Liveness-Based Pointer Analysis

Uday P. Khedker[1], Alan Mycroft[2], and Prashant Singh Rawat[1]

[1] Indian Institute of Technology Bombay
{uday,prashantr}@cse.iitb.ac.in
[2] University of Cambridge
Alan.Mycroft@cl.cam.ac.uk

Abstract. Precise flow- and context-sensitive pointer analysis (FCPA) is generally considered prohibitively expensive for large programs; most tools relax one or both of the requirements for scalability. We argue that precise FCPA has been over-harshly judged—the vast majority of points-to pairs calculated by existing algorithms are never used by any client analysis or transformation because they involve dead variables. We therefore formulate a FCPA in terms of a joint points-to and liveness analysis which we call L-FCPA. We implemented a naive L-FCPA in GCC-4.6.0 using linked lists. Evaluation on SPEC2006 showed significant increase in the precision of points-to pairs compared to GCC's analysis. Interestingly, our naive implementation turned out to be faster than GCC's analysis for all programs under 30kLoC. Further, L-FCPA showed that fewer than 4% of basic blocks had more than 8 points-to pairs. We conclude that the usable points-to information and the required context information is small and sparse and argue that approximations (e.g. weakening flow or context sensitivity) are not only undesirable but also unnecessary for performance.

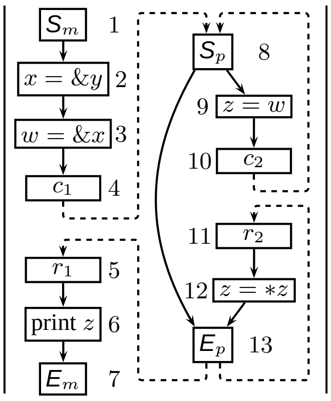
1 Introduction

Interprocedural data flow analysis extends an analysis across procedure boundaries to incorporate the effect of callers on callees and vice-versa. In order to compute *precise* information, such an analysis requires flow sensitivity (associating different information with distinct control flow points) and context sensitivity (computing different information for different calling contexts). The efficiency and scalability of such an analysis is a major concern and sacrificing precision for scalability is a common trend because the size of information could be large. Hence precise flow- and context-sensitive pointer analysis (FCPA) is considered prohibitively expensive and most methods employ heuristics that relax one or both of the requirements for efficiency.

We argue that the precision and efficiency in pointer analysis need not conflict and may actually be synergistic. We demonstrate this by formulating a liveness-based flow- and context-sensitive points-to analysis (referred to as L-FCPA): points-to information is computed only for the pointers that are live and the propagation of points-to information is restricted to live ranges of respective pointers. We use *strong liveness* to discover pointers that are directly used or are used in defining pointers that are strongly live. This includes the effect of dead code elimination and is more precise than simple liveness.

Fig. 1 provides a motivating example. Since *main* prints z, it is live at O_{12} (exit of node 12) and hence at I_{12} (entry of node 12). Thus w becomes live at O_9 and hence at

```
main()
{   x = &y;
    w = &x;
    p();
    print z;
}
p()
{   if (...)
    {   z = w;
        p();
        z = *z;
    }
}
```



Let I_n/O_n denote the entry/exit point of node n. Let (a, b) at a program point u denote that a points-to b at u. Then,

– z is live at O_9 which make w live at O_3. Hence we should compute (w, x) in node 3 and thereby (z, x) in node 9. This causes x to be live because of $*z$ in node 12. Hence we should compute (x, y) in node 2 and (z, y) in node 12.
– (w, x) and (x, y) should not be propagated to nodes 5, 6, 7 because w, x are not live in these nodes.

Fig. 1. A motivating example for L-FCPA and its supergraph representation. The solid and dashed edges represent intraprocedural and interprocedural control flow respectively.

O_3 resulting in the points-to pair (w, x) at O_3. This pair reaches I_9 giving the pair (z, x) at O_9. When this information reaches I_{12}, x becomes live. This liveness is propagated to O_2 giving the pair (x, y). Finally, we get the pair (z, y) at O_{12}. Figures 6 and 7 give fuller detail of the solution after formulating L-FCPA. Here we highlight the following:

– *Use of liveness*: points-to pairs are computed only when the pointers become live.
– *Sparse propagation*: pairs (x, y) and (w, x) are not propagated beyond the call to p in *main* because they are not live.
– *Flow sensitivity*: points-to information is different for different control flow points.
– *Context sensitivity*: (z, x) holds only for the inner call to p made from within p but not for the outer call to p made from the main procedure. Thus in spite of z being live at I_6, (z, x) is not propagated to I_6 but (z, y) is.

We achieve this using a data flow framework (Section 3) that employs an interdependent formulation for discovering strongly live pointer variables and their pointees. We compute must-points-to information from may-points-to information without fixed-point computation. Section 4 uses value-based termination of call strings for precise interprocedural analysis without having to compute a prohibitively large number of call strings. Section 5 discusses how heap locations, stack locations, and records are handled. After Section 6 (related work), Section 7 details experimental results which suggest that the traditional FCPA is non-scalable because it computes and stores (a) an order of magnitude more points-to pairs than can ever be used by a client analysis (e.g. pairs for dead pointers), and (b) a prohibitively large number of redundant contexts.

2 Background

A procedure p is represented by a control-flow graph (CFG). It has a unique entry node S_p with no predecessor and a unique exit node E_p with no successor; every node n is reachable from S_p, and E_p is reachable from every n. At the interprocedural level, a

Forward Analysis (Out_n depends on In_n)	Backward Analysis (In_n depends on Out_n)
$\mathit{In}_n = \begin{cases} \mathit{BI} & n = \mathsf{S}_p \\ \displaystyle\bigsqcap_{m \in \mathit{pred}(n)} \mathit{Out}_m & \text{otherwise} \end{cases}$	$\mathit{In}_n = f_n(\mathit{Out}_n)$
$\mathit{Out}_n = f_n(\mathit{In}_n)$	$\mathit{Out}_n = \begin{cases} \mathit{BI} & n = \mathsf{E}_p \\ \displaystyle\bigsqcap_{m \in \mathit{succ}(n)} \mathit{In}_m & \text{otherwise} \end{cases}$

Fig. 2. Typical data flow equations for some procedure p.

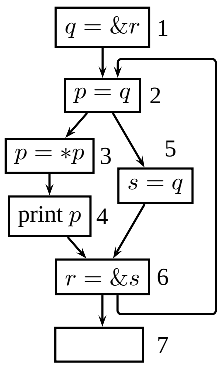
program is represented by a *supergraph* (e.g. in Fig. 1) which connects the CFGs by *interprocedural edges*. A call to procedure p at call site i is split into a *call node* c_i and a *return node* r_i with a call edge $c_i \to \mathsf{S}_p$ and a return edge $\mathsf{E}_p \to r_i$.

Formulating Data Flow Analysis. Data flow variables In_n and Out_n associate data flow information with CFG node n (respectively for its entry point I_n and exit point O_n); they must satisfy data flow equations (Fig. 2) involving node transfer functions f_n. Data flow values are taken from a meet-semilattice (meet represents confluence and the initial data flow value is $\top$). The *boundary information* BI represents the data flow information at I_{S_p} for forward analysis and O_{E_p} for backward analysis. Its value is governed by the semantics of the information being discovered. Interprocedural analysis eliminates the need for a fixed BI (except for arguments to the *main* procedure) by computing it from the calling contexts during the analysis. *Flow-insensitive* approaches disregard intraprocedural control flow for efficiency; they effectively treat the flow-equations as inequations ($\sqsubseteq$) and constrain all the In_n to be equal (and similarly all the Out_n). *Flow-sensitive* analyses honour control flow and keep the data flow information separate for each program point. *Iterative methods* solve data flow equations by repeatedly refining the values at each program point n starting from a conservative initialisation of $\top$; there are various strategies for this including *round robin* sweeps and *work list* methods.

The most precise data flow information at the intraprocedural level is the *Meet over Paths* (MoP) solution [1,2]. However, in general, an algorithm can at best compute the *Maximum Fixed Point* (MFP) solution [1,2]; however this is possible only if it is flow-sensitive. For distributive frameworks, e.g. live-variable analysis, MFP and MoP coincide; for non-distributive frameworks such as points-to analysis, they may differ.

Pointer Analysis. Points-to relations are computed by identifying locations corresponding to the left- and right-hand sides of a pointer assignment and taking their cartesian product [3,4]. The points-to pairs of locations that are modified are removed. May-points-to information at n contains the points-to pairs that hold along some path reaching n whereas must-points-to information contains the pairs that hold along every path reaching n (hence a pointer can have at most one pointee) [4]. Fig. 3 exemplifies flow-sensitive points-to analysis. By contrast an inclusion-based (Andersen) flow-insensitive analysis [5] associates (p,r), (p,s), (q,r), (r,s), (s,r) with all program points while the weaker equality-based (Steensgaard) analysis [6] further adds (q,s).

Interprocedural Data Flow Analysis. A supergraph contains control flow paths which violate nestings of matching call return pairs (e.g. 1-2-3-4-8-13-11 for the supergraph in



Node	May analysis		Must analysis	
	In_n	Out_n	In_n	Out_n
1	$\emptyset$	(q,r)	$\emptyset$	(q,r)
2	$(p,r),(p,s),$ $(q,r),(r,s),(s,r)$	$(p,r),(q,r),$ $(r,s),(s,r)$	(q,r)	$(p,r),(q,r)$
3	$(p,r),(q,r),$ $(r,s),(s,r)$	$(p,s),(q,r),$ $(r,s),(s,r)$	$(p,r),$ (q,r)	(q,r)
4	$(p,s),(q,r),$ $(r,s),(s,r)$	$(p,s),(q,r),$ $(r,s),(s,r)$	$(p,s),$ (q,r)	$(p,s),(q,r)$
5	$(p,r),(q,r),$ $(r,s),(s,r)$	$(p,r),(q,r),$ $(r,s),(s,r)$	$(p,r),$ (q,r)	$(p,r),(q,r),$ (s,r)
6	$(p,r),(p,s),$ $(q,r),(r,s),(s,r)$	$(p,r),(p,s),$ $(q,r),(r,s),(s,r)$	$(q,r),$	$(q,r),(r,s)$
7	$(p,r),(p,s),$ $(q,r),(r,s),(s,r)$	$(p,r),(p,s),$ $(q,r),(r,s),(s,r)$	$(q,r),$ (r,s)	$(q,r),(r,s)$

Fig. 3. An example of flow-sensitive intraprocedural points-to analysis.

Fig. 1). Such paths correspond to infeasible contexts. An *interprocedurally valid path* is a feasible execution path containing a legal sequence of call and return edges.

A *context-sensitive* analysis retains sufficient information about calling contexts to distinguish the data flow information reaching a procedure along different call chains. This restricts the analysis to interprocedurally valid paths. A *context-insensitive* analysis does not distinguish between valid and invalid paths, effectively merging data flow information across calling contexts. Recursive procedures have potentially infinite contexts, yet context-sensitive analysis is decidable for data flow frameworks with finite lattices and it is sufficient to maintain a finite number of contexts for such frameworks. Since this number is almost always impractically large, most context-sensitive methods limit context sensitivity in some way.

At the interprocedural level, the most precise data flow information is the *Meet over Interprocedurally Valid Paths* (IMoP) and the *Maximum Fixed Point over Interprocedurally Valid Paths* (IMFP) [7–9]. For computing IMFP, an interprocedural method must be fully flow and context sensitive. Relaxing flow (context) sensitivity admits invalid intraprocedural (interprocedural) paths; since no path is excluded, the computed information is provably safe but could be imprecise. Some examples of fully flow- and context-sensitive methods are: the graph reachability method [8] and the more general functional and full call-strings methods [7]. We use a variant of the full call-strings method [10] and compute the IMFP giving the most precise computable solution for pointer analysis; the loss of precision due to non-distributivity is inevitable.

Call-Strings Method [1, 7, 10]. This is a flow- and context-sensitive approach that embeds context information in the data flow information and ensures the validity of interprocedural paths by maintaining a history of calls in terms of call strings. A *call string* at node n is a sequence $c_1c_2 \ldots c_k$ of *call sites* corresponding to unfinished calls at n and can be viewed as a snapshot of the call stack. Call-string construction is governed by interprocedural edges. Let σ be a call string reaching procedure p. For an intraprocedural edge $m \to n$ in p, σ reaches n. For a call edge $c_i \to S_q$ where c_i belongs to p, call string σc_i reaches S_q. For a return edge $E_p \rightarrow r_j$ where r_j belongs to a caller of p there are two cases: if $\sigma = \sigma' c_j$ then σ' reaches r_j; otherwise σ and its data flow value is not propagated to r_j. This ensures that data flow information is only propagated to appropriate call sites. In a backward analysis, the call string grows on traversing a return edge and shrinks on traversing a call edge. The interprocedural data flow information at node n is a function from call strings to data flow values. Merging ($\sqcap$) the data flow values associated with all call strings reaching n gives the overall data flow value at n.

The original full call-strings method [7] used a pre-calculated length resulting in an impractically large number of call strings. We use value-based termination of call-string construction [10]. For forward flow, call strings are partitioned at S_p based on equality of their data flow values, only one call string per partition is propagated, and all call strings of the partition are regenerated at E_p (and the other way round for backward flows). This constructs only the relevant call strings (i.e. call strings with distinct data flow values) reducing the number of call strings significantly. For finite data flow lattices, we require only a finite number of call strings even in the presence of recursion. Moreover, there is no loss of precision as all relevant call strings are constructed.

We briefly describe value-based termination of call strings for forward analysis. Let $df(\sigma, n)$ denote the data flow value for call string σ at the entry of node n. Let $df(\sigma_1, S_p) = df(\sigma_2, S_p) = v$. Since data flow values are propagated along the same set of paths from S_p to E_p, $df(\sigma_1, S_p) = df(\sigma_2, S_p) \Rightarrow df(\sigma_1, E_p) = df(\sigma_2, E_p)$. Thus, we can propagate only one of them (say $\langle\sigma_1, v\rangle$) through the body of p. Let it reach E_p as $\langle\sigma_1, v'\rangle$. Then we can regenerate $\langle\sigma_2, v'\rangle$ at E_p by using $df(\sigma_1, E_p)$ if we remember that σ_2 was represented by σ_1 at S_p.

Recursion creates *cyclic* call strings $\gamma\alpha^i$ where γ and α are non-overlapping call site sequences and α occurs i times. Since the lattice is finite and the flow functions are monotonic, some $k \geq 0$ must exist such that $df(\gamma\alpha^{k+m}, S_p) = df(\gamma\alpha^k, S_p)$ where m is the *periodicity*[3] of the flow function for α. Hence $\gamma\alpha^{k+m}$ is represented by $\gamma\alpha^k$. Since $df(\gamma\alpha^{k+i\cdot m}, S_p) = df(\gamma\alpha^k, S_p), i > 0$, call string $\gamma\alpha^{k+m}$ is constructed for representation but call strings $\gamma\alpha^{k+i\cdot m}, i > 1$ are not constructed. Let $df(\gamma\alpha^k, E_p)$ be v. Then we generate $\langle\gamma\alpha^{k+m}, v\rangle$ in Out_{E_p} which is propagated along the sequence of return nodes thereby removing one occurrence of α. Thus the call string reaches E_p as $\gamma\alpha^k$, once again to be regenerated as $\gamma\alpha^{k+m}$. This continues until the values change, effectively computing $df(\gamma\alpha^{k+i\cdot m}, E_p), i > 1$ without constructing the call strings.

3 Liveness-Based Pointer Analysis

We consider the four basic pointer assignment statements: $x = \&y$, $x = y$, $x = *y$, $*x = y$ using which other pointer assignments can be rewritten. We also assume a *use* x statement to model other uses of pointers (such as in conditions). Discussion of address-taken local variables and allocation (*new* or *malloc*) is deferred to Section 5.

Let $\boldsymbol{V}$ denote the set of variables (i.e. "named locations"). Some of these variables (those in $\boldsymbol{P} \subset \boldsymbol{V}$) can hold pointers to members of $\boldsymbol{V}$. Other members of $\boldsymbol{V}$ hold non-

[3] x is a periodic point of f if $f^m(x) = x$ and $f^i(x) \neq x, 0 < i < m$. If $m = 1$, x is a fixed point of f. See Fig. 9.12 on page 316 in [1] for a points-to analysis example where $m = 2$.

pointer values. These include variables of non-pointer type such as `int`. NULL is similarly best regarded as a member of $\boldsymbol{V} - \boldsymbol{P}$; finally a special value '?' in $\boldsymbol{V} - \boldsymbol{P}$ denotes an undefined location (again Section 5 discusses this further).

Points-to information is a set of pairs (x, y) where $x \in \boldsymbol{P}$ is the pointer of the pair and $y \in \boldsymbol{V}$ is a pointee of x and is also referred to as the pointee of the pair. The pair $(x, ?)$ being associated with program point n indicates that x may contain an invalid address along some potential execution path from S_p to n.

The data flow variables Lin_n and Lout_n give liveness information for statement n while Ain_n and Aout_n give may-points-to information. Must-points-to information, Uin_n and Uout_n, is calculated from may-points-to. Note that liveness propagates backwards (transfer functions map *out* to *in*) while points-to propagates forwards.

The lattice of liveness information is $\mathcal{L} = \langle \mathcal{P}(\boldsymbol{P}), \supseteq \rangle$ (we only track the data flow of pointer variables) and lattice of may-points-to information is $\mathcal{A} = \langle \mathcal{P}(\boldsymbol{P} \times \boldsymbol{V}), \supseteq \rangle$. The overall data flow lattice is the product $\mathcal{L} \times \mathcal{A}$ with partial order $\langle l_1, a_1 \rangle \sqsubseteq \langle l_2, a_2 \rangle \Leftrightarrow (l_1 \sqsubseteq l_2) \wedge (a_1 \sqsubseteq a_2) \Leftrightarrow (l_1 \supseteq l_2) \wedge (a_1 \supseteq a_2)$ and having $\top$ element $\langle \emptyset, \emptyset \rangle$ and $\bot$ element $\langle \boldsymbol{P}, \boldsymbol{P} \times \boldsymbol{V} \rangle$. We use standard algebraic operations on points-to relations: given relation $R \subseteq \boldsymbol{P} \times \boldsymbol{V}$ and $X \subseteq \boldsymbol{P}$, define relation *application* $R\ X = \{v \mid u \in X \wedge (u, v) \in R\}$ and relation *restriction* $R|_X = \{(u, v) \in R \mid u \in X\}$.

Data Flow Equations. Fig. 4 provides the data flow equations for liveness-based pointer analysis. They resemble the standard data flow equations of strong liveness analysis and pointer analyses [1] except that liveness and may-points-to analyses depend on each other (hence the combined data flow is bi-directional in a CFG) and must-points-to information is computed from may-points-to information.

Since we use the greatest fixpoint formulation, the initial value ($\top$ of the corresponding lattices) is $\emptyset$ for both liveness and may-points-to analyses. For liveness BI is $\emptyset$ and defines Lout_{E_p}; for points-to analysis, BI is $\mathit{Lin}_n \times \{?\}$ and defines Ain_{S_p}. This reflects that no pointer is live on exit or holds a valid address on entry to a procedure.

Extractor Functions. The flow functions occurring in Equations (3) and (5) use *extractor functions* Def_n, Kill_n, Ref_n and $\mathit{Pointee}_n$ which extract the relevant pointer variables for statement n from the incoming pointer information Ain_n. These extractor functions are inspired by similar functions in [3, 4].

Def_n gives the set of pointer variables which a statement may modify and $\mathit{Pointee}_n$ gives the set of pointer values which may be assigned. Thus the new may-points-to pairs generated for statement n are $\mathit{Def}_n \times \mathit{Pointee}_n$ (Equation 5). Ref_n computes the variables that become live in statement n. Condition $\mathit{Def}_n \cap \mathit{Lout}_n$ ensures that Ref_n computes strong liveness rather than simple liveness. As an exception to the general rule, x is considered live in statement $*x = y$ regardless of whether the pointees of x are live otherwise, the pointees of x would not be discovered. For example, given `{x=&a; y=3; *x=y; return;}`, (x, a) cannot be discovered unless x is marked live. Hence liveness of x cannot depend on whether the pointees of x are live. By contrast, statement $y = *x$ uses the liveness of y to determine the liveness of x.

Kill_n identifies pointer variables that are definitely modified by statement n. This information is used to kill both liveness and points-to information. For statement $*x = y$, Given relation $R \subseteq \mathbf{P} \times \mathbf{V}$ (either Ain_n or Aout_n) we first define an auxiliary extractor function

$$\mathit{Must}(R) = \bigcup_{x \in \mathbf{P}} \{x\} \times \begin{cases} \mathbf{V} & R\{x\} = \emptyset \vee R\{x\} = \{?\} \\ \{y\} & R\{x\} = \{y\} \wedge y \neq ? \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

Extractor functions for statement n ($\mathit{Def}_n, \mathit{Kill}_n, \mathit{Ref}_n \subseteq \mathbf{P}$; $\mathit{Pointee}_n \subseteq \mathbf{V}$)
Notation: we assume that $x, y \in \mathbf{P}$ and $a \in \mathbf{V}$. A abbreviates Ain_n.

Stmt.	Def_n	Kill_n	Ref_n		$\mathit{Pointee}_n$
			if $\mathit{Def}_n \cap \mathit{Lout}_n \neq \emptyset$	Otherwise	
use x	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$	$\emptyset$
$x = \&a$	$\{x\}$	$\{x\}$	$\emptyset$	$\emptyset$	$\{a\}$
$x = y$	$\{x\}$	$\{x\}$	$\{y\}$	$\emptyset$	$A\{y\}$
$x = *y$	$\{x\}$	$\{x\}$	$\{y\} \cup (A\{y\} \cap \mathbf{P})$	$\emptyset$	$A(A\{y\} \cap \mathbf{P})$
$*x = y$	$A\{x\} \cap \mathbf{P}$	$\mathit{Must}(A)\{x\} \cap \mathbf{P}$	$\{x, y\}$	$\{x\}$	$A\{y\}$
other	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Data Flow Values: $\quad \mathit{Lin}_n, \mathit{Lout}_n \subseteq \mathbf{P} \qquad \mathit{Ain}_n, \mathit{Aout}_n \subseteq \mathbf{P} \times \mathbf{V}$

$$\mathit{Lout}_n = \begin{cases} \emptyset & n \text{ is } E_p \\ \bigcup\limits_{s \in \mathit{succ}(n)} \mathit{Lin}_s & \text{otherwise} \end{cases} \tag{2}$$

$$\mathit{Lin}_n = (\mathit{Lout}_n - \mathit{Kill}_n) \cup \mathit{Ref}_n \tag{3}$$

$$\mathit{Ain}_n = \begin{cases} \mathit{Lin}_n \times \{?\} & n \text{ is } S_p \\ \left(\bigcup\limits_{p \in \mathit{pred}(n)} \mathit{Aout}_p \right) \Bigg|_{\mathit{Lin}_n} & \text{otherwise} \end{cases} \tag{4}$$

$$\mathit{Aout}_n = ((\mathit{Ain}_n - (\mathit{Kill}_n \times \mathbf{V})) \cup (\mathit{Def}_n \times \mathit{Pointee}_n))\,|_{\mathit{Lout}_n} \tag{5}$$

Fig. 4. Intraprocedural formulation of liveness-based pointer analysis.

Kill_n depends on Ain_n filtered using the function Must. When no points-to information for x is available, the statement $*x = y$ marks all pointers as killed; this theoretically reflects the need for Kill_n to be anti-monotonic and practically that unreachable or C-undefined code is analysed liberally. When the points-to information for x is non-empty, Must performs a *weak update* or a *strong update* according to the number of pointees[4]: when x has multiple pointees we employ weak update as we cannot be certain which one will be modified because x may point to different locations along different execution paths reaching n. By contrast, when x has a single pointee *other than* '?', it indicates that x points to the same location along all execution paths reaching n and a strong update can be performed. Having BI be $\mathit{Lin}_n \times \{?\}$ completes this: if there is a definition-free path from S_p to statement n, the pair $(x, ?)$ will reach n and so a pair (x, z) reaching n cannot be incorrectly treated as a must-points-to pair.

[4] Or whether x is a summary node (see Section 5). Here we ignore summary nodes.

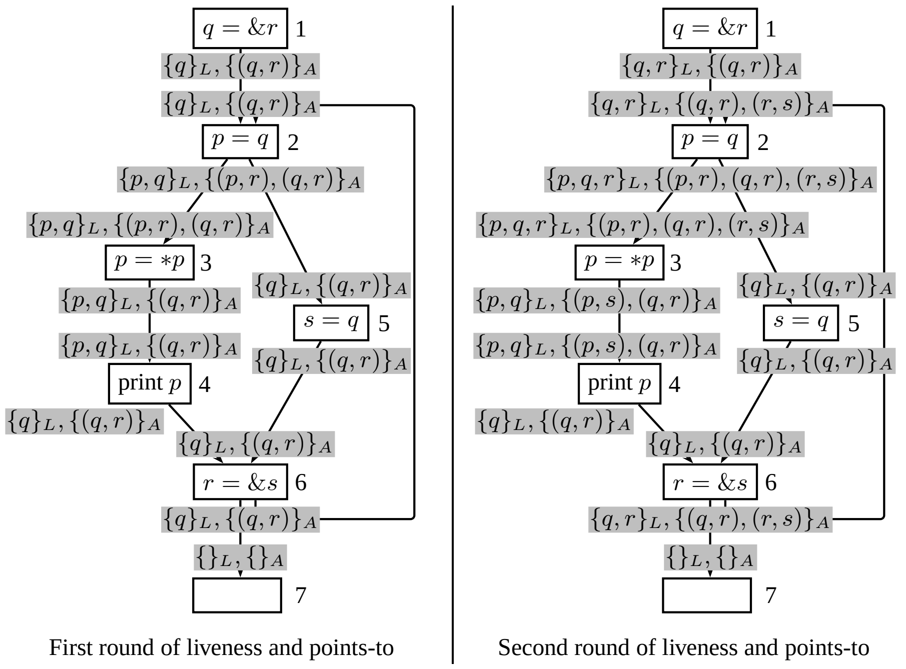


Fig. 5. Intraprocedural liveness-based points-to analysis of the program in Fig. 3. Shaded boxes show the liveness and points-to information suffixed by L and A respectively.

The above discussion of Kill_n and Must justifies why must-points-to analysis need not be performed as an interdependent fixed-point computation [4, 1]. Given pointer x, a single points-to pair (x, y) with $y \neq$? in Ain_n or Aout_n, guarantees that x points to y. Conversely multiple may-points-to pairs associated with x means that its must-points-to information is empty.[5] Hence must-points-to information can be extracted from may-points-to information by $\mathit{Uin}_n = \mathit{Must}(\mathit{Ain}_n)$ and $\mathit{Uout}_n = \mathit{Must}(\mathit{Aout}_n)$. Note that generally $\mathit{Uin}_n \subseteq \mathit{Ain}_n$ and $\mathit{Uout}_n \subseteq \mathit{Aout}_n$; the only exception would be for nodes that are not reached by the analysis because no pointer has been found to be live. For such nodes $\mathit{Uin}_n, \mathit{Uout}_n$ are $\boldsymbol{P} \times \boldsymbol{V}$ whereas $\mathit{Ain}_n, \mathit{Aout}_n$ are $\emptyset$; this matches previous frameworks and corresponds to Must being anti-monotonic (see above).

Motivating Example Revisited. Fig. 5 gives the result of liveness-based pointer analysis for our motivating example of Fig. 3. After the first round of liveness analysis followed by points-to analysis, we discover pair (p, r) in Ain_3. Thus r becomes live requiring a second round of liveness analysis. This then enables discovering the points-to pair (r, s) in node 6. A comparison with traditional may-points-to analysis (Fig. 3) shows that our analysis eliminates many redundant points-to pairs.

[5] This is more general than a similar concept for flow-sensitive kill in [11]. See Section 6.

Correctness. The following two claims are sufficient to establish soundness: (a) the flow functions in our formulation are monotonic (Theorem 1), and (b) for every use of a pointer, the points-to information defined by our formulation contains all addresses that it can hold at run time at a given program point (Theorem 2). Point (a) guarantees MFP computation at the intraprocedural level; at the interprocedural level, the full call-strings method ensures IMFP computation; point (b) guarantees that MFP (or IMFP) contains all usable pointer information.

Theorem 1. *The function Must is anti-monotonic hence the transfer functions Lin_n, Lout_n, Ain_n and Aout_n in Fig. 4 are monotonic.*

Theorem 2. *If $x \in \mathbf{P}$ holds the address of $z \in (\mathbf{V} - \{?\})$ along some execution path reaching node n, then $x \in \mathsf{Ref}_n \Rightarrow (x, z) \in \mathsf{Ain}_n$.*

4 Interprocedural Liveness-Based Pointer Analysis

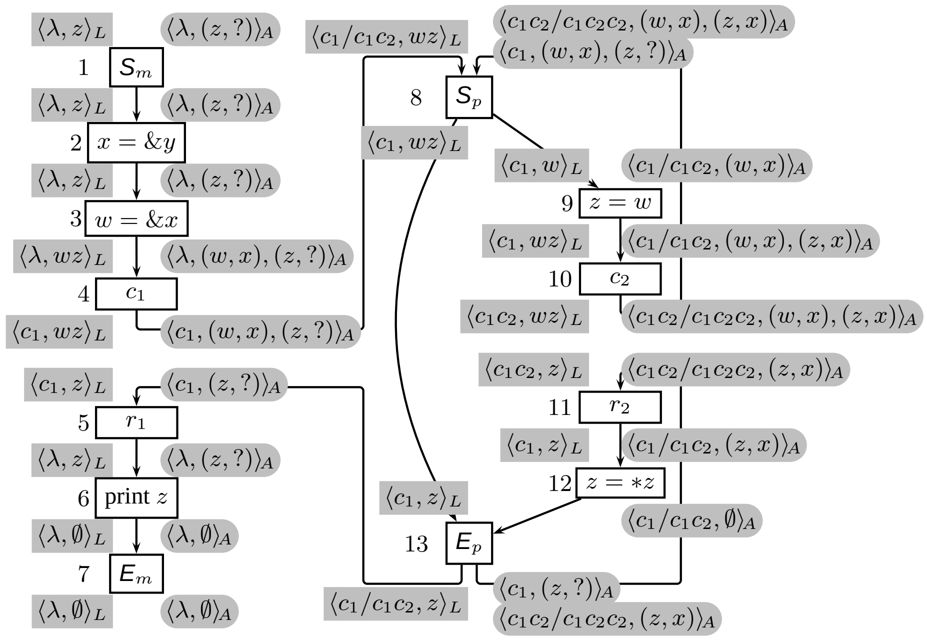
When our intraprocedural liveness-based points-to analysis is lifted to the interprocedural level using the call-strings method, $\mathsf{Lin}_n, \mathsf{Lout}_n$ and $\mathsf{Ain}_n, \mathsf{Aout}_n$ become functions of contexts written as sets of pairs $\langle \sigma, l \rangle, l \in \mathcal{L}$ and $\langle \sigma, a \rangle, a \in \mathcal{A}$ where σ is a call string reaching node n. Finally, the overall values of $\mathsf{Ain}_n, \mathsf{Aout}_n$ are computed by merging ($\sqcap$) the values along all call strings.

Matching Contexts for Liveness and Points-to Analysis. Since points-to information should be restricted to live ranges, it is propagated along the call strings constructed during liveness analysis. In the presence of recursion, we may need additional call strings for which liveness information may not yet be available. Such cases can be resolved by using the existing call strings as explained below. Let σ_a denote an acyclic call string and let $\sigma_c = \gamma\alpha^i$ be a cyclic call string (see Section 2). Then for liveness analysis:

– The partitioning information for every σ_a is available because either $\langle \sigma_a, x \rangle$ has reached node n in procedure p or σ_a has been represented by some other call string.
– Let $df(\gamma\alpha^i, n)$ differ for $0 \le i \le k$ but let $df(\gamma\alpha^k, n) = df(\gamma\alpha^{k+j}, n)$, $j > 0$ (the periodicity m for liveness analysis is 1). Then the partitioning information is available for only $\gamma\alpha^k$ and $\gamma\alpha^{k+1}$ because $\gamma\alpha^{k+j}$, $j > 1$ are not constructed.

Consider a call string σ' reaching node n during points-to analysis. If σ' is an acyclic call string then its partitioning information and hence its liveness information is available. If σ' is a cyclic call string $\gamma\alpha^i$, its liveness information may not be available if it has not been constructed for liveness. In such a situation, it is sufficient to locate the longest $\gamma\alpha^l$, $l < i$ among the call strings that have been created and use its liveness information. This effect is seen below in our motivating example.

Motivating Example Revisited. For brevity, let I_n and O_n denote the entry and exit of node n. In the first round of liveness (Fig. 6), z becomes live at I_6 as $\langle \lambda, z \rangle_L$, reaches $O_{13}, I_{13}, O_{12}, I_{12}, O_{11}$ as $\langle c_1, z \rangle_L$, becomes $\langle c_1 c_2, z \rangle_L$ at I_{11}, reaches O_{13} and



$\langle \lambda, z\rangle_L$
$\langle \lambda, (z,?)\rangle_A$
1 S_m
$\langle \lambda, z\rangle_L$
$\langle \lambda, (z,?)\rangle_A$
2 $x = \&y$
$\langle \lambda, z\rangle_L$
$\langle \lambda, (z,?)\rangle_A$
3 $w = \&x$
$\langle \lambda, wz\rangle_L$
$\langle \lambda, (w,x), (z,?)\rangle_A$
4 c_1
$\langle c_1, wz\rangle_L$
$\langle c_1, (w,x), (z,?)\rangle_A$
$\langle c_1, z\rangle_L$
$\langle c_1, (z,?)\rangle_A$
5 r_1
$\langle \lambda, z\rangle_L$
$\langle \lambda, (z,?)\rangle_A$
6 print z
$\langle \lambda, \emptyset\rangle_L$
$\langle \lambda, \emptyset\rangle_A$
7 E_m
$\langle \lambda, \emptyset\rangle_L$
$\langle \lambda, \emptyset\rangle_A$
$\langle c_1/c_1c_2, wz\rangle_L$
$\langle c_1c_2/c_1c_2c_2, (w,x), (z,x)\rangle_A$
$\langle c_1, (w,x), (z,?)\rangle_A$
8 S_p
$\langle c_1, wz\rangle_L$
$\langle c_1, w\rangle_L$
$\langle c_1/c_1c_2, (w,x)\rangle_A$
9 $z = w$
$\langle c_1, wz\rangle_L$
$\langle c_1/c_1c_2, (w,x), (z,x)\rangle_A$
10 c_2
$\langle c_1c_2, wz\rangle_L$
$\langle c_1c_2/c_1c_2c_2, (w,x), (z,x)\rangle_A$
$\langle c_1c_2, z\rangle_L$
$\langle c_1c_2/c_1c_2c_2, (z,x)\rangle_A$
11 r_2
$\langle c_1, z\rangle_L$
$\langle c_1/c_1c_2, (z,x)\rangle_A$
12 $z = *z$
$\langle c_1, z\rangle_L$
$\langle c_1/c_1c_2, \emptyset\rangle_A$
13 E_p
$\langle c_1/c_1c_2, z\rangle_L$
$\langle c_1, (z,?)\rangle_A$
$\langle c_1c_2/c_1c_2c_2, (z,x)\rangle_A$

Fig. 6. Liveness and points-to information (subscripted with L and A) after the first round of interprocedural analysis. For brevity, set of live variables are represented as strings and '$\{$' and '$\}$' are omitted. Multiple call strings with the same data flow value are separated by a '/'.

gets represented by $\langle c_1, z\rangle_L$. Hence $\langle c_1c_2, z\rangle_L$ is not propagated within the body of p. $\langle c_1c_2, z\rangle_L$ is regenerated at I_8, becomes $\langle c_1, z\rangle_L$ at I_{10}, becomes $\langle c_1, w\rangle_L$ at I_9. At O_8, it combines with $\langle c_1, z\rangle_L$ propagated from I_{13} and becomes $\langle c_1, w\ z\rangle_L$. Thus c_1c_2 is regenerated as $\langle c_1c_2, w\ z\rangle_L$ at I_8. $\langle c_1, w\ z\rangle_L$ reaches O_4 and becomes $\langle \lambda, w\ z\rangle_L$ at I_4.

In the first round of points-to analysis (Fig. 6), since z is live at I_1, $\mathit{BI} = \langle \lambda, (z, ?)\rangle_A$. $\langle \lambda, (w, x)\rangle_A$ is generated at O_3. Thus $\langle c_1, (w, x), (z, ?)\rangle_A$ reaches I_8. This becomes $\langle c_1, (w, x), (z, x)\rangle_A$ at O_9 and reaches as $\langle c_1c_2, (w, x), (z, x)\rangle_A$ at I_8. Since z is not live at I_9, $\langle c_1c_2, (w, x)\rangle_A$ is propagated to I_9. This causes $\langle c_1c_2c_2, (w, x), (z, x)\rangle_A$ to be generated at O_{10} which reaches I_9 and is represented by $\langle c_1c_2, (w, x), (z, x)\rangle_A$. This is then regenerated as $\langle c_1c_2c_2, (z, x)\rangle_A$ at O_{13} because only z is live at O_{13}. Note that we do not have the liveness information along $c_1c_2c_2$ but we know (from above) that it is identical to that along c_1c_2. We get $\langle c_1c_2, (z, x)\rangle_A$ and $\langle c_1, (z, x)\rangle_A$ at O_{11}. Since we have no points-to information for x, we get $\langle c_1c_2, \emptyset\rangle_A$ and $\langle c_1, \emptyset\rangle_A$ at O_{12}.

We leave it for the reader to verify that, in the second round (Fig. 7), x becomes live at I_{12} due to $z = *z$, reaches O_2 and causes $\langle \lambda, (x, y)\rangle_A$ to be generated. As a consequence, we get (z, y) at I_{12}. Note that (z, x) cannot reach I_6 along any interprocedurally valid path. The invocation graph method [3] which is generally considered the most precise flow- and context-sensitive method, *does* compute (z, x) at I_6. This shows that it is only partially context-sensitive. L-CFPA is more precise than [3] not only because of liveness but also because it is fully context-sensitive.

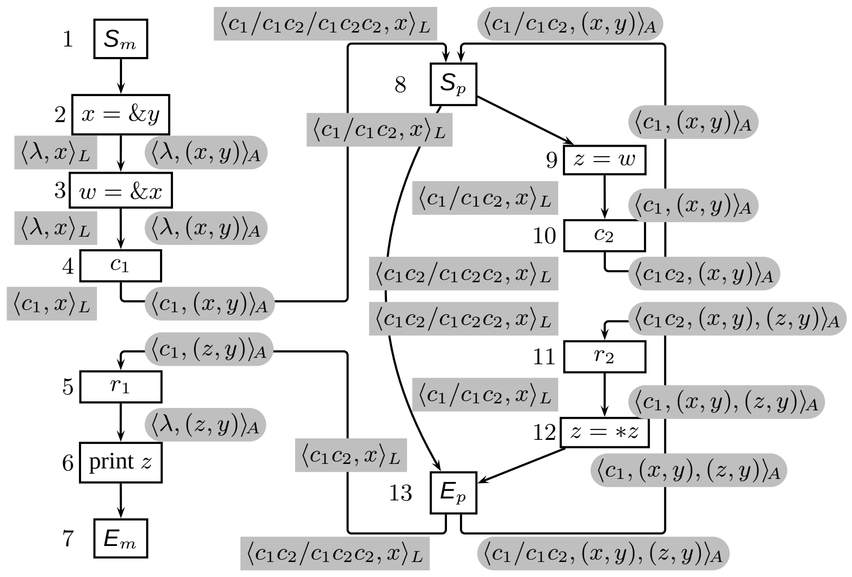

1 S_m
2 $x = \&y$
3 $w = \&x$
4 c_1
5 r_1
6 print z
7 E_m
8 S_p
9 $z = w$
10 c_2
11 r_2
12 $z = *z$
13 E_p
$\langle \lambda, x \rangle_L$
$\langle \lambda, (x, y) \rangle_A$
$\langle \lambda, x \rangle_L$
$\langle \lambda, (x, y) \rangle_A$
$\langle c_1, x \rangle_L$
$\langle c_1, (x, y) \rangle_A$
$\langle c_1, (z, y) \rangle_A$
$\langle \lambda, (z, y) \rangle_A$
$\langle c_1/c_1c_2/c_1c_2c_2, x \rangle_L$
$\langle c_1/c_1c_2, (x, y) \rangle_A$
$\langle c_1/c_1c_2, x \rangle_L$
$\langle c_1, (x, y) \rangle_A$
$\langle c_1/c_1c_2, x \rangle_L$
$\langle c_1, (x, y) \rangle_A$
$\langle c_1c_2/c_1c_2c_2, x \rangle_L$
$\langle c_1c_2, (x, y) \rangle_A$
$\langle c_1c_2/c_1c_2c_2, x \rangle_L$
$\langle c_1c_2, (x, y), (z, y) \rangle_A$
$\langle c_1/c_1c_2, x \rangle_L$
$\langle c_1, (x, y), (z, y) \rangle_A$
$\langle c_1c_2, x \rangle_L$
$\langle c_1, (x, y), (z, y) \rangle_A$
$\langle c_1c_2/c_1c_2c_2, x \rangle_L$
$\langle c_1/c_1c_2, (x, y), (z, y) \rangle_A$

Fig. 7. Second round of liveness and points-to analysis to compute dereferencing liveness and the resulting points-to information. Only the additional information is shown.

5 Heaps, Escaping Locals and Records

Each data location statically specified in a program is an abstract location and may correspond to multiple actual locations. It may be explicitly specified by taking the address of a variable or implicitly specified as the result of *new* or *malloc*. For interprocedural analysis, we categorise all abstract locations as shown in Fig. 8.

Define *interprocedural locations* as those abstract locations which are accessible in multiple contexts reaching a given program point or whose data flow values depend (via a dataflow equation) on another interprocedural location. These are the locations for which interprocedural data flow analysis is required. Global variables and heap locations are interprocedural locations. For pointer analysis, a local variable x becomes an interprocedural location if its address escapes the procedure containing it, or there is an assignment $x = y$ or $x = *z$ with y, z or one of z's pointees being an interprocedural location. Interprocedural locations for liveness analysis are similarly identified.

It is easy to handle different instances of a local variable which is not an interprocedural location (even if its address is taken). To see how other local variables are handled, consider a local variable x which becomes interprocedural from assignment $x = y$ or $x = *z$ as in the previous paragraph. Since call strings store context-sensitive data flow values of y and z, they also distinguish between instances of x whose data flow values may differ. Thus, call strings inherently support precise interprocedural analysis of global variables and locals (even interprocedural locals) whose addresses do not escape (the entry "No*" for the latter category in Fig. 8 indicates that interprocedural analysis is either not required or is automatically supported by call-strings method without any special treatment).

Issue	Global Variable	Local Variable		Heap allocation at a given source line
		Address escapes	Address does not escape	
How many instances can exist?	Single	Arbitrarily many	Arbitrarily many	Arbitrarily many
Can a given instance be accessed in multiple calling contexts?	Yes	Yes	No	Yes
Number of instances accessible at a given program point?	At most one	Arbitrarily many	At most one	Arbitrarily many
Is interprocedural data flow analysis required?	Yes	Yes	No*	Yes
Is a summary node required?	No	Yes	No	Yes

Fig. 8. Categorisation of data locations for interprocedural pointer analysis

Since the number of accessible instances of heap locations and locals whose addresses escape is not bounded,[6] we need to create summary nodes for them. It is difficult to distinguish between instances which are accessible in different contexts. Hence creating a summary node implies that the data flow values are stored context insensitively (but flow sensitively) by merging values of all instances accessible at a given program point. A consequence of this decision is that strong updates on these abstract locations are prohibited; this is easily engineered by Must returning $\emptyset$ for summary-node pointees which is consistent with the requirements of $\mathit{Uin}_n/\mathit{Uout}_n$ computation.

Recall that Equation 1 does not treat '?' as a summary node. This depends on the language-defined semantics of indirect writes via uninitialised pointers. In C (because the subsequent program behaviour is undefined) or Java (because of 'NullPointerException') it is safe to regard Must as returning all possible values when only '?' occurs. Alternatively, were the semantics to allow subsequent code to be executed in a defined manner, then '?' needs to be treated as a summary node so that Must returns $\emptyset$ and indirect writes kill nothing (in general this results in reduced optimisation possibilities).

Our implementation treats an array variable as a single scalar variable with weak update (no distinction is made between different index values). Stack-allocated structures are handled field-sensitively by using the offsets of fields. Heap-allocated structures are also handled field sensitively where possible. Function pointers are handled as in [3].

6 Related Work

The reported benefits of flow and context sensitivity for pointer analysis have been mixed in literature [12–15] and many methods relax them for efficiency [5, 6, 11, 16]. It has also been observed that an increase in precision could increase efficiency [17, 11]. Both these aspects have been studied without the benefit of liveness, partially explaining marginal results. Some methods lazily compute pointer information on demand [18–21]. By contrast, L-FCPA does not depend on a client analysis and proactively computes the entire usable pointer information. If there are many demands, repeated incremental computations could be rather inefficient [22]. Efficient encoding of information by using BDDs [23] has been an orthogonal approach of achieving efficiency. Although the

[6] Local variables whose addresses escape may belong to recursive procedures.

usable pointer information discovered by L-FCPA is small, recording it flow sensitively in a large program may benefit from BDDs.

The imprecision caused by flow insensitivity can be partially mitigated by using SSA representation which enables a flow-insensitive method to compute flow-sensitive information for local scalar variables. For pointers, the essential properties of SSA can only be guaranteed for top-level pointers whose address is not taken. Some improvements are enabled by Factored SSA [24] or Hashed SSA [25]. In the presence of global pointer variables or multiple indirections, the advantages of SSA are limited unless interleaved rounds of SSA construction and pointer analysis are performed [26, 27]. A recent method introduces flow-sensitive kill in an otherwise flow-insensitive method [11].

Full context sensitivity can be relaxed in many ways: (a) using a context-insensitive approach, (b) using a context-sensitive approach for non-recursive portions of a program but merging data flow information in the recursive portions (e.g. [3, 27–29]), or (c) using limited depth of contexts in both recursive and non-recursive portions (e.g. the k-limited call-strings method [7] or [23]). Most context-sensitive approaches that we are aware of belong to category (b). Our fully context-sensitive approach generalises partially context-sensitive approaches such as *object-sensitivity* [30, 12, 17] as follows. For an object x and its method *f*, a (virtual) call $x.f(e_1, \ldots, e_n)$ is viewed as the call $(x.f\_in\_vtab)(\&x, e_1, \ldots, e_n)$. Thus object identification reduces to capturing the flow of values which is inherently supported by full flow and context sensitivity.

We highlight some key ideas that have not been covered above. A memoisation-based functional approach enumerates partial transfer functions [28] whereas an alternative functional approach constructs full transfer functions hierarchically in terms of pointer indirection levels [27]. The invocation-graph-based approach unfolds a call graph in terms of call chains [3]. Finally, a radically different approach begins with flow- and context-insensitive information which is refined systematically to restrict it to flow- and context-sensitive information [29]. These approaches merge points-to information in recursive contexts (category (b) above). Fig. 9.6 (page 305) in [1] contains an example for which a method belonging to category (b) or (c) above cannot compute precise result—the pointer assignments in the recursion unwinding part undo the effect of the pointer assignments in the part that builds up recursion and the overall function is an identity function. When all recursive calls receive the same (merged) information, the undo effect on the pointer information cannot be captured.

Finally, many investigations tightly couple analysis specification and implementation; by contrast our formulation maintains a clean separation between the two and does not depend on intricate procedural algorithms or ad-hoc implementation for efficiency.

7 Implementation and Empirical Measurements

We implemented L-FCPA and FCPA in GCC 4.6.0 using the GCC's Link Time Optimisation (LTO) framework.[7] We executed them on various programs from SPEC CPU2006 and CPU2000 Integer Benchmarks on a machine with 16 GB RAM with 8 64-bit Intel i7-960 CPUs running at 3.20GHz. We compared the performance of three

[7] They can be downloaded from http://www.cse.iitb.ac.in/grc/index.php?page=lipta.

Program	kLoC	Call Sites	Time in milliseconds				Unique points-to pairs		
			L-FCPA		FCPA	GPTA	L-FCPA	FCPA	GPTA
			Liveness	Points-to					
lbm	0.9	33	0.55	0.52	1.9	5.2	12	507	1911
mcf	1.6	29	1.04	0.62	9.5	3.4	41	367	2159
libquantum	2.6	258	2.0	1.8	5.6	4.8	49	119	2701
bzip2	3.7	233	4.5	4.8	28.1	30.2	60	210	8.8×10^4
parser	7.7	1123	1.2×10^3	145.6	4.3×10^5	422.12	531	4196	1.9×10^4
sjeng	10.5	678	858.2	99.0	3.2×10^4	38.1	267	818	1.1×10^4
hmmer	20.6	1292	90.0	62.9	2.9×10^5	246.3	232	5805	1.9×10^6
h264ref	36.0	1992	2.2×10^5	2.0×10^5	?	4.3×10^3	1683	?	1.6×10^7

Table 1. Time and unique points-to pairs measurements. For h264ref, FCPA ran out of memory.

methods: L-FCPA, FCPA and GPTA (GCC's points-to analysis). Both L-FCPA and FCPA are flow and context sensitive and use call strings with value-based termination. L-FCPA uses liveness whereas FCPA does not. GPTA is flow and context insensitive but acquires partial flow sensitivity through SSA.

Since our main goal was to find out if liveness increases the precision of points-to information, both L-FCPA and FCPA are naive implementations that use linked lists and linear searches within them. Our measurements confirm this hypothesis beyond doubt, but we were surprised by the overall implementation performance because we had not designed for time/space efficiency or scalability. We were able to run naive L-FCPA on programs of around 30kLoC but not on the larger programs.

Table 1 presents the computation time and number of points-to pairs whereas Tables 2 and 3 present measurements of points-to information and context information respectively. To measure the sparseness of information, we created four buckets of the numbers of points-to pairs and call strings: 0, 1–4, 5–8 and 9 or more. We counted the number of basic blocks for each bucket of points-to information and the number of functions for each bucket of context information. Our data shows that:

– The usable pointer information is (a) rather sparse (64% of basic blocks have 0 points-to pairs), and (b) rather small (four programs have at most 8 points-to pairs and in other programs, 9+ points-to pairs reach fewer than 4% basic blocks). In contrast, GPTA computes an order-of-magnitude-larger number of points-to pairs at each basic block (see the last column in Table 1).
– The number of contexts required for computing the usable pointer information is (a) rather sparse (56% or more basic blocks have 0 call strings), and (b) rather small (six programs have at most 8 call strings; in other programs, 9+ call strings reach less than 3% basic blocks). Thus, contrary to the common apprehension, context information need not be exponential in practice. Value-based termination reduces the number of call strings dramatically [10] and the use of liveness enhances this effect further by restricting the computation of data flow values to the usable information.

Program	Total no. of BBs	No. and percentage of basic blocks (BBs) for points-to (pt) pair counts							
		0 pt pairs		1-4 pt pairs		5-8 pt pairs		9+ pt pairs	
		L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA
lbm	252	229 (90.9%)	61 (24.2%)	23 (9.1%)	82 (32.5%)	0	66 (26.2%)	0	43 (17.1%)
mcf	472	356 (75.4%)	160 (33.9%)	116 (24.6%)	2 (0.4%)	0	1 (0.2%)	0	309 (65.5%)
libquantum	1642	1520 (92.6%)	793 (48.3%)	119 (7.2%)	796 (48.5%)	3 (0.2%)	46 (2.8%)	0	7 (0.4%)
bzip2	2746	2624 (95.6%)	1085 (39.5%)	118 (4.3%)	12 (0.4%)	3 (0.1%)	12 (0.4%)	1 (0.0%)	1637 (59.6%)
	9+ pt pairs in L-FCPA: Tot 1, Min 12, Max 12, Mean 12.0, Median 12, Mode 12								
sjeng	6000	4571 (76.2%)	3239 (54.0%)	1208 (20.1%)	12 (0.2%)	221 (3.7%)	41 (0.7%)	0	2708 (45.1%)
hmmer	14418	13483 (93.5%)	8357 (58.0%)	896 (6.2%)	21 (0.1%)	24 (0.2%)	91 (0.6%)	15 (0.1%)	5949 (41.3%)
	9+ pt pairs in L-FCPA: Tot 6, Min 10, Max 16, Mean 13.3, Median 13, Mode 10								
parser	6875	4823 (70.2%)	1821 (26.5%)	1591 (23.1%)	25 (0.4%)	252 (3.7%)	154 (2.2%)	209 (3.0%)	4875 (70.9%)
	9+ pt pairs in L-FCPA: Tot 13, Min 9, Max 53, Mean 27.9, Median 18, Mode 9								
h264ref	21315	13729 (64.4%)	?	4760 (22.3%)	?	2035 (9.5%)	?	791 (3.7%)	?
	9+ pt pairs in L-FCPA: Tot 44, Min 9, Max 98, Mean 36.3, Median 31, Mode 9								

Table 2. Liveness restricts the analysis to usable pointer information which is small and sparse.

The significant increase in precision achieved by L-FCPA suggests that a pointer analysis need not compute exponentially large information. We saw this sub-exponential trend in programs of up to around 30kLoC and anticipate it might hold for larger programs too—because although reachable pointer information may increase significantly, usable information need not accumulate and may remain distributed in the program.

A comparison with GPTA shows that using liveness reduces the execution time too—L-FCPA outperforms GPTA for most programs smaller than 30kLoC. That a flow- and context-sensitive analysis could be faster than flow- and context-insensitive analysis came as a surprise to us. In hindsight, this is possible because the information that we can gainfully use is much smaller than commonly thought. Note that a flow- and context-insensitive analysis cannot exploit the small size of usable pointer information because it is small only when considered flow and context sensitively.

The hypothesis that our implementation suffers because of linear search in linked lists was confirmed by an accidental discovery: in order to eliminate duplicate pairs in GPTA, we used our linear list implementation of sets from L-FCPA which never adds duplicate entries. The resulting GPTA took more than an hour for the *hmmer* program instead of the original 246.3 milliseconds! Another potential source of inefficiency concerns the over-eager liveness computation to reduce the points-to pairs in L-CFPA: a new round of liveness is invoked when a new points-to pair for y is discovered for $x = *y$ putting on hold the points-to analysis. This explains the unusually large time

Program	Total no. of functions	No. and percentage of functions for call-string counts							
		0 call strings		1-4 call strings		5-8 call strings		9+ call strings	
		L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA	L-FCPA	FCPA
lbm	22	16 (72.7%)	3 (13.6%)	6 (27.3%)	19 (86.4%)	0	0	0	0
mcf	25	16 (64.0%)	3 (12.0%)	9 (36.0%)	22 (88.0%)	0	0	0	0
bzip2	100	88 (88.0%)	38 (38.0%)	12 (12.0%)	62 (62.0%)	0	0	0	0
libquantum	118	100 (84.7%)	56 (47.5%)	17 (14.4%)	62 (52.5%)	1 (0.8%)	0	0	0
sjeng	151	96 (63.6%)	37 (24.5%)	43 (28.5%)	45 (29.8%)	12 (7.9%)	15 (9.9%)	0	54 (35.8%)
hmmer	584	548 (93.8%)	330 (56.5%)	32 (5.5%)	175 (30.0%)	4 (0.7%)	26 (4.5%)	0	53 (9.1%)
parser	372	246 (66.1%)	76 (20.4%)	118 (31.7%)	135 (36.3%)	4 (1.1%)	63 (16.9%)	4 (1.1%)	98 (26.3%)
	9+ L-FCPA call strings: Tot 4, Min 10, Max 52, Mean 32.5, Median 29, Mode 10								
h264ref	624	351 (56.2%)	?	240 (38.5%)	?	14 (2.2%)	?	19 (3.0%)	?
	9+ L-FCPA call strings: Tot 14, Min 9, Max 56, Mean 27.9, Median 24, Mode 9								

Table 3. Context information for computing usable pointer information is small and sparse.

spent in liveness analysis compared to points-to analysis for programs *parser* and *sjeng*. The number of rounds of analysis required for these programs was much higher than in other programs of comparable size. Finally, GCC's LTO framework has only two options: either to load no CFG or to load all CFGs at the same time. Since the size of the entire program could be large, this affects the locality and hence the cache behaviour.

8 Conclusions and Future Work

We have described a data flow analysis which jointly calculates points-to and liveness information. It is fully flow- and context-sensitive and uses recent refinements of the call-strings approach. One novel aspect of our approach is that it is effectively bi-directional (such analysis seem relatively rarely exploited).

Initial results from our naive prototype implementation were impressive: unsurprisingly our analysis produced much more precise results, but by an order of magnitude (in terms of the size of the calculated points-to information). The reduction of this size allowed our naive implementation also to run faster than GCC's points-to analysis at least for programs up to 30kLoC. This is significant because GCC's analysis compromises both on flow and context sensitivity. This confirms our belief that the usable pointer information is so small and sparse that we can achieve both precision and efficiency without sacrificing one for the other. Although the benefit of precision in efficiency has been observed before [17, 11], we are not aware of any study that shows the sparseness and small size of points-to information to this extent.

We would like to take our work further by exploring the following:

- Improving our implementation in ways such as: using efficient data structures (vectors or hash tables, or perhaps BDDs); improving GCC's LTO framework to allow on-demand loading of individual CFGs instead of loading the complete supergraph; and experimenting with less-eager strategies of invoking liveness analysis.
- Exploring the reasons for the 30kLoC speed threshold; perhaps there are ways in practice to partition most bigger programs (around loosely-coupled boundaries) without significant loss of precision.
- We note that data flow information often only slightly changes when revisiting a node compared to the information produced by the earlier visits. Hence, we plan to explore incremental formulations of L-FCPA.
- GCC passes hold alias information in a per-variable data structure thereby using the same information for every occurrence of the variable. We would like to change this to use point-specific information computed by L-FCPA and measure how client analyses/optimisations benefit from increased precision.

Acknowledgements. Prashant Singh Rawat was supported by GCC Resource Center funding as part of the Government of India's National Resource Center for Free and Open Source Software (NRCFOSS). Empirical measurements were carried out by Prachee Yogi and Aboli Aradhye. Prachee also implemented intraprocedural analysis in Prolog. Ashwin Paranjape was involved in initial explorations. We are grateful to the anonymous referee who requested more detail on our treatment of allocation and summary nodes (Section 5); this helped our presentation.

References

1. Khedker, U.P., Sanyal, A., Karkare, B.: Data Flow Analysis: Theory and Practice. CRC Press, Inc. (2009)
2. Kildall, G.A.: A unified approach to global program optimization. In: Proc. of POPL'73. (1973) 194–206
3. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proc. of PLDI'94. (1994) 242–256
4. Kanade, A., Khedker, U.P., Sanyal, A.: Heterogeneous fixed points with application to points-to analysis. In: Proc. of APLAS'05. (2005) 298–314
5. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
6. Steensgaard, B.: Points-to analysis in almost linear time. In: Proc. of POPL'96. (1996) 32–41
7. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In Muchnick, S.S., Jones, N.D., eds.: Program Flow Analysis : Theory and Applications. Prentice-Hall Inc. (1981)
8. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. of POPL'95. (1995) 49–61
9. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Proc. of CC'92. (1992) 125–140
10. Khedker, U.P., Karkare, B.: Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In: Proc. of CC'08. (2008) 213–228

11. Lhoták, O., Chung, K.A.: Points-to analysis with efficient strong updates. In: Proc. of POPL'11. (2011) 3–16
12. Lhoták, O., Hendren, L.J.: Context-sensitive points-to analysis: Is it worth it? In: Proc. of CC'06. (2006) 47–64
13. Ruf, E.: Context-insensitive alias analysis reconsidered. In: Proc. of PLDI'95. (1995) 13–22
14. Shapiro, M., Horwitz, S.: The effects of the precision of pointer analysis. In: Proc. of SAS'97. (1997) 16–34
15. Hind, M., Pioli, A.: Assessing the effects of flow-sensitivity on pointer alias analyses. In: Proc. of SAS'98. (1998) 57–81
16. Hardekopf, B.C., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Proc. of PLDI'07. (2007) 290–299
17. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proc. of POPL'11. (2011) 17–30
18. Guyer, S.Z., Lin, C.: Client-driven pointer analysis. In: Proc. of SAS'03. (2003) 214–236
19. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Proc. of PLDI'01. (2001) 24–34
20. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: Proc. of OOPSLA'05. (2005) 59–76
21. Zheng, X., Rugina, R.: Demand-driven alias analysis for C. In: Proc. of POPL'08. (2008) 197–208
22. Rosen, B.K.: Linear cost is sometimes quadratic. Proc. of POPL'81 (1981) 117–124
23. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proc. of PLDI'04. (2004) 131–144
24. Choi, J.D., Cytron, R., Ferrante, J.: On the efficient engineering of ambitious program analysis. IEEE Trans. Softw. Eng. **20** (1994) 105–114
25. Chow, F.C., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: Proc. of CC'96. (1996) 253–267
26. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Proc. of PLDI'98. (1998) 97–105
27. Yu, H., Xue, J., Huo, W., Feng, X., Zhang, Z.: Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In: Proc. of CGO'10. (2010) 218–229
28. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Proc. of POPL'95. (1995) 1–12
29. Kahlon, V.: Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In: Proc. of PLDI'08. (2008) 249–259
30. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. **14** (2005) 1–41