Points-To Analysis with Efficient Strong Updates[1]

Ondřej Lhoták   Kwok-Chiang Andrew Chung
D. R. Cheriton School of Computer Science
University of Waterloo
{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.

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, 11, 13, 17, 23, 24], 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 [12, 20].

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. Sridharan and Fink showed that the cubic time bound is actually quadratic in typical programs [26].

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 dereferenced may-point-to set is a singleton; otherwise, the analysis could not guarantee that any one of the targets in the set is definitely overwritten. In addition, 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.

[1] Technical report number CS-2010-01. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

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.
- 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 [22].
- It experimentally evaluates the implementation on the SPECINT 2000 and SPECCPU 2006 benchmark suites [27], shows that its practical performance is comparable to that of the flow-insensitive analysis, and presents data on the precision benefits of strong updates and flow sensitivity.

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 a 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 ben-

$p = \&a$	$\{a\} \subseteq pt(p)$	[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

efits to precision. The results show that the performance of the strong update analysis is comparable to that of the flow-insensitive analysis. 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 notation that will be used in 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 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.

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 set 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 [9].

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

$\ell : p = \&a$	$\{a\} \subseteq pt[{}_\bullet\ell](p)$	[AddrOf]
$\ell : p = q$	$pt[{}^\bullet\ell](q) \subseteq pt[{}_\bullet\ell](p)$	[Copy]
$\ell : *p = q$	$\forall a \in pt[{}^\bullet\ell](p) \,.\, pt[{}^\bullet\ell](q) \subseteq pt[{}_\bullet\ell](a)$	[Store]
$\ell : p = *q$	$\forall a \in pt[{}^\bullet\ell](q) \,.\, pt[{}^\bullet\ell](a) \subseteq pt[{}_\bullet\ell](p)$	[Load]
$\ell_1 \in pred(\ell_2)$	$\forall v \in \mathcal{V} \,.\, pt[{}_\bullet\ell_1](v) \subseteq pt[{}^\bullet\ell_2](v)$	[CFlow]
$\ell \in \mathcal{L}$	$\forall v \in \mathcal{V} \setminus kill(\ell) \,.\, pt[{}^\bullet\ell](v) \subseteq pt[{}_\bullet\ell](v)$	[Preserve]

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

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 ${}^\bullet\ell$ and ${}_\bullet\ell$ indicates the program points immediately before and after the instruction labelled ℓ, respectively. For example, $pt[{}_\bullet\ell](v)$ 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 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.

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

$$\mathit{kill}(\ell : p = \ldots) \triangleq \{p\}$$

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

Figure 3. Definition of kill sets

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 may have caused a to be in the points-to set of p, 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 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 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.

3. Design Overview

In this section, we present the design objectives for the points-to analysis with cheap strong updates. 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 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 $\mathit{pt}(a) = \{b, c\}$. A flow-sensitive analysis, even one without strong updates, would determine that after line 4, a does not yet point to c: $\mathit{pt}[{}_{\bullet}4](a) = \{b\}$. 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 $\mathit{pt}[\ell](a) = \{b, c\}$ 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*

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

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

the kill sets are empty, $\mathit{pt}[{}^{\bullet}\ell_1](v) = \mathit{pt}[{}^{\bullet}\ell_2](v)$ for every variable v.

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

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 find themselves 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. As a more synthetic example, the benchmarks in many benchmark suites are usually run from a test harness that executes the benchmark several times; thus the whole benchmark is inside 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). The effect of strong updates of top-level variables can be easily achieved by first converting the program to Static Single Assignment (SSA) form [8]. 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 $\mathit{pt}[{}^{\bullet}\ell_u](p) = \mathit{pt}[{}_{\bullet}\ell_d](p)$.*

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

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

Proof. Since the program is in strict SSA form, there is a path in the ICFG from ℓ_d to ℓ_u, so $pt[{}_\bullet\ell_d](p) \subseteq \cdots \subseteq pt[{}^\bullet\ell_u](p)$ (using CFLOW and PRESERVE constraints and the fact that no instruction other than ℓ_d kills p). Consider all of the subset constraints having any points-to set $pt[*](p)$ on their right-hand side, where $*$ can be any program point. Notice that all of these constraints have a points-to set of p at some label on their left-hand side (except for the constraints modelling the effect of ℓ_d). Therefore, one solution to all of these subset constraints is $\forall * \in {}^\bullet\mathcal{L} \cup {}_\bullet\mathcal{L} \,.\, pt[*](p) = pt[{}_\bullet\ell_d](p)$. Since the analysis finds the least solution, the analysis will find a solution for which $pt[{}^\bullet\ell_u](p) \subseteq pt[{}_\bullet\ell_d](p)$. Since we also showed that $pt[{}_\bullet\ell_d](p) \subseteq pt[{}^\bullet\ell_u](p)$, we conclude that $pt[{}^\bullet\ell_u](p) = pt[{}_\bullet\ell_d](p)$. $\square$

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 $O(|\mathcal{P}||\mathcal{A}| + |\mathcal{A}|^2|\mathcal{L}|)$.

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 $pt[\ell](a)$. 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 set will not directly enable strong updates, so there is little benefit in spending much space or time on it.

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.

$$\begin{tikzcd}
 & & \top \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\{a\} \arrow[drr, dash] & \{b\} \arrow[dr, dash] & \{c\} \arrow[d, dash] & \cdots \arrow[dl, dash] \\
 & & \{\} &
\end{tikzcd}$$

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

$\ell : p = \&a$	$\{a\} \subseteq pt(p)$	[ADDROF]
$\ell : p = q$	$pt(q) \subseteq pt(p)$	[COPY]
$\ell : *p = q$	$\forall a \in pt(p) \,.\, pt(q) \sqsubseteq su[{}_\bullet\ell](a)$	[STORE]
	$\forall a \in pt(p) \,.\, pt(q) \subseteq pt(a)$	
$\ell : p = *q$	$\forall a \in pt(q) \,.\, ptsu[{}^\bullet\ell](a) \subseteq pt(p)$	[LOAD]
$\ell_1 \in pred(\ell_2)$	$\forall a \in \mathcal{A} \,.\, su[{}_\bullet\ell_1](a) \sqsubseteq su[{}^\bullet\ell_2](a)$	[CFLOW]
$\ell \in \mathcal{L}$	$\forall a \in \mathcal{A} \setminus kill(\ell) \,.\, su[{}^\bullet\ell](a) \sqsubseteq su[{}_\bullet\ell](a)$	[PRESERVE]

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

Figure 7. Constraints for Strong Update Analysis

To implement this, we define the singleton-set lattice $\mathcal{S}$ as shown in Figure 6. An element of $\mathcal{S}$ is either the empty set, a singleton set, or $\top$ indicating some larger set. For each program point ℓ and pointer target a, the analysis stores an element $su[\ell](a)$ of this lattice. The analysis also stores a flow-insensitive points-to set $pt(a)$ for each pointer target a. The STORE constraint updates both $pt(a)$ and $su[\ell](a)$ if the points-to set of the variable q being stored is a singleton (and sets $su[\ell](a)$ to $\top$ if it is not). When the LOAD constraint needs the points-to set of a at ℓ, it first consults $su[\ell](a)$ for a possible singleton; 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.

Although the asymptotic complexity 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 $su[\ell]$ 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 $su[\ell](a)$ is a singleton. If $su[\ell](a) = \top$ for most values of a, we could use a hash table 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. However, there are also program points at which $su[\ell](a) = \{\}$ for all values of a, namely the program points following a store through a pointer p whose points-to set is empty. For these program points, this representation would require a hash table with $|\mathcal{A}|$ entries. Fortunately, the analysis never encounters a case in which an incoming *su* value is all $\{\}$ and only some of the keys have their values changed; that is, at no point in the analysis do we need to represent an $su[\ell](a)$ function that is $\{\}$ for most but not all values of a. Therefore, in our implementation, we use a hybrid data structure to represent $su[\ell]$. A boolean flag dictates whether $su[\ell](a)$ is $\{\}$ for all a or not. When this flag is false, a hash map then stores the values of $su[\ell](a)$ other than $\top$, and $\top$ is returned when a pointer target is not found in the hash map. When the boolean flag is true, the hash map is unnecessary and is simply ignored. This hybrid representation is compact in all of the common use cases.

3.4 Sparse Allocation of Labels

The analysis can be simplified and made more efficient by removing redundant labels from the program representation. In our discussion thus far, every instruction was assigned its own unique label ℓ. But most instructions do not change the points-to sets of address taken variables $a \in \mathcal{A}$. That is, most instructions do not affect the flow-sensitive *su* sets. The *su* value after a given instruction is equal to the *su* value after the immediate control flow predecessor of the instruction except when the instruction is a STORE, when the instruction has multiple control flow predecessors (i.e. it is a control flow merge), or when the instruction has no control flow predecessors because it is the very first instruction in the program.

```
1  foreach ADDROF constraint $p = \&a$ do $pt(p) \cup= \{a\}$; $worklist \cup= \{p\}$ od
2  foreach COPY constraint $p = q$ do $graph \cup= \{q \rightarrow p\}$ od
3  while $worklist \neq \{\}$ do
4     remove a variable $v$ from $worklist$
5     $\Delta \leftarrow pt(v) \setminus oldpt(v)$
6     $oldpt(v) \leftarrow pt(v)$
7     foreach STORE constraint $*v = q$ do foreach $a \in \Delta$ do AddEdge($q$, $a$) od od
8     foreach LOAD constraint $p = *v$ do ProcessLoad($p$, $\Delta$) od
9     foreach $v \rightarrow w \in graph$ do
10       $pt(w) \cup= \Delta$
11       if $pt(w)$ changed then $worklist \cup= \{w\}$ fi
12    od
13 od
14 proc ProcessLoad($p$, $\Delta$)
15    foreach $a \in \Delta$ do AddEdge($a$, $p$) od
16 endproc
17 proc AddEdge($v$, $w$)
18    if $v \rightarrow w \notin graph$ then $graph \cup= \{v \rightarrow w\}$; $pt(w) \cup= pt(v)$; if $pt(w)$ changed then $worklist \cup= \{w\}$ fi fi
19 endproc
```

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

When it is certain that the *su* sets at one instruction are identical to those at its predecessor, we can 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 (i.e. a store, a merge, or a clear).

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, *graph*, maintains a set of edges corresponding to the subset constraints being solved. The presence of the edge $v \rightarrow w$ 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 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 $v \rightarrow w$ 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

```
1    foreach AddrOf constraint $p = \&a$ do $pt(p) \cup= \{a\}$;  $worklist \cup= \{p\}$ od
2    foreach Copy constraint $p = q$ do $graph \cup= \{q \to p\}$ od
3    while $worklist \neq \{\}$ do
4      remove a variable $v$ or a label $\ell$ from $worklist$
5      if a variable $v$ was removed then
6        $\Delta \leftarrow pt(v) \setminus oldpt(v)$
7        $oldpt(v) \leftarrow pt(v)$
8*       foreach Store constraint $\ell : *v = q$ do $worklist \cup= \{\ell\}$ od
9*       $worklist \cup= affected[v]$
10       foreach Store constraint $\ell : *v = q$ do foreach $a \in \Delta$ do AddEdge($q$, $a$) od od
11       foreach Load constraint $\ell : p = *v$ do ProcessLoad($\ell$, $p$, $\Delta$) od
12       foreach $v \to w \in graph$ do
13         $pt(w) \cup= \Delta$
14         if $pt(w)$ changed then $worklist \cup= \{w\}$ fi
15       od
16*    else // a label $\ell$ was removed
17*      if $\ell$ is a clear then $su[\ell] \leftarrow \lambda a.\top$
18*      else if $\ell$ is a merge then $su[\ell] \leftarrow \bigsqcup_{\ell' \in pred(\ell)} su[\ell']$
19*      else // $\ell$ is a store $*p = q$
20*        if $pt(p) = \{\}$ then continue fi
21*        $su[\ell] \leftarrow su[pred(\ell)]$
22*        if $|pt(q)| \leq 1$ then $affected[q] \cup= \{\ell\}$ else $affected[q] \setminus= \{\ell\}$ fi
23*        if $pt(p) = \{a\}$ and $a \in singletons$
24*          then $su[\ell](a) \leftarrow$ PtToSu($q$)  // strong update
25*          else foreach $a \in pt(p)$ do $su[\ell](a) \sqcup=$ PtToSu($q$) od fi // weak update
26*      fi
27*      if $su[\ell]$ changed then
28*        $worklist \cup= succ(\ell)$
29*        foreach Load constraint $\ell : p = *q$ do ProcessLoad($\ell$, $p$, $pt(q)$) od
30*      fi
31   fi od
32   proc ProcessLoad($\ell$, $p$, $\Delta$)
33     foreach $a \in \Delta$ do
34*      if $su[\ell](a) = \top$
35         then AddEdge($a$, $p$)
36*        else $pt(p) \cup= su[\ell](a)$;  if $pt(p)$ changed then $worklist \cup= \{p\}$ fi fi
37   od endproc
38   proc AddEdge($v$, $w$)
39     if $v \to w \notin graph$ then $graph \cup= \{v \to w\}$;  $pt(w) \cup= pt(v)$;  if $pt(w)$ changed then $worklist \cup= \{w\}$ fi fi
40   endproc
41*  proc PtToSu($q$)
42*    if $|pt(q)| \leq 1$ and $pt(q) \subseteq singletons$ then return $pt(q)$ else return $\top$ fi
43*  endproc
```

Figure 9. Strong Update Points-to Analysis Algorithm

and eventually established. 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 $su[\bullet\ell]$ 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 in $pt(p)$ 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 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 $a \to p$ 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 maintain 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 $a \to p$ 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 it 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 $O(E|\mathcal{A}|)$. □

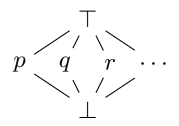

Figure 10. The top-level variable equivalence lattice

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 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 $O(|\mathcal{V}|^2|\mathcal{A}|)$.

The total time spent in each section of the algorithm is in $O(E|\mathcal{V}^2|)$, so the algorithm completes in $O(E|\mathcal{V}^2|)$ time. □

4.2 A minor improvement

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 $pt(a)$ 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.

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 [22], version 2.6. The base analysis is an implementation of the flow-insensitive algorithm of Figure 8, so extending it to implement the strong update analysis algorithm was straightforward, with only a few issues that we will explain in this section.

Before and during points-to set propagation, the existing LLVM points-to analysis simplifies the points-to constraints using Hybrid Cycle Detection [13] and Pointer Equivalence [14]. These transformations reduce the number of subset constraints by merging variables whose points-to sets are provably equal and eliminating the redundant constraints. That is, they reduce the number of constraints while guaranteeing the same analysis output. However, this guarantee assumes a fully flow-insensitive analysis; applying the same transformations when instructions are related by control flow would be incorrect because it would change the output of the analysis. Therefore, we have disabled these transformations in the implementation. We conjecture that similar simplifying transformations that do correctly take control flow information into account could be devised, but we leave this to future work.

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. Finally, we never apply strong updates to dynamically allocated variables, since multiple instances of them can be created by repeating the allocation.

Another important implementation detail is the handling of indirect function calls and of calls to external code that is unavailable for analysis. We use a simple approach to model these calls soundly: before and after any indirect or external call, we insert a "clear" node like the one at the beginning of the program, which sets $su[\ell]$ to $\lambda a.\top$; i.e. it discards all flow-sensitive information. If such calls are rare, the reduction in flow-sensitive precision is small (and even in the worst case, this is no less precise than the original flow-insensitive analysis). If more precision is needed, the analysis algorithm can be extended to use the computed points-to sets to determine targets of indirect calls, as proposed by Emami et al. [10], at the cost of increasing the asymptotic running time. The existing LLVM implementation contains simulations of the effect of common C standard library functions, and we reuse these (flow-insensitive) simulations in the strong update analysis.

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 [27] that are written in C, except for 400.perlbench and 403.gcc, which will be discussed in the next section. 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.

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 [27]. 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 defined in Section 3.4.

The next two columns show the running times of the flow-insensitive analysis and the strong update analysis. The evaluation was done on a machine with an AMD Phenom X4 9100e processor (4 cores, 1.8 GHz, .5/2/2 MB L1/2/3 cache) and 4GB RAM running Linux 2.6.28, using one of the cores. The times shown are means of ten runs. On two of the SPECCPU 2006 benchmarks, 400.perlbench and 403.gcc, neither the original flow-insensitive nor the strong update analysis completed within two hours, preventing further observations. In most of the benchmarks, most of the points-to sets remain small, but these two benchmarks give rise to very large points-to sets that spread to many pointers, leading to very slow propagation. This pathological behaviour is a characteristic of the benchmarks themselves, rather than purely their size: on several benchmarks larger than 400.perlbench, both analyses completed in reasonable time. Also, 253.perlbmk (an earlier version of 400.perlbench) has very similar size characteristics as 254.gap, yet it takes hundreds of times longer to analyze.

Relative to the flow-insensitive analysis, the strong update analysis performs consistently very well. On average (geometric mean of ratios), the strong update analysis takes only 10% longer than the flow-insensitive analysis. The relative slowdowns range from 0% to 23%, except for 55% on 176.gcc. Thus, the performance of the strong update analysis is comparable with that of the flow-insensitive analysis not only in theory, but also in practice.

The next two columns of the table present counts of stores. The first column counts the number of stores at which a strong update can be performed (i.e. the if statement in Line 23 of the algorithm in Figure 9 succeeds), while the second column counts the total number of stores. On average (geometric mean), strong updates can be performed at 81% of the stores, and this percentage ranges from 61% to 97% among the benchmarks. This high proportion is largely due to the LLVM technique of separating top-level and address-taken variables described in Section 2. For every address-taken variable a in the original code, LLVM creates a top-level pointer p pointing only to a, and all writes to a are transformed into stores through p. Since the points-to set of p is a singleton, the strong update analysis can perform strong updates on a as if it were a top-level variable whose address had not been taken (except, of course, at program points where a actually is modified through other pointers). In contrast, the original flow-insensitive analysis is forced to model all such accesses to address-taken variables imprecisely as weak updates.

The final three columns of the table present counts of loads in each benchmark. Load instructions are where the difference between the two 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 variables using a load instruction. The right-most column counts the total number of loads. Of those, the middle column counts the number of loads $\ell : p = *q$ at which $su[\ell](a)$ is not $\top$ for at least one $a \in pt(q)$. In other words, this column counts the number of loads at which some flow-sensitive information relevant to the load is available. The average (geometric mean) is 20%, but there is much benchmark-

					Analysis Time (s)		Stores		Loads		
Benchmark	kSLOC	$\|\mathcal{P}\|$	$\|\mathcal{A}\|$	$\|\mathcal{L}\|$	FI	SU	SU	total	more precise	non-$\top$	total
164.gzip	8.6	2708	1237	2818	0.14	0.17	235	246	23	232	618
175.vpr	17.8	12252	3625	7025	0.38	0.40	802	916	53	651	4419
176.gcc	230.5	258452	30822	117121	37.20	57.54	23062	26546	653	19441	91078
181.mcf	2.5	1988	376	821	0.07	0.07	204	304	9	279	967
186.crafty	21.2	8091	2609	10671	0.40	0.42	405	509	22	224	1774
197.parser	11.4	12227	3188	8509	0.65	0.77	1355	2024	5	666	5385
253.perlbmk	87.1	89678	14348	49584	1515.91	1533.13	9175	14925	424	7464	45230
254.gap	71.5	82980	14175	45431	2.00	2.46	10060	12179	478	10736	40984
255.vortex	67.3	49150	17745	30759	1.63	1.80	3786	4511	219	2836	20490
256.bzip2	4.7	1632	707	1590	0.11	0.11	29	30	0	33	339
300.twolf	20.5	28720	3607	11650	0.65	0.69	1446	1829	4	2369	12773
400.perlbench	169.9	178811	26509	93793	>7200	>7200					
401.bzip2	8.3	9825	1191	3243	0.26	0.29	220	301	49	1024	3987
403.gcc	521.1	567212	71888	272420	>7200	>7200					
429.mcf	2.7	2029	375	823	0.08	0.09	199	300	6	289	987
433.milc	15.0	13661	3041	5954	0.36	0.37	893	944	2	1159	3707
445.gobmk	197.2	74832	47709	41769	42.22	42.75	1931	2206	27	1592	8577
456.hmmer	36.0	37978	6739	17186	0.90	1.06	2216	2880	181	2494	17129
458.sjeng	13.9	6710	1852	6544	0.27	0.30	114	120	0	168	823
462.libquantum	4.4	4155	1176	1652	0.14	0.15	140	171	9	135	871
464.h264ref	51.6	67920	7986	22951	1.58	1.67	1778	2143	104	3143	24813
470.lbm	1.2	1309	235	322	0.06	0.06	45	53	4	414	599
482.sphinx3	25.1	20792	5220	10332	0.55	0.63	1542	1906	58	1636	8742

Table 1. Benchmark characteristics, analysis running times, and precision measurements

specific variation: the proportions range from 10% to 69%. The flow-sensitive information is $\top$ when the value of the address-taken variable read in the load differs depending on the control flow path taken from the last store of the variable to the load. Thus benchmarks that write through pointers within complicated control flow structures and benchmarks that read through pointers far away from the write tend to have a low proportion of non-$\top$ loads.

Finally, the third-last 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 (i.e. more precise) than the flow-insensitive points-to set of $*q$ (i.e. $\cup_{a \in pt(q)} ptsu[\ell](a) \subsetneq \cup_{a \in pt(q)} pt(a)$). At these loads, a smaller set is propagated to p than would be if the analysis were fully flow-insensitive. All but two of the benchmarks contain such loads, and therefore benefit from the flow-sensitivity of the strong update analysis. In the pointer-intensive 176.gcc benchmark, 653 loads benefit. On the other hand, on average (geometric mean), the number of these loads is only 3% of the number of non-$\top$ loads. However, this is not as negative a result as it appears for two reasons. First, it indicates that in addition to the loads for which flow sensitivity makes a difference, there are many (i.e. the other 97%) loads for which even the flow-insensitive analysis generates very precise points-to sets: these sets must be singletons because they are representable in the strong update lattice. This precision of even the flow insensitive analysis stems from the many short-lived local variables whose address is taken. The LLVM IR turns accesses to them into stores and loads, and because they are short lived, even the flow-insensitive analysis analyzes them precisely. The existence of many easily-analyzed variables is an artifact of the intermediate representation and is independent of the variables for which flow sensitivity *is* beneficial. Second, this ratio is not specific to the strong update analysis; it would be similar with every other flow-sensitive analysis. The non-$\top$ loads are those for which the strong update lattice is already sufficient to encode the full flow-sensitive points-to set. In a fully flow-sensitive analysis, these points-to sets would not be any smaller, since they are already singletons. The only sets on which a fully flow-sensitive analysis would differ would be those that are so large that they are conservatively approximated as $\top$ in the strong update analysis. But by definition, they have nothing to do with the proportion of non-$\top$ sets that provide greater precision than the flow-insensitive information. In summary, the benchmarks contain hundreds of loads for which the strong update information is beneficial, though the form of the intermediate representation also generates many others for which it is not.

7. Related Work

The study of flow-sensitive pointer analyses has a long history. Choi et al. [5] presented an early flow-sensitive alias pair analysis as an instantiation of the standard dataflow analysis framework [21]. The analysis was applied on a Sparse Evaluation Graph [4]; that is, a control flow graph with irrelevant nodes removed. In order to improve efficiency further, Choi et al. [6] devised one of the first extensions of SSA form [8] 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. [7] 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.

Emami et al. [10] 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 [31] 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 [25, 29, 30]. However, these analyses performed strong updates only on top-level variables.

Hasti and Horwitz [16] 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 flow-sensitive alias analysis.

Hind and Pioli [18, 19] performed an empirical study of the benefits of flow sensitivity in alias analysis as well as of techniques to improve its performance. Like us, they found that for some pointers, flow sensitivity improves precision, but many pointers are used in such trivial ways that flow insensitivity is sufficient.

Zhu and Calman [32] took initial steps towards using Binary Decision Diagrams (BDDs) [3] to efficiently represent flow-sensitive points-to sets.

Tok et al. [28] 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 [15] 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 CFG 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 fully known until the analysis completes, the analysis was semi-sparse in that it was sparse only on top-level variables. The analysis also 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 successfully scaled to benchmarks of hundreds of kLOC.

8. Conclusion

We 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. 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 onto these sets to gain the precision benefits of flow sensitivity and the efficiency benefits of flow insensitivity.

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 *PLDI 2003*, pages 103–114, 2003.

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

[4] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL 1991*, pages 55–66, 1991.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL 1993*, pages 232–245, 1993.

[6] 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.

[7] F. Chow, S. Chan, S.-M. Liu, and R. Lo. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267, 1996.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL 1989*, pages 25–35, 1989.

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

[10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI 1994*, pages 242–256, 1994.

[11] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI 1998*, pages 85–96, 1998.

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

[13] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, pages 290–299, 2007.

[14] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS 2007*, pages 265–280, 2007.

[15] B. Hardekopf, and C. Lin,. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238, 2009.

[16] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI 1998*, pages 97–105, 1998.

[17] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *PLDI 2001*, pages 254–263, 2001.

[18] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98*, pages 57–81, 1998.

[19] M. Hind and A. Pioli. Which pointer analysis should I use? In *ISSTA 2000*, pages 113–123, 2000.

[20] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08*, pages 249–259, 2008.

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

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, page 75, 2004.

[23] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC 2003*, pages 153–169, Apr. 2003.

[24] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135, 2009.

[25] A. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, MIT, Sept. 2006.

[26] M. Sridharan and S. J. Fink. The complexity of andersen's analysis in practice. In *SAS 2009*, pages 205–221, 2009.

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

[28] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *CC 2006*, pages 17–31, 2006.

[29] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *PLDI 2001*, pages 35–46, 2001.

[30] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA 1999*, pages 187–206, 1999.

[31] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI 1995*, pages 1–12, 1995.

[32] J. Zhu. Towards scalable flow and context sensitive pointer analysis. In *DAC '05*, pages 831–836, 2005.