Points-To Analysis with Efficient Strong Updates

Ondřej Lhoták Kwok-Chiang Andrew Chung

D. R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{olhotak,kachung}@uwaterloo.ca

Abstract

This paper explores a sweet spot between flow-insensitive and flow-sensitive subset-based points-to analysis. Flow-insensitive analysis is efficient: it has been applied to million-line programs and even its worst-case requirements are quadratic space and cubic time. Flow-sensitive analysis is precise because it allows strong updates, so that points-to relationships holding in one program location can be removed from the analysis when they no longer hold in other locations. We propose a "Strong Update" analysis combining both features: it is efficient like flow-insensitive analysis, with the same worst-case bounds, yet its precision benefits from strong updates like flow-sensitive analysis. The key enabling insight is that strong updates are applicable when the dereferenced points-to set is a singleton, and a singleton set is cheap to analyze. The analysis therefore focuses flow sensitivity on singleton sets. Larger sets, which will not lead to strong updates, are modelled flow insensitively to maintain efficiency. We have implemented and evaluated the analysis as an extension of the standard flow-insensitive points-to analysis in the LLVM compiler infrastructure.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Algorithms, Design, Experimentation, Languages, Performance, Verification

Keywords points-to analysis, flow sensitivity, strong updates, Andersen's analysis, LLVM

1. Introduction

One of the design decisions facing a developer selecting a subset based points-to analysis is flow sensitivity. On one hand, flow-insensitive analyses are well understood, and techniques have been developed that make them quite efficient and scalable (e.g. [2, 12, 15, 19, 25, 27], among many others). On the other hand, flow-sensitive analyses promise potentially more precise results. Recently, there has been a resurgence of interest in techniques that reduce the previously prohibitive cost of flow sensitivity [14, 22, 31, 32].

This paper proposes a hybrid subset-based analysis algorithm that has desirable properties of both flow-insensitive and flow-sensitive analyses. This "Strong Update" analysis provides the key precision benefit that flow sensitivity brings, strong updates. However, its performance is comparable to that of flow-insensitive analysis: in the worst case, it requires quadratic space and cubic time, and in practice, it is almost as fast as flow-insensitive analysis. More precisely, the strong update analysis requires $O(VA)$ space and $O(EV^2)$ time, where V is the number of pointer variables in the program, A is the number of variables whose address is taken (i.e. possible pointer targets), and E is the number of edges in the interprocedural control flow graph.

The idea that enables this good compromise is the realization that the precise points-to sets that matter most are also cheap to propagate, even flow sensitively. A strong update can only be performed if the may-point-to set of the dereferenced pointer corresponds to exactly one (runtime) target; otherwise, the analysis could not guarantee that any one of the possible targets is definitely overwritten. A necessary condition for this case is that the points-to set must be a singleton (i.e., contain one abstract target). When one strong update improves the precision of a given points-to set, the more precise set may enable a chain of further strong updates. Thus in order to be precise overall, an analysis must model these small sets precisely. Yet singleton sets are also very cheap to represent and propagate. In particular, it is possible to propagate singleton sets flow-sensitively without significantly increasing the asymptotic complexity of an otherwise flow-insensitive analysis or its practical running time.

Thus our strong update points-to analysis can be summarized as follows. It is a flow-insensitive subset-based analysis extended with flow-sensitive modeling of singleton sets, which are used to enable strong updates. The analysis maintains sound flow-insensitive points-to sets for all pointers. In addition, it provides flow-sensitive points-to sets for those pointers and at those program points where the sets are singletons. When a flow-sensitive set is available, the analysis uses it, possibly to perform a strong update. When no flow-sensitive set is available (because it is not a singleton), the analysis falls back to the flow-insensitive information. Although we have described the analysis here as a combination of two separate analyses, both analyses are intertwined in the actual algorithm and performed at the same time so that they can query each other. Thus the flow-sensitive analysis improves the precision of the flow-insensitive analysis, and the flow-insensitive analysis provides a fall-back to the flow-sensitive analysis when necessary.

This paper makes the following contributions:

- It identifies and discusses the characteristics of flow-sensitive analyses that give rise to improved precision over flow-insensitive analyses. It argues that strong updates are the most important such characteristic.

POPL'11, January 26–28, 2011, Austin, Texas, USA.

- It presents the hybrid strong update analysis algorithm, first as a system of constraints, and then as an algorithm extending the flow-insensitive algorithm.
- It shows that the worst-case complexity of the strong update analysis is the same as that of the flow-insensitive analysis, quadratic in space and cubic in time.
- It describes an implementation of the strong update analysis in the LLVM compiler infrastructure [24]. The implementation is available for download at `http://plg.uwaterloo.ca/~olhotak/su`.
- It experimentally evaluates the implementation on the SPECINT 2000 and SPECCPU 2006 benchmark suites [33], shows that its practical performance is comparable to that of the flow-insensitive analysis, and that it performs 98% of the strong updates and propagates more precise sets than a flow-insensitive analysis at 98% of the loads at which a fully flow-sensitive analysis would.

The paper is organized as follows. Section 2 presents background material. It first defines the form of the intermediate representation on which the analyses work. It then presents a high level specification, in the form of subset constraints, of three existing analyses: a flow-insensitive analysis and a flow-sensitive analysis without and with strong updates. Section 3 presents the high-level decisions guiding the design of the strong update analysis. It discusses the key benefits of flow sensitivity and assumptions about the intermediate representation that make analyses easier to express. It then presents, at the same high level of subset constraints, the strong update analysis for comparison with the existing flow-insensitive and flow-sensitive analyses. Section 4 presents the strong update analysis algorithm in detail. It also proves the worst-case complexity results. Section 5 presents details of the implementation of the strong update analysis in LLVM as an extension of the flow-insensitive analysis already existing in that framework. Section 6 presents results of an experimental evaluation of the strong update analysis measuring both its practical efficiency and the benefits to precision. The results show that the performance of the strong update analysis is comparable to that of the flow-insensitive analysis, and that the analysis provides the same benefits as a fully flow-sensitive analysis at 98% of stores and loads. Section 7 surveys other work related to efficient flow-sensitive points-to analysis. Finally, Section 8 concludes.

2. Background

This section defines the program model and the notation that will be used throughout the rest of the paper, briefly reviews flow-insensitive subset-based points-to analysis (often called Andersen's analysis [1]), and specifies a flow-sensitive extension of that analysis.

The program model commonly used in the points-to analysis literature and in the points-to analysis implementation in LLVM represents the program using a control flow graph containing the four kinds of pointer-manipulating instructions shown in the left column of Figure 1. More complicated statements that manipulate pointers (such as statements containing multiple levels of indirection) are decomposed into these basic instructions. The ADDROF instruction is used to model all statements that cause a pointer p to point to some new target a. This includes not only statements that take the address of a variable, but also statements that allocate new objects dynamically, in which case the pointer target is the statement at which the allocation takes place, the allocation site. The COPY instruction is used to model all copying of one pointer to another, including the interprocedural copying of arguments to procedure parameters due to procedure calls. The STORE and LOAD instructions model dereferencing of and writes and reads through pointers.

$p = \&a$	$\{a\} \subseteq pt(p)$	[ADDROF]
$p = q$	$pt(q) \subseteq pt(p)$	[COPY]
$*p = q$	$\forall a \in pt(p) \,.\, pt(q) \subseteq pt(a)$	[STORE]
$p = *q$	$\forall a \in pt(q) \,.\, pt(a) \subseteq pt(p)$	[LOAD]

Figure 1. Constraints for flow-insensitive subset-based points-to analysis

For simplicity of presentation, we follow the LLVM convention of separating variables into two disjoint sets of top-level and address-taken variables. The set $\mathcal{A}$ is defined to contain all possible targets of a pointer, including address-taken variables and dynamic allocation sites. The set $\mathcal{P}$ contains all top-level pointer variables. The instructions in Figure 1 are restricted to operate only on top-level pointers $p, q \in \mathcal{P}$, except for the ADDROF instruction that takes the address of an address-taken variable $a \in \mathcal{A}$. If a program contains a variable v violating this restriction (i.e. it has its address taken, and is also used in a copy, store, or load instruction), the program is transformed into an equivalent program that replaces v with a separate top-level pointer p_v and target variable a_v by adding the instruction $p_v = \&a_v$ and replacing all occurrences of v in the original program with $*p_v$. The set of all variables is denoted $\mathcal{V} = \mathcal{P} \cup \mathcal{A}$. We use a, b, and c to range over $\mathcal{A}$, p, q, and r to range over $\mathcal{P}$, and v and w to range over $\mathcal{V}$.

The flow-insensitive points-to relation $pt : \mathcal{V} \to 2^{\mathcal{A}}$, is defined as the least solution to the subset constraints shown in Figure 1. For each pointer in the program, it provides a set of targets to which the pointer may point. The solution can be computed by initializing all points-to sets to the empty set, then iteratively choosing a subset constraint that is violated and propagating the contents of the points-to set on the left-hand-side of the constraint into the right-hand-side, thereby satisfying the constraint. In formal terms, this process is equivalent to applying a monotone function on the cartesian product lattice of the powerset lattices $2^{\mathcal{A}}$ associated with each of the individual points-to sets. The height of this lattice is finite. The constraints therefore have a unique least solution, and the iterative process converges to it [10].

The concretization of a given points-to analysis result is defined as all execution states in which the C expression $v == \&a$ evaluates to false for all v and a for which $a \notin pt(v)$.

The feature that distinguishes a flow-sensitive analysis from a flow-insensitive one is that the flow-sensitive analysis takes control flow between instructions into account and computes a possibly different result for each program point. The subset-based points-to analysis can be extended to be flow-sensitive as shown in Figure 2. Each instruction is annotated with a label $\ell \in \mathcal{L}$ to indicate its position in the control flow graph. The points-to relation is extended with an extra parameter that dictates the program point at which the points-to information applies. The notation $\overline{\ell}$ and $\underline{\ell}$ indicates the program points immediately before and after the instruction labelled ℓ, respectively. For example, $pt[\underline{\ell}](v)$ gives the points-to set of pointer v after the instruction labelled ℓ. The subset constraints modelling the four kinds of instructions are similar to those in the flow-insensitive analysis, except they now relate a points-to set before each instruction with a points-to set after that instruction. The new CFLOW constraints model the effect of control flow: whenever ℓ_2 follows ℓ_1 in the control flow graph, the points-to sets before ℓ_2 contain everything contained in the points-to sets after ℓ_1. The new PRESERVE constraint accounts for the fact that any pointers not affected by an instruction maintain the values that they had before the instruction executed. For each pointer v not in the kill set of the instruction, the points-to set after the instruction contains all the targets that were in the points-to

$$
\begin{array}{lrl}
\ell : p = \&a & \{a\} \subseteq pt[\underline{\ell}](p) & \text{[ADDROF]} \\
\ell : p = q & pt[\overline{\ell}](q) \subseteq pt[\underline{\ell}](p) & \text{[COPY]} \\
\ell : *p = q & \forall a \in pt[\overline{\ell}](p) \,.\, pt[\overline{\ell}](q) \subseteq pt[\underline{\ell}](a) & \text{[STORE]} \\
\ell : p = *q & \forall a \in pt[\overline{\ell}](q) \,.\, pt[\overline{\ell}](a) \subseteq pt[\underline{\ell}](p) & \text{[LOAD]} \\
\ell_1 \in \mathit{pred}(\ell_2) & \forall v \in \mathcal{V} \,.\, pt[\underline{\ell_1}](v) \subseteq pt[\overline{\ell_2}](v) & \text{[CFLOW]} \\
\ell \in \mathcal{L} & \forall v \in \mathcal{V} \setminus \mathit{kill}(\ell) \,.\, pt[\overline{\ell}](v) \subseteq pt[\underline{\ell}](v) & \text{[PRESERVE]}
\end{array}
$$

Figure 2. Constraints for flow-sensitive subset-based points-to analysis

set before the instruction. For a simple implementation of a flow-sensitive analysis, it is sufficient (and sound) to define all the kill sets to be empty, so that the PRESERVE subset constraints apply to every pointer at every instruction.

Following past work on flow-sensitive points-to analysis, we focus on path-insensitive analysis (i.e. the analysis ignores condition expressions in conditional branches). Path-sensitive analyses are a different compromise in the tradeoff between analysis precision and efficiency, and they are beyond the scope of our study.

Additional precision can be obtained using strong updates, which are implemented in the analysis by defining kill sets that are not empty. A strong update occurs when it is known that an instruction completely overwrites a previous value of a given pointer. In this case, the pointer is listed in the kill set of the instruction to prevent the PRESERVE constraints from propagating the previous value of the pointer through the instruction.

To soundly include an abstract pointer v in the kill set, we must be sure that the instruction definitely writes to v, and that the abstract pointer v represents no more than a single concrete pointer in the execution of the program. For example, if v is a dynamic allocation site, an instruction may overwrite one but not all of the objects allocated there, so it would be unsound to include v in the kill set. In Section 5, we will define a set *singletons* $\subseteq \mathcal{V}$ of abstract pointers corresponding to a single concrete pointer at run time.

Precise kill sets to implement strong updates are defined in Figure 3. Each of the ADDROF, COPY, and LOAD instructions overwrites a target top-level pointer p, so that pointer is in the kill set. For a STORE instruction $*p = q$, the kill set depends on the points-to set of p before the instruction. If its size is greater than 1, the analysis cannot determine which of the targets will be overwritten, so the kill set is empty (because no specific target is certain to be overwritten). If its size is exactly 1, and the unique target a is in *singletons*, then the instruction will definitely overwrite a, so a is in the kill set.

For correctness, we must also consider the case when the points-to set of p is empty. It is tempting but incorrect to suggest that in this case, the instruction cannot have any effect (except to dereference a null pointer, halting the program), so the kill set should be empty. Such a definition would violate the monotonicity of the subset constraints, which would invalidate the guarantee of a unique least solution and cause the analysis to loop forever on some programs without converging to a fixed point. Concretely, suppose the points-to set of p before $\ell : *p = q$ were empty, so that PRESERVE constraints would be created at ℓ for all variables. Later, some target a might be added to the points-to set of p and therefore to the kill set of ℓ. This would entail the removal of the PRESERVE constraint for a. But this constraint might have been responsible for causing a to be in the points-to set of p in the first place, so fully removing the constraint would require removing a from the points-to set of p, thus forcing the constraint to be added back again. Thus the analysis would loop forever.

When the points-to set of p is empty, the correct definition of the kill set is $\mathcal{V}$, the set of all variables. As a result, no PRESERVE constraints are generated until the points-to set of p becomes non-empty. No subset constraints ever need to be removed after they

$$
\begin{aligned}
\mathit{kill}(\ell : p = \ldots) &\triangleq \{p\} \\
\mathit{kill}(\ell : *p = q) &\triangleq \begin{cases} \{\} & \text{if } \left|pt[\overline{\ell}](p)\right| > 1 \\ \{\} & \text{if } pt[\overline{\ell}](p) = \{a\} \wedge a \notin \mathit{singletons} \\ \{a\} & \text{if } pt[\overline{\ell}](p) = \{a\} \wedge a \in \mathit{singletons} \\ \mathcal{V} & \text{if } pt[\overline{\ell}](p) = \{\} \end{cases}
\end{aligned}
$$

Figure 3. Definition of kill sets

$$
\begin{array}{l}
1 : p_a = \&a \\
2 : p_b = \&b \\
3 : p_c = \&c \\
4 : *p_a = p_b \\
5 : *p_a = p_c
\end{array}
$$

Figure 4. Example of straight-line code on which flow sensitivity improves precision

are generated, so the non-monotonicity of the constraints and non-termination of the analysis are avoided. Suggesting that a dereference of an empty points-to set kills the values of all pointers may be surprising, but it is sound. If p can only point to null, dereferencing p causes the program to abort, and therefore the values of any pointers before the null dereference cannot be observed anywhere in the program after the dereference. Since the statements after the null dereference are unreachable, from the point of view of abstract interpretation, their points-to sets should be $\bot$, and indeed, in the points-to domain, this value corresponds to all points-to sets being empty.

3. Design Overview

In this section, we present the design objectives for the strong update points-to analysis. We begin with a discussion of the beneficial effects of flow sensitivity in a points-to analysis that are desirable in our strong update analysis. We then discuss the performance tradeoffs that are made to achieve those precision improvements.

3.1 Benefits of flow sensitivity

The advantage of flow sensitivity can be classified into two benefits: handling of straight-line code and strong updates. Of the two, strong updates generally provide the greater improvement in precision. The strong update algorithm that we will present aims to provide the benefit of strong updates at a cost comparable to that of a flow-insensitive analysis.

The flow-sensitive points-to analysis that was presented in Figure 2 provides some improvement in precision even without strong updates (i.e. when all of the kill sets are defined to be empty). If the program being analyzed contains code that is not inside any loop and can never be executed more than once, a flow-sensitive analysis can determine that facts established at the end of such code do not yet hold at the beginning of such code. For example, consider the short program in Figure 4. The program sets pointer a to point to b in line 4 and then to c in line 5. A flow-insensitive analysis would report that $pt(a) = \{b, c\}$. A flow-sensitive analysis, even one without strong updates, would determine that after line 4, a does not yet point to c: $pt[\underline{4}](a) = \{b\}$. Thus flow sensitivity improves precision for this program even without strong updates.

However, this benefit is brittle: if the same code appeared inside a loop, the analysis would determine that $pt[\ell](a) = \{b, c\}$ at all points ℓ. More generally, we can show that the points-to sets at every point inside a given loop are always identical:

Proposition 1. *Suppose that there is a cycle in the interprocedural control flow graph leading from ℓ_1 to ℓ_2 and back to ℓ_1. Then if all the kill sets are empty, $pt[\underline{\ell_1}](v) = pt[\underline{\ell_2}](v)$ for every variable v.*

Proof. The cycle in the control flow graph induces a similar cycle of CFLOW and PRESERVE constraints $pt[\underline{\ell_1}](v) \subseteq \cdots \subseteq pt[\underline{\ell_2}](v) \subseteq \cdots \subseteq pt[\underline{\ell_1}](v)$. Thus $pt[\underline{\ell_1}](v) = pt[\underline{\ell_2}](v)$. □

Most of the code of most programs is found inside loops. Many compiler optimizations target loops because loop bodies are where the most frequently executed code appears. Even many long straight-line sequences of code occur inside a large outer loop. For example, long-running programs such as web servers or database servers run most of their code inside an outer loop that handles individual requests. Relying on code to be outside of any loop is also brittle. For example, when a program is incorporated into a benchmark suite, it is generally invoked from a test harness that executes it multiple times, in a loop. In all of these cases, due to Proposition 1, a flow-sensitive analysis without strong updates would compute the same points-to sets at all program points inside the loop. That is, its result would be no more precise than that of a flow-insensitive analysis.

We therefore focus the design of the strong update analysis algorithm on providing the benefits of strong updates at low cost. The benefit of precisely handling straight-line code is minimal, and it is difficult to achieve without an expensive analysis that maintains distinct, large points-to sets at different program points. On the other hand, we will show how to achieve the more significant benefit of strong updates within the quadratic space and cubic time bounds of a flow-insensitive analysis.

3.2 Using SSA form for strong updates of top-level variables

The kill sets from Figure 3 define strong updates of both top-level variables (the first definition in the figure) and of address-taken pointer targets (the second definition). It is well known that the effect of strong updates of top-level variables can be easily achieved by first converting the program to Static Single Assignment (SSA) form [9]. In SSA form, every variable is written to only once. Conversion to SSA form requires identifying all of the writes to a variable. Therefore, for a program with pointers, SSA conversion requires points-to information to enumerate the indirect writes to variables through pointers, so full SSA conversion cannot be done before the points-to analysis. However, since top-level variables cannot be accessed through pointers, it is possible to convert the top-level variables into SSA form prior to points-to analysis. Specifically, we require the program to be converted to strict SSA form, which enforces that every use of a variable is dominated by its (unique) definition. We can show that for a program whose top-level variables are in strict SSA form, a flow-insensitive analysis provides the precision of flow-sensitive analysis with strong updates for top-level variables:

Proposition 2. *Given a program whose top-level variables are in strict SSA form, a top-level variable p whose unique definition is at ℓ_d, and an arbitrary label ℓ_u at which p is used, a flow-sensitive points-to analysis with strong updates will determine that $pt[\overline{\ell_u}](p) = pt[\underline{\ell_d}](p)$.*

Proof. Since the program is in strict SSA form, there is a path in the ICFG from ℓ_d to ℓ_u, so $pt[\underline{\ell_d}](p) \subseteq \cdots \subseteq pt[\overline{\ell_u}](p)$ (using CFLOW and PRESERVE constraints and the fact that no instruction other than ℓ_d kills p). Notice that in the constraints in Figure 2, whenever the points-to set $pt[\ell](p)$ of a top-level variable p appears on the right-hand side of a constraint, either there is a definition of p at ℓ, or the left-hand side of the constraint also contains the same top-level variable (i.e., the constraint is $pt[\ell'](p) \subseteq pt[\ell](p)$, where ℓ' is some other label). The only label ℓ for which there can be a constraint whose left-hand side is not a points-to set of p is $\underline{\ell_d}$, the unique definition of p. Therefore, every path $\{a\} \subseteq \cdots \subseteq pt[\overline{\ell_u}](p)$ of constraints from a pointer target $\&a$ to $pt[\overline{\ell_u}](p)$ must pass through $pt[\underline{\ell_d}](p)$. Since the analysis finds the least solution, the only pointer targets in $pt[\overline{\ell_u}](p)$ will be ones for which there is such a path, and are therefore also in $pt[\underline{\ell_d}](p)$. That is, $pt[\overline{\ell_u}](p) \subseteq pt[\underline{\ell_d}](p)$. Since we also showed that $pt[\underline{\ell_d}](p) \subseteq pt[\overline{\ell_u}](p)$, we conclude that $pt[\overline{\ell_u}](p) = pt[\underline{\ell_d}](p)$. □

$\ell : p = \&a$	$\{a\} \subseteq pt(p)$	[ADDROF]
$\ell : p = q$	$pt(q) \subseteq pt(p)$	[COPY]
$\ell : *p = q$	$\forall a \in pt(p)\ .\ pt(q) \subseteq pt[\underline{\ell}](a)$	[STORE]
$\ell : p = *q$	$\forall a \in pt(q)\ .\ pt[\overline{\ell}](a) \subseteq pt(p)$	[LOAD]
$\ell_1 \in pred(\ell_2)$	$\forall a \in \mathcal{A}\ .\ pt[\underline{\ell_1}](a) \subseteq pt[\overline{\ell_2}](a)$	[CFLOW]
$\ell \in \mathcal{L}$	$\forall a \in \mathcal{A} \setminus kill(\ell)\ .\ pt[\overline{\ell}](a) \subseteq pt[\underline{\ell}](a)$	[PRESERVE]

Figure 5. Constraints for flow-sensitive subset-based points-to analysis on SSA form

As a result of Proposition 2, we can merge all of the flow-sensitive points-to sets $pt[*](p)$ of p into a single flow-insensitive points-to set $pt(p)$ without reducing the precision of the analysis. The subset constraints after this simplification are shown in Figure 5. Note that the CFLOW and PRESERVE constraints for a top-level variable p reduce to the trivial $pt(p) \subseteq pt(p)$ and are therefore not needed. While this analysis is as precise as the flow-sensitive analysis, it has regained some of the simplicity of the flow-insensitive analysis. The space required to store the points-to sets has been reduced from $O(|\mathcal{V}||\mathcal{L}||\mathcal{A}|)$ to $O(|\mathcal{P}||\mathcal{A}| + |\mathcal{A}|^2|\mathcal{L}|)$.

SSA transformation has also been used as a preparatory step for sparse analysis of top-level variables [17], which follows def-use chains between top-level variables instead of paths in the control flow graph. SSA form simplifies these def-use chains.

3.3 Quadratic-space representation of points-to sets of pointer targets

To achieve space requirements that are quadratic in the size of the program, we must further reduce the $|\mathcal{A}|^2|\mathcal{L}|$ term in the above bound, which is due to the size of the points-to sets $pt[\underline{\ell}](a)$ of address-taken variables. The strong update algorithm does this by taking advantage of the following insights:

- Most of the precision benefit of flow-sensitivity comes from strong updates.
- A strong update requires the points-to set of the dereferenced pointer to contain at most one pointer target.
- A singleton points-to set is cheap to store and manipulate.
- Any larger points-to sets will not directly enable strong updates, so there is little benefit in spending much space or time storing them.

Therefore, the strong update analysis stores points-to sets of pointer targets flow sensitively when they are singletons, and only flow insensitively when they are larger.

To implement this, we define the singleton-set lattice $\mathcal{S}$ as shown in Figure 6. An element of this lattice is either the empty set, a singleton set, or $\top$ indicating some larger set. For each program point ℓ and for each pointer target a, the analysis stores an element of this lattice $su[\underline{\ell}](a)$. However, unlike in a constant propagation analysis, the value of $\top$ in the strong update analysis does not immediately imply that a pointer can point to anything. Instead, the analysis *also* stores a flow-insensitive points-to set $pt(a)$ for each pointer target a to be used when the flow-sensitive analysis indicates $\top$. The STORE constraint updates both $pt(a)$ and $su[\underline{\ell}](a)$ if

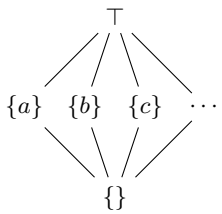

Figure 6. The singleton-set lattice $\mathcal{S}$

$$
\begin{array}{lrl}
\ell : p = \&a & \{a\} \subseteq pt(p) & [\textsc{AddrOf}] \\
\ell : p = q & pt(q) \subseteq pt(p) & [\textsc{Copy}] \\
\ell : *p = q & \forall a \in pt(p)\ .\ pt(q) \sqsubseteq su[\underline{\ell}](a) & [\textsc{Store}] \\
 & \forall a \in pt(p)\ .\ pt(q) \subseteq pt(a) & \\
\ell : p = *q & \forall a \in pt(q)\ .\ ptsu[\overline{\ell}](a) \subseteq pt(p) & [\textsc{Load}] \\
\ell_1 \in pred(\ell_2) & \forall a \in \mathcal{A}\ .\ su[\underline{\ell_1}](a) \sqsubseteq su[\overline{\ell_2}](a) & [\textsc{CFlow}] \\
\ell \in \mathcal{L} & \forall a \in \mathcal{A} \setminus kill(\ell)\ .\ su[\overline{\ell}](a) \sqsubseteq su[\underline{\ell}](a) & [\textsc{Preserve}]
\end{array}
$$

Where $ptsu[\overline{\ell}](a) \triangleq \begin{cases} su[\overline{\ell}](a) & \text{if } su[\overline{\ell}](a) \neq \top \\ pt(a) & \text{if } su[\overline{\ell}](a) = \top \end{cases}$

Figure 7. Constraints for Strong Update Analysis

the points-to set of the variable q being stored is a singleton (and sets $su[\underline{\ell}](a)$ to $\top$ if it is not). When the LOAD constraint needs the points-to set of a at $\overline{\ell}$, it first looks for a possible singleton by consulting $su[\overline{\ell}](a)$; if this returns $\top$, it falls back on the points-to set $pt(a)$. This is implemented by the *ptsu* function in Figure 7. Only the small *su* sets need to be propagated flow-sensitively along the control flow edges of the program, so the CFLOW and PRESERVE constraints act only on these sets. The possibly large *pt* sets are stored only once for the whole program. As a result, the space bound of this representation is $O(|\mathcal{P}||\mathcal{A}| + |\mathcal{A}|^2 + |\mathcal{L}||\mathcal{A}|)$, reflecting the space requirements of the points-to sets of $\mathcal{P}$, the points-to sets of $\mathcal{A}$, and the sets *su*, respectively.

Of course, it is possible to construct examples on which this simplification loses precision compared to a fully flow-sensitive analysis:

```
p = &a;
*p = &b;
if(*) *p = &c; else *p = &d;
q = *p;
r = *q;
```

Both the strong update analysis and the flow-sensitive analysis will perform strong updates at all three stores. At the final load, $su[\overline{\ell}](a) = \top$ due to the control flow merge, so the flow-insensitive points-to set $pt(a) = \{b, c, d\}$ will be propagated to r in the strong update analysis, whereas the flow-sensitive analysis would propagate $\{c, d\}$. However, as we will see in Section 6, such examples are very rare in practice.

Although the asymptotic complexity of the strong update representation is low, we should also consider actual behaviour on realistic programs. In most programs, only a small number of pointer targets a will have singleton points-to sets at a given label. Thus the representation of the sets $su[\underline{\ell}]$ at each program point ℓ should be worst-case linear not only in $|\mathcal{A}|$, but also in the (much smaller) subset of pointer targets a for which $su[\underline{\ell}](a)$ is a singleton. Since $su[\underline{\ell}](a) = \top$ for most values of a, we use a map storing only those pointer targets whose associated value is not $\top$, and default to $\top$ when we do not find a particular pointer target in the table. There is one other special case, however. Following a store through a pointer p whose points-to set is empty, $su[\underline{\ell}](a) = \{\}$ for all values of a. Therefore, in our implementation, we use a boolean flag to indicate this special value. When the flag is true, $su[\underline{\ell}](a) = \{\}$ for all values of a. When the flag is false, a map stores the values of $su[\underline{\ell}](a)$ other than $\top$, and $\top$ is returned when a pointer target is not found in the map. This hybrid representation is compact in all of the common use cases. As we will see in Section 6, the mean number of entries in each map is less than 2.2 in all of the SPEC benchmarks.

3.4 Sparse Allocation of Labels

Almost all practical flow-sensitive analyses share the representation of flow-sensitive facts for successive program points at which the facts cannot change, and the same can be done for the strong update analysis. We do this by removing redundant labels from the program representation. In our discussion thus far, every instruction was assigned its own unique label ℓ. However, most instructions do not change the points-to sets of address taken variables $a \in \mathcal{A}$. The only instructions whose outgoing *su* value is different from its incoming *su* value are STORE instructions, instructions with multiple control flow predecessors (i.e. control flow merges), and the very first instruction in the program.

When it is certain that the *su* sets at one instruction are identical to those at its predecessor, we assign both instructions the same label. Specifically, we relabel the instructions in the program in the following way. First, every STORE instruction is assigned a unique label. Second, at every control flow merge point, we add a new no-op instruction and give it a unique label. The *su* value computed at this label will be the join of the *su* values at the control flow predecessors. Third, we add a no-op instruction at the very beginning of the program and also give it a unique label. The *su* value computed at this label will be $\lambda a.\top$, meaning that no flow-sensitive information is known. Finally, we label every other instruction with the label of its (unique) control flow predecessor. As a result, every label in the program can be classified as either a store, a merge, or a clear (the beginning of the program). In particular, every LOAD instruction in the program is now labelled with the same label as the most recent instruction at which the *su* value may have changed.

After this relabelling process, it is necessary to update the references to program points in the constraints from Figure 7. The notation $\overline{\ell}$ and $\underline{\ell}$ for the program points immediately before and after the instruction at ℓ may no longer designate the appropriate program points when there are multiple instructions with the same label ℓ. Instead of distinguishing $su[\overline{\ell}]$ and $su[\underline{\ell}]$, the strong update analysis algorithm defines one strong update value $su[\ell]$ for each label ℓ. The value $su[\ell]$ is defined to describe the program state at the point immediately after the first instruction at label ℓ. By construction, this first instruction is always a STORE or a no-op. Because a STORE instruction is always the first instruction at its label, the $su[\underline{\ell}]$ in the STORE constraint from Figure 7 is equivalent to $su[\ell']$, where ℓ' is the new label of the instruction. Similarly, because a LOAD instruction always comes after the first instruction at a label, the $ptsu[\overline{\ell}]$ in the LOAD constraint is equivalent to $ptsu[\ell']$, where ℓ' is the new label of the instruction. Thus, in the remainder of this paper, we no longer use the notation $\overline{\ell}$ and $\underline{\ell}$, but use simply ℓ to refer to the program point immediately after the first instruction with label ℓ.

4. Strong Update Analysis Algorithm

This section presents the strong update analysis algorithm used to solve the constraints of Figure 7. The algorithm is an extension of the flow-insensitive subset-based points-to analysis algorithm already implemented in LLVM and other compilers. We therefore begin with a brief review of that algorithm, and follow it with an explanation of the extensions that enable strong updates.

The original flow-insensitive algorithm that solves the constraints of Figure 1 is shown in Figure 8. The core data structure,

```
1   foreach ADDROF constraint p = &a do pt(p) ∪= {a}; worklist ∪= {p} od
2   foreach COPY constraint p = q do graph ∪= {q → p} od
3   while worklist ≠ {} do
4     remove a variable v from worklist
5     Δ ← pt(v) \ oldpt(v)
6     oldpt(v) ← pt(v)
7     foreach STORE constraint ∗v = q do foreach a ∈ Δ do AddEdge(q, a) od od
8     foreach LOAD constraint p = ∗v do ProcessLoad(p, Δ) od
9     foreach v → w ∈ graph do
10      pt(w) ∪= Δ
11      if pt(w) changed then worklist ∪= {w} fi
12    od
13  od
14  proc ProcessLoad(p, Δ)
15    foreach a ∈ Δ do AddEdge(a, p) od
16  endproc
17  proc AddEdge(v, w)
18    if v → w ∉ graph then graph ∪= {v → w}; pt(w) ∪= pt(v); if pt(w) changed then worklist ∪= {w} fi fi
19  endproc
```

Figure 8. Original Flow-insensitive Points-to Analysis Algorithm in LLVM

graph, maintains a set of edges corresponding to the subset constraints being solved. The presence of the edge $v \to w$ corresponds to the subset constraint $pt(v) \subseteq pt(w)$. The *graph* is initialized with the constraints corresponding to COPY instructions in Line 2, and the constraints induced by STORE and LOAD instructions are added to it as they are discovered during the analysis. The *worklist* keeps track of the variables $v \in \mathcal{V}$ whose points-to set has grown since the variable was last processed. The body of the main loop in Lines 3 to 13 is executed for each such variable. In Lines 9 to 12, the new elements are propagated along the edges in the constraint *graph*; as a result, all of the subset constraints with v on their left-hand side become satisfied. Any other variables whose points-to sets grow in the process are added to the worklist. Lines 7 and 8 and the ProcessLoad and AddEdge helper procedures add new subset constraints induced by STORE and LOAD instructions to the *graph*. Whenever a new constraint $v \to w$ is added, the AddEdge procedure immediately propagates the existing contents of $pt(v)$ into $pt(w)$ in Line 18. This is necessary because the normal propagation in Lines 9 to 12 propagates only the part of the points-to set that was added since the last propagation. The algorithm maintains the invariant that if for any variable v, there may be a constraint $pt(v) \subseteq pt(w)$ that is not satisfied, then v is on the worklist. Therefore, once the worklist empties, all of the constraints are satisfied. Every iteration increases the size of $oldpt(v)$, and since every *oldpt* is a subset of $\mathcal{A}$, the iteration must eventually terminate.

The extended algorithm that enables strong updates and solves the constraints of Figure 7 is shown in Figure 9. The lines marked with asterisks are additions to the original flow-insensitive algorithm. Lines not marked with asterisks are identical to or only trivially changed from corresponding lines in the flow-insensitive algorithm of Figure 8.

An important change is in the worklist: in the strong update algorithm, the worklist holds not only variables v whose subset constraints need to be reprocessed, but additionally labels ℓ whose *su* constraints need to be reprocessed. More precisely, the algorithm maintains the following invariants:

1. If there is a constraint $pt(v) \subseteq pt(w)$ that is not satisfied, then v is on the worklist.
2. If there is a LOAD or STORE instruction dereferencing p that induces subset constraints not already in *graph*, then p is on the worklist.
3. If there is a constraint $ptsu[\ell](a) \subseteq pt(p)$ induced by a LOAD that is not satisfied, then a is on the worklist.
4. If there is a constraint of the form $su[\ell](a) \sqsubseteq su[\ell'](a')$ that is not satisfied, then ℓ is on the worklist.
5. If there is a STORE instruction $(\ell : *p = q)$ that induces the constraint $pt(q) \sqsubseteq su[\ell](a)$ and this constraint is not satisfied, then ℓ is on the worklist.

The first two invariants were already present in the original flow-insensitive points-to analysis algorithm. Invariant 3 is a variation of Invariant 1 adapted to the modified constraint involving *ptsu* that is induced by a LOAD instruction. Invariants 4 and 5 ensure that all violated constraints involving *su* are tracked by the worklist and eventually satisfied. We will explain how the invariants are maintained shortly.

First, however, we explain how the algorithm processes a label ℓ appearing on the worklist. As was explained in Section 3.4, each label is associated with either a clear, a control flow merge, or a unique store instruction $\ell : *p = q$. The first two possibilities are handled in the obvious manner in Lines 17 and 18. The interesting case is that of a STORE instruction. If $pt(p)$ is empty, then $su[\ell]$ remains at $\bot$ (i.e. $\lambda a.\{\}$), as was explained in Section 2, so nothing needs to be done (Line 20). Note that it is not possible for $pt(p)$ to be empty when $su[\ell]$ is not $\bot$, because all of the code that modifies $su[\ell]$ is conditional on $pt(p)$ being non-empty, and $pt(p)$ never shrinks, so once it is non-empty, it can never become empty again. When $pt(p)$ is non-empty, the algorithm needs to establish the constraints $\forall a \in pt(p) \,.\, pt(q) \sqsubseteq su[\ell](a)$ due to the STORE instruction. The algorithm first converts $pt(q)$ into an element of the singleton set lattice of Figure 6, substituting $\top$ if $pt(q)$ is not a singleton set; this is done by the PtToSu procedure. Then a strong or weak update is done to $su[\ell]$. If the points-to set of p is a singleton $\{a\}$, the target a of p is certain to be overwritten by the store, so the algorithm simply assigns PtToSu(q) to $su[\ell](a)$, overwriting the existing value (which came from the control flow predecessor of ℓ in Line 21). This is a strong update (Line 24). If $pt(p)$ is not a singleton, weak updates to all the locations a to which p may be pointing are performed, by joining PtToSu(q) with the existing value of $su[\ell](a)$ which came from the control flow predecessor of ℓ (Line 25).

The processing of LOAD instructions is updated to take advantage of the flow-sensitive information available in *su* in Lines 34

```
1   foreach ADDROF constraint p = &a do pt(p) ∪= {a}; worklist ∪= {p} od
2   foreach COPY constraint p = q do graph ∪= {q → p} od
3   while worklist ≠ {} do
4     remove a variable v or a label ℓ from worklist
5     if a variable v was removed then
6       Δ ← pt(v) \ oldpt(v)
7       oldpt(v) ← pt(v)
8*      foreach STORE constraint ℓ : *v = q do worklist ∪= {ℓ} od
9*      worklist ∪= affected[v]
10      foreach STORE constraint ℓ : *v = q do foreach a ∈ Δ do AddEdge(q, a) od od
11      foreach LOAD constraint ℓ : p = *v do ProcessLoad(ℓ, p, Δ) od
12      foreach v → w ∈ graph do
13        pt(w) ∪= Δ
14        if pt(w) changed then worklist ∪= {w} fi
15      od
16*   else // a label ℓ was removed
17*     if ℓ is a clear then su[ℓ] ← λa.⊤
18*     else if ℓ is a merge then su[ℓ] ← ⨆_{ℓ' ∈ pred(ℓ)} su[ℓ']
19*     else // ℓ is a store *p = q
20*       if pt(p) = {} then continue fi
21*       su[ℓ] ← su[pred(ℓ)]
22*       if |pt(q)| ≤ 1 then affected[q] ∪= {ℓ} else affected[q] \= {ℓ} fi
23*       if pt(p) = {a} and a ∈ singletons
24*         then su[ℓ](a) ← PtToSu(q) // strong update
25*         else foreach a ∈ pt(p) do su[ℓ](a) ⊔= PtToSu(q) od fi // weak update
26*     fi
27*     if su[ℓ] changed then
28*       worklist ∪= succ(ℓ)
29*       foreach LOAD constraint ℓ : p = *q do ProcessLoad(ℓ, p, pt(q)) od
30*     fi
31  fi od
32  proc ProcessLoad(ℓ, p, Δ)
33    foreach a ∈ Δ do
34*     if su[ℓ](a) = ⊤
35        then AddEdge(a, p)
36*       else pt(p) ∪= su[ℓ](a); if pt(p) changed then worklist ∪= {p} fi fi
37  od endproc
38  proc AddEdge(v, w)
39    if v → w ∉ graph then graph ∪= {v → w}; pt(w) ∪= pt(v); if pt(w) changed then worklist ∪= {w} fi fi
40  endproc
41* proc PtToSu(q)
42*   if |pt(q)| ≤ 1 and pt(q) ⊆ singletons then return pt(q) else return ⊤ fi
43* endproc
```

Figure 9. Strong Update Points-to Analysis Algorithm

and 36. These lines simply implement the *ptsu* function and the modified LOAD constraint that uses it from Figure 7. Whereas in the original flow-insensitive algorithm, a subset constraint $a \rightarrow p$ was added to *graph* unconditionally, it is now done only when $su[\ell](a)$ is $\top$; otherwise, only $su[\ell](a)$ is propagated to $pt(p)$.

The algorithm must ensure that it maintains the invariants enumerated earlier. Invariants 1 and 2 are guaranteed by the existing code from the original flow-insensitive algorithm and by the similar addition of p to the *worklist* in Line 36. Invariant 3 for a LOAD $\ell : *p = q$ can be violated when, for some $a \in pt(p)$, either $su[\ell](a)$ changes, or $su[\ell](a) = \top$ and $pt(a)$ changes. The first case is handled by Line 29, which calls ProcessLoad, which updates $pt(p)$ to restore the invariant in Line 36. The second case is handled the same way as in the original flow-insensitive algorithm: when $su[\ell](a)$ becomes $\top$, an edge $a \rightarrow p$ is added to *graph* in Line 35, which establishes the invariant and ensures that it remains established in response to changes in $pt(a)$ using the normal propagation code of Lines 12 to 15. Invariant 4 applies to constraints modelling control flow in the program. Line 28 restores the invariant by ensuring that whenever $su[\ell]$ changes, every control-flow successor of ℓ is added to the *worklist*. Invariant 5 is the most complicated. For a given STORE $\ell : *p = q$, the invariant can be invalidated when either $pt(p)$ or $pt(q)$ grows. Growth of $pt(p)$ is detected by the loop on Line 8. Growth of $pt(q)$ is handled by Line 9, by adding all *affected* stores to the *worklist*. The *affected* array is used to keep track of all the STORES $\ell : *p = q$ whose invariant may be invalidated by a change in $pt(q)$. These are all stores whose right-hand side is q, but excluding those for which $pt(q)$ was already seen to be a non-singleton in Line 22 and whose *su* values are therefore already $\top$. Line 22 ensures that the *affected* array is correctly maintained.

The invariants ensure that when the *worklist* is empty, all of the constraints of Figure 7 are satisfied. A variable v is added to the worklist only when $pt(v)$ grows. A label ℓ is added to the worklist only when some $su[\ell']$ grows or when ℓ labels a STORE $*p = q$ and either $pt(p)$ or $pt(q)$ has grown. Since each $pt(v)$ and $su[\ell]$ can grow

only a finite number of times, the algorithm eventually terminates at a fixed point that satisfies all of the constraints. Since each update of $pt(v)$ or $su[\ell]$ is the application of a monotone function, and since the algorithm begins with all of these values at $\bot$, the fixed point at which it converges is the least fixed point of all the constraints.

4.1 Worst-case complexity

As we have already discussed in Section 3.3, the strong update algorithm maintains the quadratic space bound of the flow-insensitive points-to analysis algorithm. We will now show that the strong update algorithm also maintains the cubic time bound of the flow-insensitive algorithm.

For the worst-case analysis, we assume that the set propagation operation $s_1 \cup= s_2$ takes time proportional to the size of the set being propagated (i.e. $O(|s_2|)$ time).

Lemma 1. *The total number of times that a variable is removed from the worklist is $O(|\mathcal{V}||\mathcal{A}|)$, and the sum of the sizes of all the sets Δ computed in Line 6 is also $O(|\mathcal{V}||\mathcal{A}|)$.*

Proof. A given variable v is added to the worklist only when $pt(v)$ changes. Since the maximum size of $pt(v)$ is $|\mathcal{A}|$, and elements are never removed from $pt(v)$, $pt(v)$ can only change $|\mathcal{A}|$ times. Moreover, the sum of all the increases in the size of $pt(v)$ is at most $|\mathcal{A}|$. Thus a variable is added to the worklist $O(|\mathcal{V}||\mathcal{A}|)$ times and the sum of the sizes of Δ is also $O(|\mathcal{V}||\mathcal{A}|)$. □

Lemma 2. *The total number of times that a label is removed from the worklist is $O(E|\mathcal{A}|)$, where E is the number of edges in the interprocedural control flow graph.*

Proof. A given label ℓ is added to the worklist only when $su[\ell']$ changes for some $\ell' \in pred(\ell)$, or, if ℓ is a store $*p = q$, when $pt(p)$ or $pt(q)$ changes. The total number of times that the former can happen is at most $2E|\mathcal{A}|$, since for any given $a \in \mathcal{A}$, $su[\ell'](a)$ can change at most twice (from an empty set to a singleton, then to $\top$). The total number of times that the latter can happen is $2|\mathcal{A}|$ for any given store, or a total of $2|\mathcal{L}||\mathcal{A}|$ times. Since every label in the control flow graph has a predecessor (else it would not be reachable), $|\mathcal{L}| < E$, and so the total number of times that a label can be added to the worklist is $O(E|\mathcal{A}|)$. □

Theorem 1. *The worst-case running time of the strong update algorithm is $O(E|\mathcal{V}|^2)$, where E is the number of edges in the interprocedural control flow graph. Thus it is cubic in the size of the program being analyzed.*

Proof. By Lemma 1, the block from Line 6 to 15 is executed at most $O(|\mathcal{V}||\mathcal{A}|)$ times. Most of the lines in this block take at most $O(\max\{|\mathcal{V}|, |\mathcal{L}|\})$ time. The only exceptions are Lines 10 and 13, which take $O(|\mathcal{L}||\Delta|)$ and $O(|\mathcal{V}||\Delta|)$ time. Since the sum of the sizes of all the Δ sets is $O(|\mathcal{V}||\mathcal{A}|)$, the total time spent in these lines is $O(|\mathcal{V}||\mathcal{A}|(|\mathcal{L}| + |\mathcal{V}|))$. Since $|\mathcal{V}|$ is in $O(|\mathcal{L}|)$, the total amount of time spent in the block from Line 6 to 15 is $O(|\mathcal{V}||\mathcal{A}||\mathcal{L}|)$.

By Lemma 2, the block from Line 17 to 26 is executed at most $O(E|\mathcal{A}|)$ times. All of the operations in it complete in $O(|\mathcal{A}|)$ time, so the total time spent in this block is $O(|\mathcal{A}|^2|\mathcal{L}|)$.

For each load $\ell : p = *q$, ProcessLoad is called only when $su[\ell]$ or $pt(q)$ changes, each of which can happen $O(|\mathcal{A}|)$ times. Each call to ProcessLoad completes in $O(\mathcal{A})$ time. Therefore the total time spent in ProcessLoad is $O(|\mathcal{A}|^2|\mathcal{L}|)$.

AddEdge does a propagation taking $O(|\mathcal{A}|)$ time, but only when a new edge is added to *graph*, which can happen at most $O(|\mathcal{V}|^2)$ times, so the total time spent in AddEdge is $O(|\mathcal{V}|^2|\mathcal{A}|)$.

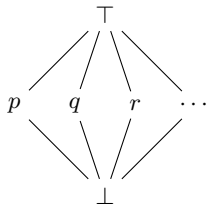

Figure 10. The top-level variable equivalence lattice

All of the bounds on the total time spent in each section of the algorithm are in $O(E|\mathcal{V}^2|)$, so the overall algorithm completes in $O(E|\mathcal{V}^2|)$ time. □

In general graphs, the number of edges can be up to quadratic in the number of vertices. However, it is well known that control flow graphs are very sparse, since most instructions have one successor, some have two (conditional branches), and few rare ones have more (switch statements and indirect function calls). Other complexity analyses of flow-sensitive points-to analysis also rely on this empirical fact [13].

4.2 An improvement: equivalence to top-level variables

The precision of the strong update algorithm can be further improved "for free" by a small extension to the lattice from which $su[\ell](a)$ values are chosen. Given a store $*p = q$ where $pt(p) = \{a\}$, the lattice presented so far (and shown in Figure 6) can represent the fact that $pt(a)$ is a singleton set after the store, if $pt(q)$ happens to be a singleton set. However, the analysis can be easily extended to track that $pt(a) = pt(q)$ even when $pt(q)$ is not a singleton, and this extension has no effect on the asymptotic complexity. In the extended analysis, $su[\ell]$ maps each address taken variable a to a pair $\langle \alpha, \beta \rangle$. The α component is a value from the singleton set lattice, as in the original analysis. The β component is an element of the lattice shown in Figure 10: it is either a top-level variable p, or $\top$ or $\bot$. For example, the pair $su[\ell](a) = \langle \{b\}, p \rangle$ indicates that at ℓ, $pt(a) = \{b\} = pt(p)$, while $su[\ell](a) = \langle \top, p \rangle$ indicates that although $pt(p)$ may not be a singleton set, $pt(a) = pt(p)$.

To adapt the algorithm to this extended lattice, only minor changes are needed. The if statement in Line 42 is updated to return $\langle pt(q), q \rangle$ in the then clause and $\langle \top, q \rangle$ in the else clause. To take advantage of the additional information, an extra else-if clause is added to the if statement in Line 34. When $su[\ell](a)$ is $\langle \top, q \rangle$ but not $\langle \top, \top \rangle$, this new clause calls AddEdge(q, p) so that the contents of $pt(q)$ (which the *su* information says are equal to $pt(a)$) are propagated to $pt(p)$.

This simple extension increases the height of the lattice that $su[\ell](a)$ ranges over from 3 to only 4, so it does not affect the asymptotic complexity of the algorithm and has a negligible effect on actual running times. Knowing that $pt(a) = pt(p)$ when $pt(p)$ is not a singleton may not directly enable additional strong updates, but it can yield some improvement in analysis precision.

4.3 Constraint optimization

For flow-insensitive points-to analysis, several techniques have been developed to speed up the analysis by simplifying the subset constraints [12, 15, 16, 26, 28]. With suitable modifications, these techniques can also be used for strong update analysis. We have adapted and implemented strong update versions of the three techniques used in LLVM: Hash-based Value Numbering (HVN), HVN with Union (HU), and Hybrid Cycle Detection (HCD).

4.3.1 HVN and HU

Both HVN and HU, as well as the earlier Off-line Variable Substitution (OVS) [28], are intended to simplify the original constraints before the constraint solving begins. All three of the techniques perform the following two steps:

1. Identify top-level pointers whose points-to sets are known to be equal.
2. Merge the nodes representing these top-level pointers and update all the subset constraints to refer to the newly merged node.

The techniques differ in how aggressive they are in Step 1: each technique finds possibly different sets of pointers that are provably equal.

All three techniques are based on the idea of a *subset graph* [28] (also called the *pointer assignment graph* [25] and the *offline constraint graph* [16]). Nodes in the graph are top-level pointers p, addresses $\&a$, and dereferences $*q$, and edges in the graph correspond to the ADDROF, COPY, and LOAD statements from Figure 1. For each node of the subset graph, we can define a subset-graph points-to set pt_{SG} as follows: $pt_{SG}(p) = pt(p)$, $pt_{SG}(\&a) = \{a\}$, $pt_{SG}(*q) = \bigcup_{a \in pt(q)} pt(a)$. All three techniques depend only on the following property of the subset graph.

Proposition 3. *For every top-level pointer p, $pt(p)$ is the union of $pt_{SG}(\alpha)$, where α ranges over all predecessors of p in the subset graph.*

The techniques merge pointers appearing in a common cycle in the subset graph and pointers having similar ancestors in the subset graph. The three techniques differ in their precise definition of similar ancestors and the method used to compute them. HU finds more pointer equivalences than HVN, but HU takes cubic time while HVN takes only quadratic time. Therefore, LLVM uses the more aggressive HU, but precedes it with a pass of HVN to reduce the size of the input to HU and therefore its cost.

Thanks to the use of SSA form as described in Section 3.2, the strong update algorithm models top-level pointers flow-insensitively. Therefore, the strong update analysis can merge top-level pointers (Step 2 above) in the same way as the flow-insensitive algorithm. It is only Step 1 that must be adapted to the strong update algorithm, since the increased precision may cause different pointers to have equal points-to sets than in a flow-insensitive analysis.

The only change needed is to separate dereference nodes by program point label (i.e. change nodes of the form $*q$ to the form $\ell : *q$), since in the strong update algorithm, the result of a load from $*q$ depends on the location ℓ of the load. The definition of pt_{SG} is changed to include the label ℓ as follows: $pt_{SG}(\ell : *q) = \bigcup_{a \in pt(q)} ptsu[\ell](a)$. This updated definition satisfies Proposition 3, so OVS, HVN, and HU can be safely applied without further modifications.

4.3.2 HCD

Unlike OVS, HVN, and HU, the HCD technique is intended to find subset constraint cycles that do not arise until the analysis begins to solve the constraints. Suppose there is a path $*p \to q \to \ldots \to r$ in the subset graph and a store $*p = r$. Before constraint solving, HCD creates a data structure compactly recording all such paths. During constraint solving, when some address a is added to $pt(p)$, the following constraint cycle is created: $pt(a) \subseteq pt(q) \subseteq \ldots \subseteq pt(r) \subseteq pt(a)$. When this happens, the top-level pointers on the path from q to r can be merged. HCD also "merges" the address node a with the top-level pointers q to r, but we must be careful in interpreting the meaning of "merging" an address a and a top-level pointer p. HCD merges *only* the representation of the points-to sets $pt(p)$ and $pt(a)$. It does *not* merge the notion of the address a; ADDROF constraints involving $\&a$ are not changed. If two addresses a and b are both merged with p, although $pt(a)$ is known to equal $pt(b)$, the addresses a and b are still distinct, and membership of a in a given points-to set is still independent of the membership of b in that points-to set. When merging a top-level pointer p with an address a, by convention, we can always choose the top-level pointer p as the representative of the merged node. This has the advantage that none of the instructions from Figure 1 involving p need to be rewritten with a, and preserves the property that all instructions involve only top-level pointers (except the right-hand side of ADDROF, of course).

The strong update analysis analogue of an HCD path is a path $\ell : *p \to q \to \ldots \to r$ and a store $\ell : *p = r$ at the same label ℓ. When a is added to $pt(p)$ during constraint solving, the following constraint cycle is created: $ptsu[\ell](a) \subseteq pt(q) \subseteq \ldots \subseteq pt(r) \subseteq ptsu[\ell](a)$. Thus the top-level pointers q to r can be merged. The flow-insensitive HCD also merges $pt(a)$ with $pt(q)$; the analogous merge of $su[\ell](a)$ and $pt(q)$ cannot be done in the strong update analysis because $su[\ell](a)$ is not a points-to set. However, such a merge is also unnecessary because unlike $pt(a)$ in the flow-insensitive analysis, which is of size $\Theta(|\mathcal{A}|)$, $su[\ell](a)$ has a constant size of only a few bytes.

5. Implementation

We have implemented the strong update algorithm by extending the existing flow-insensitive subset-based points-to analysis that is included in the LLVM compiler infrastructure [24], version 2.6. In the rest of this paper, we call this base analysis "flow-insensitive", even though some flow sensitivity is achieved by analyzing an intermediate representation in SSA form as discussed in Section 3.2. The base analysis is an implementation of the flow-insensitive algorithm of Figure 8 using sparse bit vectors to represent points-to sets, and with extensions for the constraint optimizations discussed in the previous section. Thus it was straightforward to extend the implementation to implement the strong update analysis algorithm, with only a few issues that we will explain in this section. Our implementation is publicly available at `http://plg.uwaterloo.ca/~olhotak/su`.

An implementation detail that is important for soundness is identifying which address-taken variables are completely overwritten by strong updates (i.e. what the *singletons* set should be). First, we strongly update only variables that are the same size as a pointer, because, for example, a store to an array of multiple pointers would only update one element of the array, so the analysis should not strongly update the whole array. Second, we strongly update only global variables and local variables of procedures that are not recursive (either directly or mutually through other procedures). A local variable of a recursive procedure can have many instances on the stack at the same time, and a store only updates one of those instances, so a strong update would be unsound. Recursive procedures are found by detecting cycles in the call graph built ahead of time by LLVM, which conservatively assumes that a procedure pointer could point to any procedure whose address has been taken. Finally, we never apply strong updates to dynamically allocated variables, since multiple instances of them can be created by repeating the allocation.

To test the correctness of the implementation, we enabled the LLVM transformations that take advantage of points-to information and used the analysis to compile the SPEC CINT 2000 and SPEC CPU 2006 benchmarks [33] that are written in C. The SPEC harness validated that all of the compiled benchmarks generated the correct output. On these benchmarks, with these test inputs, and for these LLVM transformations using the analysis results, our implementation of the strong update analysis is sound.

To compare the results of the strong update analysis with fully flow-sensitive analysis results, we also expressed the flow-sensitive analysis constraints of Figure 2 in Datalog. We used the LogicBlox Datalog implementation and applied the manual Datalog optimization techniques suggested by Bravenboer and Smaragdakis [3]. We extracted the input to the Datalog program from the LLVM points-to analysis to ensure that both analysis implementations were using the same input constraints. This setup enabled us to compare the strong update analysis results with fully flow-sensitive results.

6. Empirical Evaluation

We compared the strong update analysis with the original flow-insensitive points-to analysis by running both of them on the C benchmarks from the SPECINT 2000 and the SPECCPU 2006 suites [33]. The first column of Table 1 gives the name of the benchmarks, and the following four columns give various measurements of the size of each benchmark. Column 2 shows the number of lines of source code. The next three columns show the number of top-level pointers, the number of address-taken pointer targets, and the number of labels in the sparse labelling that was defined in Section 3.4. These counts are taken after applying the HVN and HU constraint simplifications discussed in Section 4.3.

The next three columns show the running times of the flow-insensitive analysis and the strong update analysis, and the relative time difference between them. The times shown are means of ten runs. The time differences have a geometric mean of a 5% slowdown and range from a speedup of 9% to a slowdown of 22%. The 9% speedup on 400.perlbench is due to the smaller points-to sets that need to be propagated as a result of the increased precision of the strong update analysis. These results confirm that the speed of the strong update analysis is comparable to that of the flow-insensitive analysis not only in theory, but also in practice.

For information only, the next two columns show the running times of two flow-sensitive analysis implementations that are not directly comparable with LLVM's built-in flow-insensitive analysis implementation and our strong update adaptation. The FS column shows the running time of the Datalog/LogicBlox implementation of the fully flow-sensitive analysis. This analysis runs on the same input constraints as the analyses in the FI and SU columns, and therefore produces comparable output (except for the added precision from flow sensitivity). However, this implementation is written in Datalog, whereas the LLVM implementation is written in C. On four of the benchmarks, the fully flow-sensitive analysis did not complete even after running for 48 hours. The HL [17] column shows the running time of Hardekopf and Lin's SSO semi-sparse analysis [17]. Ben Hardekopf provided an implementation and helped us make it run on the SPEC benchmarks. In theory, the HL analysis should compute the same output as the FS analysis, satisfying the constraints from Figure 2. Like the LLVM implementations of FI and SU, the HL analysis is written in C, but there are important differences that affect performance and make it impossible to directly compare the analysis output:

1. The LLVM implementation contains code that defines the sets of top-level variables ($\mathcal{P}$) and possible pointer targets ($\mathcal{A}$) and extracts ADDROF, COPY, STORE, and LOAD constraints out of the intermediate representation. The same extracted constraints are used as input to the FI, SU, and FS implementations. The HL implementation does not use this constraint generation code, but implements its own definition of $\mathcal{P}$ and $\mathcal{A}$ and its own extraction of an analogous but different set of constraints. Thus the inputs to the analysis are different, and the intermediate and final points-to sets are different in cases where top-level variables and pointer targets are modelled differently in the input constraints.
2. The LLVM implementation of FI and our adaptation SU use sparse bit vectors to represent points-to sets. The HL implementation uses binary decision diagrams (BDDs).
3. The SU implementation is built on top of the LLVM 2.6 version of the FI analysis. The HL implementation is built on top of LLVM 2.5.

The final column shows the space usage of the strong update analysis in terms of the number of tuples $\langle \ell, a, b, p \rangle$ such that $su[\ell](a) = \langle \{b\}, p \rangle$. This metric was chosen because it is proportional to the size of the *su* sets, the only data structure of significant size added in the transformation of the flow-insensitive analysis to the strong update analysis. This data structure is an array of maps, one for each label ℓ. The last column in the table shows the total number of entries in these maps. Each such entry records three 32-bit numbers representing a, b, and p. These maps could be implemented using different data structures. Even assuming a 2x space overhead for the map data structure, for the largest benchmark (403.gcc), the strong update analysis uses at most 6.1 MB more memory (268375 entries times 24 bytes) than the flow-insensitive analysis. The overall memory usage is lowered by the reduction in points-to set sizes due to the increased precision of the strong update analysis.

Table 2 compares the strong update analysis with the fully flow-sensitive analysis. The three columns under Stores – Strong Updates measure the number of store instructions in the program at which a strong update can be performed (i.e. the points-to set of the dereferenced pointer contains only one target, and this target is in the *singletons* set). As discussed in Section 3.1, strong updates are the main benefit of a flow-sensitive analysis compared to a flow-insensitive analysis. The FS column counts the number of stores at which the flow-sensitive analysis performs a strong update. In theory, the strong update analysis can perform a strong update at some subset of these stores; the size of this subset is shown in the SU column. In total (excluding the four benchmarks on which the flow-sensitive analysis does not complete), the strong update analysis performs 98% of the strong updates that the flow-sensitive analysis performs.

The right section of the table presents counts of loads in each benchmark. Load instructions are where any difference between analyses is observed because they are the only instructions in the LLVM IR in which address-taken variables are read. Every other instruction works directly only with top-level variables; if an address-taken variable is to be used, it must first be loaded into a top-level variable using a load instruction. The FS column in the table counts the number of loads $\ell : p = *q$ at which the flow-sensitive points-to set of $*q$ is strictly smaller than the flow-insensitive points-to set of $*q$ (i.e. $\cup_{a \in pt(q)} pt[\ell](a) \subsetneq \cup_{a \in pt(q), \ell' \in \mathcal{L}} pt[\ell'](a)$). At these loads, a smaller (i.e. more precise) set is propagated to p than would be if the analysis were flow-insensitive. The SU column presents the analogous counts for the strong update analysis (i.e. $\cup_{a \in pt(q)} ptsu[\ell](a) \subsetneq \cup_{a \in pt(q)} pt(a)$). In total, of all the loads at which the fully flow-sensitive propagates more precise sets than a flow-insensitive analysis would, on 98% of them the strong update analysis does too.

The benefit that a given client analysis derives from flow sensitivity in the points-to analysis needs to be evaluated separately for each proposed client. Our study has shown that the strong update analysis improves points-to precision at 98% of program points at which a flow sensitive analysis does. Thus we conclude that if a client analysis benefits from flow sensitivity in the points-to analysis, then it will similarly benefit from the strong update analysis.

Benchmark	kSLOC	$\|\mathcal{P}\|$	$\|\mathcal{A}\|$	$\|\mathcal{L}\|$	Analysis Time (s) FI	SU	slowdown	FS	HL [17]	Space SU
164.gzip	8.6	1740	971	2818	0.13	0.14	3%	7	0.50	1347
175.vpr	17.8	7241	2896	7025	0.51	0.54	6%	46	1.02	7552
176.gcc	230.5	104663	24518	117121	41.86	45.79	9%	—	97.72	142152
181.mcf	2.5	1092	262	821	0.08	0.08	6%	3	0.42	1745
186.crafty	21.2	4145	2345	10671	0.40	0.41	2%	45	0.76	1037
197.parser	11.4	6741	2442	8509	0.76	0.92	21%	1206	2.39	9262
253.perlbmk	87.1	45803	11544	49584	13.75	16.74	22%	—	52.88	37284
254.gap	71.5	53285	11560	45431	8.81	9.47	8%	7029	11.47	55112
255.vortex	67.3	34531	14305	30759	4.39	4.52	3%	1665	7.68	17802
256.bzip2	4.7	951	577	1590	0.08	0.09	1%	3	0.46	325
300.twolf	20.5	13423	3255	11650	1.20	1.25	4%	100	1.88	12004
400.perlbench	169.9	89661	21441	93792	66.99	60.71	-9%	—	306.70	70229
401.bzip2	8.3	3265	915	3243	0.28	0.30	9%	10	0.67	4161
403.gcc	521.1	240239	55012	272420	190.17	213.01	12%	—	3526.02	268375
429.mcf	2.7	1096	260	823	0.08	0.09	12%	2	0.44	1744
433.milc	15.0	5269	2343	5954	0.43	0.45	5%	30	1.04	4260
445.gobmk	197.2	54022	43881	41769	23.42	23.33	0%	7223	41.80	12391
456.hmmer	36.0	20240	4982	17186	2.10	2.22	6%	229	3.11	19713
458.sjeng	13.9	2591	1551	6544	0.26	0.27	2%	20	0.62	1687
462.libquantum	4.4	1742	912	1652	0.14	0.14	4%	5	0.49	534
464.h264ref	51.6	26884	6817	22951	3.35	3.41	2%	547	3.28	18685
470.lbm	1.2	337	150	322	0.05	0.05	-4%	2	0.42	228
482.sphinx3	25.1	12410	4013	10332	1.01	1.06	5%	115	2.51	11778

Table 1. Benchmark characteristics, analysis running times and space requirements of strong update and flow-insensitive analysis

Benchmark	Stores Total	Strong Updates SU	FS	%	Loads Total	More Precise SU	FS	%
164.gzip	246	235	235	100%	564	12	12	100%
175.vpr	916	802	802	100%	3757	16	16	100%
176.gcc	26776	23061	—	—	88050	690	—	—
181.mcf	304	204	207	99%	780	3	3	100%
186.crafty	509	405	423	96%	1493	13	13	100%
197.parser	2024	1355	1584	86%	5147	6	6	100%
253.perlbmk	14926	9175	—	—	41091	54	—	—
254.gap	12182	10060	10067	100%	39089	358	361	99%
255.vortex	4511	3786	3942	96%	17815	48	48	100%
256.bzip2	30	29	29	100%	270	0	0	100%
300.twolf	1829	1446	1446	100%	9718	0	0	100%
400.perlbench	25196	16899	—	—	79543	105	—	—
401.bzip2	316	220	237	93%	2139	22	22	100%
403.gcc	62134	40215	—	—	204112	149	—	—
429.mcf	300	199	202	99%	784	3	3	100%
433.milc	944	893	893	100%	2360	0	0	100%
445.gobmk	2206	1931	1955	99%	7043	3	4	75%
456.hmmer	2880	2216	2267	98%	13925	142	152	93%
458.sjeng	120	114	115	99%	677	0	0	100%
462.libquantum	171	140	147	95%	736	0	0	100%
464.h264ref	2143	1778	1844	96%	18944	354	363	98%
470.lbm	53	45	45	100%	134	0	0	100%
482.sphinx3	1906	1542	1549	100%	7471	17	17	100%

Table 2. Comparison of strong update and flow-sensitive analysis

7. Related Work

The study of flow-sensitive pointer analyses has a long history. Choi et al. [6] presented an early flow-sensitive alias pair analysis as an instantiation of the standard dataflow analysis framework [23]. The analysis was applied on a Sparse Evaluation Graph [5]; that is, a control flow graph with irrelevant nodes removed. In order to improve efficiency further, Choi et al. [7] devised one of the first extensions of SSA form [9] to represent indirect writes through pointers. Their Factored SSA (FSSA) form allowed "preserving" definitions, analogous to weak updates that may or may not overwrite the value of a variable.

Chow et al. [8] proposed a different extension of SSA form for handling pointers, Hashed SSA (HSSA) form. This intermediate representation added two new kinds of nodes. A χ node was placed after every store to indicate that address-taken variables may or may not have been updated (similar to a preserving definition). A μ node was used to indicate a possible use of an address-taken variable by a load.

Emami et al. [11] defined a points-to analysis that was not only flow-sensitive but also context-sensitive. Each points-to relationship was annotated as either possible or definite to enable strong updates. Like earlier analyses, the analysis was implemented as a dataflow analysis on the control flow graph.

Wilson and Lam [37] presented a context-sensitive pointer analysis based on partial transfer functions (PTF) summarizing the effects of procedures. Each PTF was constructed using a flow-sensitive analysis, which was efficient because it was intra-procedural. The PTFs were then combined to obtain context-sensitive interprocedural results. They presented experimental results on programs of up to 5 kLOC. This work sparked a line of similar points-to analyses that were flow-sensitive intra-procedurally and generated procedure summaries that could be instantiated at call sites [29, 35, 36]. However, these analyses performed strong updates only on top-level variables.

Hasti and Horwitz [18] proposed a technique that iteratively builds SSA form for variables with known aliasing, then performs alias analysis to increase the set of variables for which aliasing is known. It remains an open question whether the fixed point of this technique matches the results of a fully flow-sensitive alias analysis.

Hind and Pioli [20, 21] performed an empirical study of the benefits of flow sensitivity in alias analysis as well as of techniques to improve its performance. They found that flow sensitivity improves precision for a small subset of pointers.

Goyal [13] derived a flow-sensitive points-to analysis algorithm that uses a fine-grained worklist to run in asymptotically cubic time and cubic space in the worst case. The worst-case cubic bounds are likely to be achieved on typical programs because the algorithm explicitly generates a full points-to graph for each program point, and points-to graphs are typically quadratic in size. In contrast, the strong update algorithm is similar in both structure and empirical behaviour to flow-insensitive analysis, which has been shown to run in quadratic time in practice [30]. We are not aware of any implementation or empirical evaluation of Goyal's algorithm, but Staiger-Stöhr [31, 32] designed and implemented a similar algorithm with the same asymptotic complexity.

Zhu and Calman [38] took initial steps towards using Binary Decision Diagrams (BDDs) [4] to efficiently represent flow-sensitive points-to sets.

Tok et al. [34] presented a technique to speed up flow-sensitive dataflow analysis on a control flow graph using computed def-use chains for address-taken variables. As the analysis discovers new def-use chains, the chains are used to reorder the instructions in the worklist to reduce the analysis time.

Hardekopf and Lin [17] presented a semi-sparse algorithm to improve the running time of a fully flow-sensitive subset-based points-to analysis. The analysis was sparse in that it did not process control flow graph nodes as a whole, but instead followed def-use chains to directly find the stores that produce the values for each load. Because def-use chains for address-taken variables are not known until the analysis completes, the analysis was semi-sparse in that it was sparse only on top-level variables. The analysis used BDDs to keep the memory requirements of full flow sensitivity manageable. It was the first fully flow-sensitive subset-based points-to analysis that scaled to benchmarks of hundreds of kLOC.

8. Conclusion

We have presented a subset-based points-to analysis algorithm that combines the key advantages of flow-insensitive and flow-sensitive analyses. Like a flow-sensitive analysis, the algorithm enables strong updates, which are the main precision benefit of flow sensitivity. Like a flow-insensitive analysis, the strong update algorithm requires, in the worst case, quadratic space and cubic time. We have shown that its running time in practice is comparable to that of the flow-insensitive analysis. We have also shown that the strong update analysis performs 98% of the strong updates of a fully flow-sensitive analysis, and propagates more precise points-to sets at 98% of the loads at which a fully flow-sensitive analysis does. These benefits of the algorithm stem from the notion that it is the precise points-to sets that enable strong updates (and therefore further precision), yet it is also these sets that can be manipulated efficiently. Thus the strong update algorithm focuses attention on these sets to gain the precision benefits of flow sensitivity and the efficiency benefits of flow insensitivity.

Acknowledgements

We thank Ben Hardekopf for his assistance with the semi-sparse analysis of [17]. We are grateful to the anonymous POPL reviewers for their particularly constructive suggestions that helped improve this paper. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[3] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 243–262, 2009.

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[5] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 1991.

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

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

[8] F. Chow, S. Chan, S.-M. Liu, and R. Lo. Effective representation of aliases and indirect memory operations in SSA form. In *Compiler Construction: 6th International Conference, CC'96*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267, 1996.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[10] B. A. Davey and H. A. Priestly. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, first edition, 1990.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[12] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[13] D. Goyal. Transformational derivation of an improved alias analysis algorithm. *Higher Order Symbol. Comput.*, 18(1-2):15–49, 2005.

[14] B. Hardekopf. *Pointer Analysis: Building a Foundation for Effective Program Analysis*. PhD thesis, University of Texas at Austin, May 2009.

[15] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–299, 2007.

[16] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 265–280, 2007.

[17] B. Hardekopf, and C. Lin,. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, 2009.

[18] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[19] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[20] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static analysis: 5th International Symposium, SAS '98*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, 1998.

[21] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[22] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–259, 2008.

[23] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[24] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

[25] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Apr. 2003.

[26] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, 2004.

[27] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 126–135, 2009.

[28] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 47–56, 2000.

[29] A. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, Massachusetts Institute of Technology, Sept. 2006.

[30] M. Sridharan and S. J. Fink. The complexity of andersen's analysis in practice. In J. Palsberg and Z. Su, editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pages 205–221, 2009.

[31] S. Staiger-Stöhr. Implementing sparse flow-sensitive andersen analysis. Technical Report 2009/03, Universität Stuttgart, 2009.

[32] S. Staiger-Stöhr. *Kombinierte statische Ermittlung von Zeigerzielen, Kontroll- und Datenfluss*. PhD thesis, Universität Stuttgart, 2009.

[33] Standard Performance Evaluation Corporation. URL `http://www.spec.org/`.

[34] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference, CC 2006*, volume 3923 of *Lecture Notes in Computer Science*, pages 17–31, 2006.

[35] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

[36] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.

[37] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[38] J. Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC '05: Proceedings of the 42nd Annual Conference on Design automation*, pages 831–836, 2005.