} \) in some benchmarks (with _209_db being the worst case at 31%). However \( \text{PARCFL}_{DQ}^{16} \) is still slightly faster than \( \text{PARCFL}_{DQ}^{8} \) on average.

5) Memory Usage: As garbage collection is enabled, it is difficult to monitor memory usage precisely. By avoiding redundant graph traversals, \( \text{PARCFL}_{DQ}^{1} \) reduces the memory usage by \( \text{SEQCFL} \) (the open-source sequential implementation [18]) by 35% (32%) in terms of the peak memory usage, despite the extra memory required for storing jmp edges. In the worst case attained at _tomcat (fop), \( \text{PARCFL}_{DQ}^{16} \) consumes 103% (118%) of the memory consumed by \( \text{SEQCFL} \) ([18]).

V. RELATED WORK

A. CFL-Reachability-Based Pointer Analysis

In the sequential setting, there is no shortage of optimisations on improving the performance of demand-driven CFL-reachability-based pointer analysis. To ensure quick response, queries are commonly processed under budget constraints [17], [18], [19], [26], [27]. In addition, refinement-based schemes [18], [19] can be effective for certain clients, e.g., type casting if field-sensitivity is gradually introduced. Summary-based schemes avoid redundant graph traversals by reusing the method-local points-to relations summarised statically [26] or on-demand [17], achieving up to 3X speedups. Must-not-alias information obtained during a pre-analysis can be exploited to yield an average speedup of 3X through reducing unnecessary alias-related computations [25]. Incremental techniques [6], [16], which are tailored for scenarios where code changes are small, take advantage of previously computed CFL-reachable paths to avoid unnecessary reanalysis.

Unlike these efforts on sequential CFL-reachability-based pointer analysis, this paper introduces the first parallel solution on multicore processors with significantly better speedups.

B. Parallel Pointer Analysis

In recent years, there have been a number of attempts on parallelising pointer analysis algorithms for analysing C or Java programs on multi-core CPUs and/or GPUs [3], [7], [8], [9], [10], [14], [20]. As compared in Table II, all these parallel solutions are different forms of Andersen’s pointer analysis [2] with varying precision considerations in terms of context-, flow- and field-sensitivity. Our parallel solution is the only one that can be performed on-demand based on CFL-reachability.

Méndez-Lojo et al. [8] introduced the first parallel implementation of Andersen’s pointer analysis for C programs. While being context- and flow-insensitive, their parallel analysis is field-sensitive, achieving a speedup of up to 3X on eight CPU cores. The same baseline sequential analysis was later parallelised on a Tesla C2070 GPU [7] (achieving an average speedup of 7X with 14 streaming multiprocessors) and a CPU-GPU system [20] (boosting the CPU-only parallel solution by 51% and the GPU-only parallel solution by 79% on average).

Edvinsson et al. [3] described a parallel implementation of Andersen’s analysis for Java programs, achieving a maximum speedup of 4.4X on eight CPU cores. Their analysis is field-insensitive but flow-sensitive (only partially since it does not perform strong updates). Recently, both field- and flow-sensitivity (with strong updates for singleton objects) are handled for C programs on multi-core CPUs [9] and GPUs [10]. The speedups are up to 2.6X on eight CPU cores and 7.8X (with precision loss) on a Tesla C2070 GPU, respectively.

Putta and Nasre [14] use replications of points-to sets to enable more constraints (i.e., more pointer-manipulating statements) to be processed in parallel. Their context-sensitive implementation of Andersen-style analysis delivers an average speedup of 3.4X on eight CPU cores.

This paper presents the first parallel implementation of demand-driven CFL-reachability-based pointer analysis, achieving an average speedup of 16.2X for a set of 20 Java
programs on 16 CPU cores. Based on a version of CFL-reachability-based pointer analysis for C [27], our parallel solution is expected to generalise to C programs as well.

C. Parallel Graph Algorithms

There are many parallel graph algorithms, including breadth-first search (BFS) [1], [4], [11], minimum-cost path [12] and flow analysis [13]. However, parallelising CFL-reachability-based pointer analysis poses different challenges. The presence of both context- and field-sensitivity that is enforced during graph traversals makes it hard to avoid redundant traversals efficiently, limiting the amount of parallelism exploited (especially within a single query). Exploiting inter-query parallelism, this work has demonstrated significant performance benefits that can be achieved on parallelising CFL-reachability-based pointer analysis on multi-core CPUs.

VI. CONCLUSION

This paper presents the first parallel implementation of CFL-reachability-based pointer analysis on multi-core CPUs. Despite the presence of redundant graph traversals, this demand-driven analysis is non-trivial to parallelise due to the dependences introduced by context- and field-sensitivity during graph traversals. We have succeeded in parallelising it by using (1) a data sharing scheme that enables the concurrent query-processing threads to avoid traversing earlier discovered paths via graph rewriting and (2) a query scheduling scheme that allows more redundancies to be eliminated based on the dependences statically estimated among the queries to be processed. For a set of 20 Java benchmarks evaluated, our parallel implementation significantly boosts the performance of a state-of-the-art sequential implementation with an average speedup of 16.2X on 16 CPU cores.

ACKNOWLEDGMENT

This work is supported by Australian Research Grants, DP110104628 and DP130101970.

REFERENCES


<table>
<thead>
<tr>
<th>Analysis</th>
<th>Algorithm</th>
<th>On-demand</th>
<th>Sensitivity (Precision)</th>
<th>Applications</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Context</td>
<td>Field</td>
<td>Flow</td>
</tr>
</tbody>
</table>

* Partial flow-sensitivity without performing strong updates

TABLE II: Comparing different parallel pointer analysis.