# Hash Consed Points-To Sets

Mohamad Barbar[1,2] and Yulei Sui[1]

[1] University of Technology Sydney, Australia
[2] CSIRO's Data61, Sydney, Australia

**Abstract.** Points-to analysis is a fundamental static analysis, on which many other analyses and optimisations are built. The goal of points-to analysis is to statically approximate the set of abstract objects that a pointer can point to at runtime. Due to the nature of static analysis, points-to analysis introduces much redundancy which can result in duplicate points-to sets and duplicate set union operations, particularly when analysing large programs precisely. To improve performance, there has been extensive effort in mitigating duplication at the algorithmic level through, for example, cycle elimination and variable substitution. Unlike previous approaches which make algorithmic changes to points-to analysis, this work aims to improve the underlying data structure, which is less studied. Inspired by hash consing from the functional programming community, this paper introduces the use of hash consed points-to sets to reduce the effects of this duplication on both space and time without any high-level algorithmic change. Hash consing can effectively handle duplicate points-to set by representing points-to sets once, and referring to such representations through references, and can speed up duplicate union operations through efficient memoisation. We have implemented and evaluated our approach using 16 real-world C/C++ programs (more than 9.5 million lines of LLVM instructions). Our results show that our approach speeds up state-of-the-art Andersen's analysis by 1.85× on average (up to 3.21×) and staged flow-sensitive analysis (SFS) by 1.69× on average (up to 2.23×). We also observe an average ≥4.93× (up to ≥15.52×) memory usage reduction for SFS.

**Keywords:** Points-to analysis · Hash consing · Memoisation.


## 1 Introduction

Points-to analysis is a fundamental static analysis used to, for example, detect memory errors [32, 53], detect concurrency bugs [9, 37], perform typestate verification [17, 51], enforce control-flow integrity [14, 15], perform symbolic execution [48, 49], and perform code embedding [10, 45]. The aim of points-to analysis

Published as: Barbar M., Sui Y. (2021) Hash Consed Points-To Sets. In: Drăgoi C., Mukherjee S., Namjoshi K. (eds) Static Analysis. SAS 2021. Lecture Notes in Computer Science, vol 12913. Springer, Cham.

The final authenticated version is available online at https://doi.org/10.1007/978-3-030-88806-0_2.

is to compute an approximation of the set of abstract objects that a pointer can refer to. Inclusion-based analysis, as the most commonly used form of points-to analysis, formulates points-to resolution as a set constraint solving problem whereby each program statement produces one or more set constraints which are translated into union operations between two points-to sets and are solved until a fixed-point is reached.

When analysing real-world programs, many pointers may yield exactly the same points-to sets during constraint resolution. This becomes more prevalent especially as analyses become more precise. For example, unlike flow-insensitive analysis which computes a single points-to set for each pointer, flow-sensitive analysis computes and maintains points-to sets at different program points, but unfortunately introduces many duplicate points-to sets. Table 1 provides the proportions of duplicate points-to sets under two popular points-to analyses (Andersen's analysis [36] and staged flow-sensitive analysis or SFS [23]) for 16 real-world programs. Columns 2 and 5 show the number of pointers maintained in the analyses. Columns 3 and 6 list the number of those pointers which refer to the 5 most common points-to sets. Columns 4 and 7 list the proportions of the pointers in Columns 3 and 6. The empty points-to set and pointers which have an empty points-to set are excluded. Both Andersen's analysis and SFS are field-sensitive inclusion-based analyses, however, SFS maintains pointers on a per program point basis to achieve flow-sensitivity, resulting in more pointers and duplicate points-to sets. We see that, on average, the 5 most common points-to sets are referred to by around 60% and 90% of pointers for Andersen's analysis and SFS, respectively. Clearly, repeatedly representing the same points-to sets is redundant, memory-wise.

Furthermore, since the resulting points-to sets of many pointers are the same, most may have reached that result with the same union operations and it is very costly to perform duplicate unions. That is, if two pointers points-to set are the same (i.e., $pt(p) = pt(q)$) by the end of the analysis, it is possible that both points-to sets were built up through the same union operations. Thus, many union operations are in fact duplicates of operations which have been previously performed. This has strong implications on performance as conducting points-to set unions produced by the set constraints forms a bulk of analysis time.

Both the number of duplicate points-to sets tracked and the number of unions performed can be reduced but most previous solutions have been analysis-specific requiring algorithmic changes, which may not be applicable to other points-to analyses. For example, either, or both, can be achieved by merging equivalent pointers offline [4, 21, 23, 22, 39] or online [20, 29, 34], selectively applying precision [30, 41], or carefully choosing how to solve constraints [35, 36]. Despite these efforts, duplication still exists and pushing the boundaries through algorithmic changes to the points-to analysis may lead to increasingly diminishing returns on performance.

In this paper, we aim to explore solutions at the data structure level – which is easily applicable to a range of points-to analyses – to reduce the influence of these duplicate operations and points-to sets on time and space. We leverage

Table 1: The number of pointers tracked, the number of those pointers which refer to one of the 5 most common points-to sets, and that proportion for Andersen's analysis and SFS.

| Program | Andersen's | | | SFS | | |
|---|---|---|---|---|---|---|
| | **Pointers** | **Top 5** | **Prop.** | **Pointers** | **Top 5** | **Prop.** |
| `dhcpcd` | 21572 | 13651 | 63.28% | 851784 | 839518 | 98.56% |
| `nsd` | 38328 | 28022 | 73.11% | 2423193 | 2399449 | 99.02% |
| `tmux` | 49080 | 36999 | 75.39% | 4331232 | 2483020 | 57.33% |
| `gawk` | 47673 | 30631 | 64.25% | 8467667 | 8353204 | 98.65% |
| `bash` | 36924 | 27118 | 73.44% | 6067608 | 5470244 | 90.15% |
| `mutt` | 65756 | 44886 | 68.26% | 8261029 | 7897442 | 95.60% |
| `lynx` | 260220 | 181359 | 69.69% | 17451804 | 16362946 | 93.76% |
| `xpdf` | 105743 | 55651 | 52.63% | 32507885 | 32387655 | 99.63% |
| `python3` | 184189 | 119043 | 64.63% | 114439707 | 94946890 | 82.97% |
| `svn` | 213125 | 167042 | 78.38% | 91817728 | 88324837 | 96.20% |
| `emacs` | 250739 | 163956 | 65.39% | 252728727 | 248665346 | 98.39% |
| `git` | 243388 | 132674 | 54.51% | 182364152 | 155306147 | 85.16% |
| `kakoune` | 182631 | 55491 | 30.38% | 37689778 | 37157978 | 98.59% |
| `ruby` | 114277 | 66634 | 58.31% | 71941456 | 69333326 | 96.37% |
| `squid` | 725067 | 389949 | 53.78% | 189749146 | 159336073 | 83.97% |
| `wireshark` | 326974 | 147939 | 45.24% | 23789094 | 22960321 | 96.52% |
| **Geo. Mean** | | | 60.48% | | | 91.19% |

the idea of hash consing [7, 16, 19, 24], which aims to quickly identify structurally equivalent values, from the functional programming community, to help solve the problem of duplicate points-to sets and unions operations. Hash consing is the process of maintaining single immutable representations of data structures which can then be shared elsewhere referentially [38, 42]. In our context, this means that each unique points-to set is maintained only once such that points-to sets becomes persistent.

Originally, hash consing was used to memoise construction to avoid creating the same object twice, transforming construction into a hash table lookup of the elements of the object. If we view our union operation as a constructor, taking two points-to sets to create a new one, we can transform many union operations into hash table lookups (of a pair of references), which would be much cheaper than standard set unions as points-to sets become larger. Thus hash consing is a means for efficient memoisation allowing us to perform faster set unions. During points-to set resolution, we build up hash tables of previously performed operations, and use those results if the same operation occurs again.

Moreover, with points-to sets being represented as references we can perform fast comparisons between such sets in constant, instead of linear, time. Thus, we also explore the possibility of practically skipping some set operations completely by exploiting mathematical set properties. For example, since each points-to set, e.g., $x$ and $y$, is represented as a reference, the operands of a union like $x \cup y$

can be compared cheaply for equality, in which case the result is $x/y$, since the union operation is idempotent.

Our approach is efficient yet simple to implement, independent to the points-to analysis used, maintains precision, and works alongside the many algorithmic advances listed earlier. Moreover, our approach does not mandate a specific representation of points-to sets as long as each pointer would otherwise be assigned discrete points-to sets. As far as we know, this paper is the first to describe general hash consing and its effects to the base aspects of inclusion-based points-to analysis. We have implemented our approach on top of points-to analysis framework SVF [47] and evaluated our approach using 16 real-world open-source programs (more than 9.5 million lines of LLVM instructions). For these programs, we find an average improvement in time taken of 1.85× for Andersen's analysis and 1.69× for SFS, and we observe improvements of up to 3.21× and 2.23× for the two analyses, respectively. Along with improved time, we see roughly the same memory usage for Andersen's analysis and an average reduction of ≥4.93× (up to ≥15.52×) for SFS.

To summarise, our contributions are:

- Persistent points-to data structure using hash consed points-to sets with less memory for precise whole program points-to analyses.
- The use of hash consing to more efficiently perform points-to set union operations through cheap memoisation and exploitation of set properties.
- An evaluation of the impact of hash consing on field-sensitive Andersen's analysis and SFS using 16 real world open source C/C++ programs, as well as a discussion on the amount of duplication found in these analyses.

## 2 Background and Motivation

This section first introduces a program representation for points-to analysis to be built upon. We then provide a brief summary of whole-program flow-insensitive and flow-sensitive inclusion-based points-to analysis. Finally, we give two short examples to illustrate the presence of duplicate points-to sets and union operations produced by these two analyses to motivate how hash consing can help.

### 2.1 Background

Like many other C/C++ analyses [2, 4, 23, 30, 47], we perform points-to analysis on top of the LLVM-IR of a program. In LLVM's partial SSA form [28], the set of all program variables $\mathcal{V} = \mathcal{O} \cup \mathcal{P}$ is split into two subsets: (1) $\mathcal{O}$, or the set of *address-taken variables*, which represents all possible abstract memory objects and their fields, and (2) $\mathcal{P}$, or the set of *top-level variables*, which represents all stack virtual registers (symbols starting with `%`) and global pointers (symbols starting with `@`). Top-level variables, $\mathcal{P}$, are explicit in that they are accessed directly whereas address-taken objects, $\mathcal{O}$, are implicit and can only be accessed indirectly at Load and Store instructions through top-level variables.

Given $p, q \in \mathcal{P}$ and $o \in \mathcal{O}$, after the SSA conversion, we represent a C/C++ program using the following five types of instructions:

- ALLOC instructions, $p = alloc_o$, representing allocation of abstract object $o$.
- COPY instructions, $p = q$, representing assignment between two top-level pointers.
- FIELD instructions, $p = \&q \rightarrow f_i$, representing assignment of the $i$-th field ($f_i$) of the object which $q$ points to.
- LOAD instructions, $p = *q$, representing assignment from a dereferenced top-level pointer.
- STORE instructions, $*p = q$, representing assignment to an abstract memory object through a dereferenced top-level pointer.

With the above instructions, Figure 1 presents a flow-insensitive inclusion-based analysis commonly referred to as Andersen's analysis [1] augmented with field-sensitivity [35]. Each pointer $p$ is assigned a points-to set $pt(p)$ representing an approximation of the set of abstract memory objects that $p$ may point to. Andersen's analysis is performed by generating constraints between points-to sets according to the five inference rules. The COPY, LOAD, and STORE rules produce inclusion or union constraints like $pt(q) \subseteq pt(p)$ which means the points-to set of $p$ is the union of its old value and the points-to set of $q$, i.e., $pt(p) = pt(p) \cup pt(q)$. The produced constraints are iteratively solved with points-to sets growing monotonically until a fixed-point is reached.

$$\texttt{[ALLOC]}\ \frac{p = alloc_o}{\{o\} \subseteq pt(p)} \qquad \texttt{[COPY]}\ \frac{p = q}{pt(q) \subseteq pt(p)} \qquad \texttt{[FIELD]}\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{\{o.f_i\} \subseteq pt(p)}$$

$$\texttt{[LOAD]}\ \frac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)} \qquad \texttt{[STORE]}\ \frac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)}$$

Fig. 1: Inference rules for a flow-insensitive inclusion-based points-to analysis.

More precise analyses typically need to compute and maintain more points-to relations. For example, in a flow-sensitive analysis, an object accessed at different program points can have different points-to sets, thus requiring more points-to sets and constraints. Figure 2 gives a simple inclusion-based flow-sensitive analysis [30] augmented with field-sensitivity. Since the analysis is flow-sensitive, the order of instructions now matters and so each instruction is prefixed by a label like $\ell$ to represent the points-to information at a particular program point.

Unlike flow-insensitive analysis which computes a single points-to set for each variable, flow-sensitive analysis maintains separate points-to sets at different program points for each memory object. To represent points-to information flow-sensitively, points-to sets of memory objects are maintained before ($pt[\overline{\ell}](o)$) and after ($pt[\underline{\ell}](o)$) instructions. Thus, points-to sets need to be propagated within program points through the [SU/WU] rule and, if there exists control flow between two instructions ($\ell \rightarrow \ell'$), across program points through the [CFLOW]

$$\texttt{[ALLOC]}\ \frac{\ell : p = alloc_o}{\{o\} \subseteq pt(p)} \qquad \texttt{[COPY]}\ \frac{\ell : p = q}{pt(q) \subseteq pt(p)} \qquad \texttt{[FIELD]}\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{\{o.f_i\} \subseteq pt(p)}$$

$$\texttt{[LOAD]}\ \frac{\ell : p = *q \quad o \in pt(q)}{pt(\overline{\ell})(o) \subseteq pt(p)} \qquad \texttt{[STORE]}\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt(\underline{\ell})(o)}$$

$$\texttt{[SU/WU]}\ \frac{\ell : *p = \_ \quad o \in \mathcal{O} \setminus kill(\ell)}{pt(\overline{\ell})(o) \subseteq pt(\underline{\ell})(o)} \qquad \texttt{[CFLOW]}\ \frac{\ell \rightarrow \ell'}{\forall o \in \mathcal{O}.\ pt(\underline{\ell})(o) \subseteq pt(\overline{\ell'})(o)}$$

$$kill(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Fig. 2: Inference rules for a field-sensitive and flow-sensitive inclusion-based points-to analysis.

rule. This all results in extra set unions between points-to sets being performed. With the *kill* function, the [SU/WU] rule can perform strong updates for singletons [30], another way flow-sensitivity produces more precise results.

To reduce some of these redundancies, state-of-the-art flow-sensitive analyses like staged flow-sensitive analysis (SFS) [23] perform points-to propagation on a sparse def-use graph rather than a control-flow graph of a program. Despite this, redundancies still exist, and duplication is high, as will be shown in Section 4.

### 2.2 Motivating Examples

In this section, we show the duplication of points-to sets and operations that occurs in flow-insensitive and flow-sensitive analyses. First, let us consider flow-insensitive analysis of the small program fragment in Figure 3a where $p, q, r, x, y \in \mathcal{P}$ and $o_1, o_2, o_3, o_4 \in \mathcal{O}$. Figure 3b shows the constraints produced to analyse this program fragment following the rules in Figure 1. Since the analysis is flow-insensitive, we solve for a points-to set per variable. We use $pt(p)$ to denote the points-to set of pointer $p$ and use $\{o_1\}_p$ to denote the value of $pt(p)$ when it, for example, contains the points-to target $o_1$. In analysing the program fragment, we assume $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$.

In practice, these constraints are handed to a constraint solver [36, 35, 20, 13] which will perform unions like those in Figure 3c until a fixed-point is reached, i.e., when points-to sets no longer change. In Figure 3c, operations are numbered with the constraints they correspond to and duplicate operations are highlighted in grey. For brevity, we have only shown the first operation which would result from a constraint. Ultimately, each constraint actually results in the same initial union being performed so 3 of the 4 operations are duplicates of the first. In real-world programs, such points-to sets may be large, containing hundreds or thousands of objects, meaning repeatedly performing these unions can be expensive. The final resulting points-to sets of the analysis are shown in Figure 3d, with duplicates also highlighted in grey. We see that 5 of the 9 points-to sets have occurred before, pointing to much duplication. This can be problematic as

| | | |
|---|---|---|
| $1 : *p = r;$ | $1 : \forall o \in pt(p).\ pt(r) \subseteq pt(o)$ | $1 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_1}$ |
| $2 : *q = r;$ | $2 : \forall o \in pt(q).\ pt(r) \subseteq pt(o)$ | $2 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_2}$ |
| $3 : x = *p;$ | $3 : \forall o \in pt(p).\ pt(o) \subseteq pt(x)$ | $3 : \{o_3, o_4\}_{o_1} \subseteq \{\ \}_x$ |
| $4 : y = *q;$ | $4 : \forall o \in pt(q).\ pt(o) \subseteq pt(y)$ | $4 : \{o_3, o_4\}_{o_2} \subseteq \{\ \}_y$ |
| (a) Program fragment. | (b) Constraints. | (c) Initial operations. |

$pt(p) = \{o_1\}$ $pt(q) = \{o_2\}$ $pt(r) = \{o_3, o_4\}$ $pt(x) = \{o_3, o_4\}$ $pt(y) = \{o_3, o_4\}$

$pt(o_1) = \{o_3, o_4\}$ $pt(o_2) = \{o_3, o_4\}$ $pt(o_3) = \{\ \}$ $pt(o_4) = \{\ \}$

(d) Result.

Fig. 3: Example program fragment in (a), constraints generated for Andersen's analysis in (b), initial operations performed to fulfil the constraints in (c), and final results in (d). We assume $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$. Duplicate points-to sets and operations are highlighted in grey.

points-to sets grow, with statically sized representations, or as analyses introduce more variables.

We shorten the program fragment above in Figure 4a[3] to illustrate the same issues in flow-sensitive points-to analysis. Figure 4b lists the constraints generated according to the rules in Figure 2 followed by the (initial) operations performed to fulfil those constraints and the final result of the analysis in Figures 4c and 4d. We highlight duplicate operations and points-to sets in grey.

Since the analysis is flow-sensitive, we need to maintain points-to sets of objects at program points for precise results, thus resulting in more pointers being kept track of. By maintaining points-to sets at program points, we also require more operations to handle the flow of control. This can all be seen by the increase in number of operations and points-to sets in Figures 4c and 4d despite the smaller program fragment. The improved precision can be seen through the differing points-to sets of some objects at different program points, for example, $pt[\overline{1}](o_1) \neq pt[\underline{1}](o_1)$. However, this increased precision comes at a cost of increased redundancy as some points-to sets do not differ between program points, like those of $o_3$ and $o_4$. Thus, we see that there are only 4 unique points-to sets out of 19, and 3 unique operations out of 14, meaning that the analysis is maintaining duplicate points-to sets and performing duplicate operations.

We note that SFS, one of the analyses we evaluate our approach on, can remove some of this duplication and redundancy through complex algorithmic changes to the analysis in Figure 2, but much duplication still exists, as will be seen in Section 4. We also note that although many points-to sets in these

[3] Due to the large number of points-to sets and unions flow-sensitive analysis produces.

$1 : *p = r;$
$2 : *q = r;$

(a) Program fragment.

$1 : \forall o \in pt(p).\ pt(r) \subseteq pt(o)$
$1 : \forall o \in \mathcal{O}.\ pt[\overline{1}](o) \subseteq pt[\underline{1}](o)$
$1/2 : \forall o \in \mathcal{O}.\ pt[\underline{1}](o) \subseteq pt[\overline{2}](o)$
$2 : \forall o \in pt(q).\ pt(r) \subseteq pt(o)$
$2 : \forall o \in \mathcal{O}.\ pt[\overline{2}](o) \subseteq pt[\underline{2}](o)$

(b) Constraints.

$1 : \{o_3, o_4\}_r \subseteq \{\ \}_{[\underline{1}]o_1}$ $\quad$ $2 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_2}$

$1 : \{\ \}_{[\overline{1}]o_1} \subseteq \{o_3, o_4\}_{[\underline{1}]o_1}$ $\quad$ $1/2 : \{o_3, o_4\}_{[\underline{1}]o_1} \subseteq \{\ \}_{[\overline{2}]o_1}$ $\quad$ $2 : \{o_3, o_4\}_{[\overline{2}]o_1} \subseteq \{\ \}_{[\underline{2}]o_1}$

$1 : \{\ \}_{[\overline{1}]o_2} \subseteq \{\ \}_{[\underline{1}]o_2}$ $\quad$ $1/2 : \{\ \}_{[\underline{1}]o_2} \subseteq \{\ \}_{[\overline{2}]o_2}$ $\quad$ $2 : \{\ \}_{[\overline{2}]o_2} \subseteq \{o_3, o_4\}_{[\underline{2}]o_2}$

$1 : \{\ \}_{[\overline{1}]o_3} \subseteq \{\ \}_{[\underline{1}]o_3}$ $\quad$ $1/2 : \{\ \}_{[\underline{1}]o_3} \subseteq \{\ \}_{[\overline{2}]o_3}$ $\quad$ $2 : \{\ \}_{[\overline{2}]o_3} \subseteq \{\ \}_{[\underline{2}]o_3}$

$1 : \{\ \}_{[\overline{1}]o_4} \subseteq \{\ \}_{[\underline{1}]o_4}$ $\quad$ $1/2 : \{\ \}_{[\underline{1}]o_4} \subseteq \{\ \}_{[\overline{2}]o_4}$ $\quad$ $2 : \{\ \}_{[\overline{2}]o_4} \subseteq \{\ \}_{[\underline{2}]o_4}$

(c) Initial operations.

$pt(p) = \{o_1\}$ $\quad$ $pt(q) = \{o_2\}$ $\quad$ $pt(r) = \{o_3, o_4\}$

$pt[\overline{1}](o_1) = \{\ \}$ $\quad$ $pt[\overline{1}](o_2) = \{\ \}$ $\quad$ $pt[\overline{1}](o_3) = \{\ \}$ $\quad$ $pt[\overline{1}](o_4) = \{\ \}$

$pt[\underline{1}](o_1) = \{o_3, o_4\}$ $\quad$ $pt[\underline{1}](o_2) = \{\ \}$ $\quad$ $pt[\underline{1}](o_3) = \{\ \}$ $\quad$ $pt[\underline{1}](o_4) = \{\ \}$

$pt[\overline{2}](o_1) = \{o_3, o_4\}$ $\quad$ $pt[\overline{2}](o_2) = \{\ \}$ $\quad$ $pt[\overline{2}](o_3) = \{\ \}$ $\quad$ $pt[\overline{2}](o_4) = \{\ \}$

$pt[\underline{2}](o_1) = \{o_3, o_4\}$ $\quad$ $pt[\underline{2}](o_2) = \{o_3, o_4\}$ $\quad$ $pt[\underline{2}](o_3) = \{\ \}$ $\quad$ $pt[\underline{2}](o_4) = \{\ \}$

(d) Result.

Fig. 4: Example program fragment in (a), constraints generated for a flow-sensitive analysis in (b), initial operations performed to fulfil the constraints in (c), and final results in (d). We assume $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$. Duplicate points-to sets and operations are highlighted in grey.

examples were empty sets which can be easily represented, real-world programs show duplication of larger points-to sets and more complex union operations.

## 3 Approach

This section introduces hash consed points-to sets and its application to points-to analysis. We then describe optimisations that can use hash consing to efficiently exploit set properties for further performance improvement.

### 3.1 Hash Consed Points-To Sets

Hash consing is used to create immutable data structures which can be shared (referentially) to avoid duplication. A common example of hash consing is string

interning [18, §3.10.5] whereby a compiler or runtime stores strings in a global pool and assigns pointers to strings in that global pool rather than private copies. In our context, we want points-to sets to be stored once in a global pool, so that we deal with references to points-to sets rather than concrete points-to sets.

To do this, whenever a points-to set is created, we perform an interning routine. We check if that points-to set exists in our global pool, and

- If it exists, return a reference to the equivalent set in the global pool.
- Otherwise, add the points-to set to the global pool and return a reference to the newly added points-to set.

This process can be achieved by a single hash table mapping each points-to sets to a single canonical reference. Now, instead of using $pt(p)$ during the analysis, we use $pt_r(p)$ which is a reference to the points-to set of $p$ in the global pool. Dereferencing a points-to set reference as $dr(pt_r(p))$ would be equivalent to $pt(p)$ and can be used to iterate over the points-to set, for example. Given that $pt_r(p) = pt_r(q)$, $dr(pt_r(p))$ and $dr(pt_r(q))$ would also be equivalent and actually be accessing the same singly stored points-to set in the global pool. This can save significant memory if duplicate points-to sets are common.

On its own, this process does not save time, and may cost more time to perform the interning routine, especially as we perform many unions creating points-to sets which need to be interned. Since each unique points-to set exists once in the program, we can efficiently memoise operations, including the union operation. This can be achieved by a hash table, which we call an operations table, mapping two points-to set references to the points-to set reference which refers to the result of the actual operation. The union between two points-to set references $pt_r(p) \cup pt_r(q)$ can be performed by looking up the union operations table with the $\langle pt_r(p), pt_r(q) \rangle$ pair as the key (i.e., operation), and

- If the key exists, returning the associated value, i.e., the reference to the result of the operation.
- Otherwise, performing a concrete union between $dr(pt_r(p))$ and $dr(pt_r(q))$, interning the result, associating the operation with the result in the operations table, and returning it.

With many union operations being duplicates, those would be performed as constant time hash table lookups, rather than linear time set unions[4] which can be expensive depending on sizes of points-to sets. The intersection and difference operations can also be memoised the same way, if necessary.

Without hash consing, memoising operations would not be efficient as we would need to hash entire points-to sets, i.e., we would map $\langle pt(p), pt(q) \rangle$ to another concrete points-to set rather than mapping a reference pair to a reference. Collisions would also be expensive to resolve as determining equality would then be linear in the size of the colliding points-to set pairs. With references, equality can be determined in constant time.

[4] For our SVF-based implementation we use sparse bit-vectors (Section 4.1).

$$\boxed{\begin{aligned}\{o_1\} &\mapsto r_1\\ \{o_2\} &\mapsto r_2\\ \{o_3, o_4\} &\mapsto r_3\\ \{\ \} &\mapsto r_4\end{aligned}}$$

(a) Global pool mapping points-to sets to references.

$$\boxed{\langle r_3, r_4\rangle \mapsto r_3}$$

(b) Union operations table.

$$pt_r(p) = r_1 \quad pt_r(q) = r_2 \quad pt_r(r) = r_3 \quad pt_r(x) = r_3 \quad pt_r(y) = r_3$$
$$pt_r(o_1) = r_3 \quad pt_r(o_2) = r_3 \quad pt_r(o_3) = r_4 \quad pt_r(o_4) = r_4$$

(c) Result.

Fig. 5: Global pool of points-to sets in (a), the union operations table in (b), and the result in (c) using references instead of concrete points-to sets for the analysis in Figure 3.

$$\boxed{\begin{aligned}\{o_1\} &\mapsto r_1\\ \{o_2\} &\mapsto r_2\\ \{o_3, o_4\} &\mapsto r_3\\ \{\ \} &\mapsto r_4\end{aligned}}$$

(a) Global pool mapping points-to sets to references.

$$\boxed{\begin{aligned}\langle r_3, r_4\rangle &\mapsto r_3\\ \langle r_4, r_4\rangle &\mapsto r_4\\ \langle r_4, r_3\rangle &\mapsto r_3\end{aligned}}$$

(b) Union operations table.

$$pt(p) = r_1 \quad pt(q) = r_2 \quad pt(r) = r_3$$
$$pt[\overline{1}](o_1) = r_4 \quad pt[\overline{1}](o_2) = r_4 \quad pt[\overline{1}](o_3) = r_4 \quad pt[\overline{1}](o_4) = r_4$$
$$pt[\underline{1}](o_1) = r_3 \quad pt[\underline{1}](o_2) = r_4 \quad pt[\underline{1}](o_3) = r_4 \quad pt[\underline{1}](o_4) = r_4$$
$$pt[\overline{2}](o_1) = r_3 \quad pt[\overline{2}](o_2) = r_4 \quad pt[\overline{2}](o_3) = r_4 \quad pt[\overline{2}](o_4) = r_4$$
$$pt[\underline{2}](o_1) = r_3 \quad pt[\underline{2}](o_2) = r_3 \quad pt[\underline{2}](o_3) = r_4 \quad pt[\underline{2}](o_4) = r_4$$

(c) Result.

Fig. 6: Global pool of points-to sets in (a), the union operations table in (b), and the result in (c) using references instead of concrete points-to sets for the analysis in Figure 4.

Figures 5 and 6 show how our analysis would look for the examples in Figures 3 and 4 respectively. Three of the four union operations between points-to sets $\{o_3, o_4\}$ and $\{\ \}$ are performed as cheap lookups in the operations table in Figure 5b. This is because the first time we perform a concrete operation, we cache it in the operations table, and perform a fast lookup on subsequent operations. As in Figure 5c, we store references to points-to sets in the global pool (Figure 5a) rather than concrete points-to sets, and so we only store 4 concrete points-to sets. Figure 6 illustrates that all of the initial points-to unions in the flow-sensitive example are translated into 3 unique operations. Furthermore, for the flow-sensitive example, the effect of using references into the global pool for points-to sets is more drastic since there are so many pointers tracked, saving significant memory.

### 3.2 Exploiting Set Properties

In this section, we describe some optimisations which exploit the properties of sets to further improve efficiency of union operations on hash consed points-to sets. We note that even though our rules in Figures 1 and 2 only perform unions, practical implementations may perform intersection and difference operations. Furthermore, clients may perform some of these operations too, like alias analysis which performs intersection tests. These operations can be memoised in the same way as unions above, and we exploit their properties in this section too.

**Commutative operations** For commutative operations like unions and intersections, performing an operation twice with the operands flipped is duplication though this would not be detected in the operations tables. For example, assuming $pt_r(p) = x$ and $pt_r(q) = y$, if we perform $x \cup y = z$ for the first time, we would store a mapping from the pair $\langle x, y\rangle$ to the result $z$ in the union operations table. If the analysis was to perform $y \cup x$, it would not find the operation memoised, despite the result also being $z$, as $\langle y, x\rangle$ would not be cached in the union operations table.

To resolve this, operations should always be ordered deterministically. This is easy to achieve with hash consing because points-to sets are references and can be compared in constant time. Now, to perform $x \cup y$ or $y \cup x$, we would perform the operation in the same order depending on whether $x$ is "less than" $y$, and so only a single instance would be stored in the union operations table. In Figure 6b, the first and third operation are actually equivalent, and under this scheme would be stored once as $\langle r_3, r_4\rangle \mapsto r_3$.

**Property Operations** In some cases, the result of an operation can be determined instantly with only trivial comparisons without any concrete operation or hash table lookup. We refer to these cases as *property operations*, and we describe these cases for unions, intersections, and differences below. We set $e$ to refer to the empty points-to set, and for commutative operations (i.e., unions and intersections), we assume the operands have already been ordered and that

the reference $e$ is the least reference (so it is always the first operand in the commutative operations it appears in).

*Unions* Given the ordered union operation between references $x$ and $y$, $x \cup y$, and that the result would be $r$,

$$x = e \Rightarrow r = y, \text{ and}$$
$$x = y \Rightarrow r = x.$$

All operations in Figures 5b and 6b are actually property operations and caching is unnecessary.

*Intersections* Given the ordered intersection operation between references $x$ and $y$, $x \cap y$, and that the result would be $r$,

$$x = e \Rightarrow r = e, \text{ and}$$
$$x = y \Rightarrow r = x.$$

*Difference* Given the difference operation between references $x$ and $y$, $x - y$, and that the result would be $r$,

$$x = e \Rightarrow r = e,$$
$$y = e \Rightarrow r = x, \text{ and}$$
$$x = y \Rightarrow r = e.$$

**Preemptive Memoisation** After performing an actual operation and caching that operation in the operation table, we can preemptively cache other operations too by exploiting standard set properties. This would avoid performing an actual operation later if the analysis needed that result. An implementation can choose which operations are worth preemptively memoising and which are not.

*Unions* Assume the ordered operation $x \cup y = r$ is not a property operation. If $x \neq r$, we can instantly determine and cache

$$x \cup r = r, \text{ and}$$
$$x \cap r = x,$$

and similarly if $y \neq r$,

$$y \cup r = r, \text{ and}$$
$$y \cap r = y.$$

We guard with the conditions $x \neq r$ and $y \neq r$ because in each of these cases the preemptively cached unions would be property unions.

*Intersections* Assume the ordered operation $x \cap y = r$ is not a property operation. If $r \neq e \wedge x \neq r$, we can instantly determine and cache

$$x \cap r = r, \text{ and}$$
$$x \cup r = x,$$

and similarly if $r \neq e \wedge y \neq r$,

$$y \cap r = r, \text{ and}$$
$$y \cup r = y.$$

We are not interested in preemptively memoising when $r = e$ because these intersections and unions would otherwise be property operations.

*Difference* Assume the difference operation $x - y = r$ is not a property operation. If $r \neq e \wedge x \neq r$ we can instantly determine and cache

$$x \cup r = x, \text{ and}$$
$$x \cap r = r,$$

and similarly if $r \neq e$,

$$y - r = y,$$
$$r - y = r, \text{ and}$$
$$r \cap y = e.$$

## 4 Evaluation

This section describes our implementation, programs used to evaluate our approach, and then discusses results obtained when applying our hash consed points-to sets to state-of-the-art inclusion-based flow-insensitive analysis (Andersen's analysis [1, 36]) and inclusion-based flow-sensitive analysis (staged flow-sensitive analysis [23]).

### 4.1 Implementation and Experimental Setup

We have implemented our approach using open source points-to analysis framework SVF [47] built on LLVM 10.0.0. We have not modified any algorithms, rather just how points-to sets are represented, that is, when an analysis attempts to perform a union or access a points-to set, our code is called. For concrete points-to sets, we use LLVM's sparse bit-vector. SVF's flow-insensitive points-to analysis or Andersen's analysis uses a state-of-the-art constraint resolution algorithm, *wave propagation* [36], and performs cycle detection. Indirect calls (function pointers and virtual calls) are resolved on-the-fly during points-to resolution. SVF's flow-sensitive analysis is staged flow-sensitive analysis (SFS) as

Table 2: Program versions, bitcode sizes, lines of LLVM instructions, and descriptions.

| Program | Version | Size | LOI | Description |
|---|---|---|---|---|
| `dhcpcd` | 9.3.4 | 1.19 MB | 82 939 | DHCP client |
| `nsd` | 4.3.4 | 1.72 MB | 117 191 | Name server |
| `tmux` | 3.1c | 2.41 MB | 156 872 | Terminal multiplexer |
| `gawk` | 5.1.0 | 2.48 MB | 179 805 | GNU AWK interpreter |
| `bash` | 5.0.18 | 2.68 MB | 196 168 | Bourne Again Shell (Unix shell) |
| `mutt` | 2.0.3 | 3.28 MB | 224 500 | Text-based email client |
| `lynx` | 2.8.9 | 5.31 MB | 287 159 | Text-based web browser |
| `xpdf` | 4.03 | 7.90 MB | 494 764 | PDF viewer |
| `python3` | 3.7.9 | 9.80 MB | 635 361 | Python 3 interpreter |
| `svn` | 1.14.0 | 11.40 MB | 673 144 | Version control system |
| `emacs` | 27.2 | 11.85 MB | 804 291 | extensible text editor |
| `git` | 2.29.2 | 12.29 MB | 739 968 | Distributed version control system |
| `kakoune` | 2020.08.04 | 12.39 MB | 733 327 | Modal text editor |
| `ruby` | 2.7.2 | 13.05 MB | 864 114 | Ruby interpreter |
| `squid` | 4.13 | 20.36 MB | 1 252 756 | Web proxy cache |
| `wireshark` | 3.4.0 | 32.59 MB | 2 145 391 | Network packet analyser |

described in Section V of the original work [23]. Unlike Figure 2, SFS performs points-to analysis on a pre-computed def-use graph, not a control-flow graph, vastly reducing the number of constraints. Both analyses are field-sensitive and assume analysed programs do not perform pointer arithmetic to access fields. Fields of struct objects are distinguished by their unique indices [35].

For our hash consed points-to sets, we map concrete points-to sets to unique integer identifiers (which act as our references), and a second map, implemented as an array for performance, mapping those identifiers back to the concrete points-to set. This allows us to use 32-bit identifiers, rather than 64-bit addresses as would be required if our references were pointers. Our operations tables are implemented as maps mapping two such identifiers to another. Our hash function is simply the concatenation of the two 32-bit identifier operands which is another benefit of using integral identifiers as references.

We have run Andersen's analysis and SFS with and without hash consed points-to sets on 16 real-world open source programs from various domains. Table 2 lists these programs along with their version, bitcode size, number of lines of LLVM instructions, and a short description. `xpdf`, `kakoune`, `squid`, and `wireshark` are written in C++ and the remainder are C programs. We ran the analyses on a machine running 64-bit Ubuntu 18.04.2 LTS with an Intel Xeon 6132 processor and we limited analyses to 100 GB of memory. To measure time, we use C's `clock` function and to measure memory we refer to the maximum resident set size of the entire SVF execution reported by GNU's `time`.

Table 3: Time taken (seconds) and memory usage (GB) for Andersen's analysis with and without hash consing, followed by the time and memory difference of the two approaches.

| Program | Baseline | | Hash consed | | Time diff. | Memory diff. |
|---|---|---|---|---|---|---|
| | Time | Memory | Time | Memory | | |
| `dhcpcd` | 4.52 | 0.30 | 3.58 | 0.28 | 1.26× | 1.07× |
| `nsd` | 9.32 | 0.55 | 7.23 | 0.51 | 1.29× | 1.07× |
| `tmux` | 18.86 | 0.59 | 14.11 | 0.56 | 1.34× | 1.05× |
| `gawk` | 19.92 | 0.64 | 14.15 | 0.58 | 1.41× | 1.10× |
| `bash` | 10.93 | 0.64 | 7.29 | 0.58 | 1.50× | 1.11× |
| `mutt` | 41.79 | 1.01 | 20.09 | 0.95 | 2.08× | 1.06× |
| `lynx` | 61.09 | 1.11 | 44.51 | 1.03 | 1.37× | 1.08× |
| `xpdf` | 179.52 | 1.94 | 111.80 | 1.88 | 1.61× | 1.03× |
| `python3` | 5509.52 | 4.13 | 1779.64 | 3.51 | 3.10× | 1.18× |
| `svn` | 5869.05 | 4.24 | 1829.20 | 2.82 | 3.21× | 1.50× |
| `emacs` | 5082.81 | 13.63 | 2651.32 | 13.05 | 1.92× | 1.05× |
| `git` | 5905.84 | 6.73 | 2499.55 | 6.79 | 2.36× | 0.99× |
| `kakoune` | 673.88 | 3.07 | 263.08 | 3.26 | 2.56× | 0.94× |
| `ruby` | 67.32 | 2.74 | 32.08 | 2.58 | 2.10× | 1.06× |
| `squid` | 2752.84 | 6.30 | 949.33 | 5.03 | 2.90× | 1.25× |
| `wireshark` | 271.60 | 6.42 | 211.42 | 6.21 | 1.28× | 1.03× |
| **Geo. Mean** | | | | | 1.85× | 1.09 × |

### 4.2 Results and Discussion

In this section, we discuss the effects of hash consing on points-to analysis. We first look at Andersen's analysis then SFS, and conclude with a brief discussion on preemptive memoisation.

**Andersen's Analysis** Table 3 shows the time and memory of Andersen's analysis with and without hash consing, and comparisons are shown in the **Time diff.** and **Memory diff.** columns. We see a positive trend in time, showing that using hash consing speeds up the analysis by a geometric mean of 1.85× for our programs. At most, the analysis is 3.21× faster, and at worst 1.26× faster. Generally, slower to analyse programs saw the greatest improvement in speed, with all programs which originally took over 5000 seconds to analyse seeing an improvement of over 2× with the exception of `emacs` which saw a slightly lower improvement.

For memory, we see around the same usage generally with the hash consed analysis using slightly more or slightly less. We have not implemented garbage collection for the global pool of points-to sets. When there exists no references to a certain points-to set in the global pool, that points-to set can be destroyed, or garbage collected. This would save memory, as intermediate points-to sets which are no longer in use litter the global pool. We strongly suspect that garbage

Table 4: Number of union operations which are concrete operations, property operations (and their proportion), lookups into the union operations table (and their proportion), and the total for Andersen's analysis using our approach.

| Program | Concrete | Property | Lookup | Total |
|---|---|---|---|---|
| `dhcpcd` | 3766 (3.10%) | 58 424 (48.14%) | 59 185 (48.76%) | 121 375 |
| `nsd` | 2900 (1.76%) | 98 068 (59.45%) | 64 001 (38.80%) | 164 969 |
| `tmux` | 5651 (1.58%) | 102 511 (28.58%) | 250 552 (69.85%) | 358 714 |
| `gawk` | 6378 (2.26%) | 155 251 (55.09%) | 120 172 (42.64%) | 281 801 |
| `bash` | 1358 (0.93%) | 126 307 (86.61%) | 18 167 (12.46%) | 145 832 |
| `mutt` | 8135 (3.07%) | 145 604 (55.02%) | 110 881 (41.90%) | 264 620 |
| `lynx` | 10 750 (3.19%) | 188 602 (56.04%) | 137 205 (40.77%) | 336 557 |
| `xpdf` | 29 622 (4.32%) | 249 768 (36.41%) | 406 582 (59.27%) | 685 972 |
| `python3` | 33 274 (3.16%) | 560 048 (53.25%) | 458 319 (43.58%) | 1 051 641 |
| `svn` | 22 879 (1.45%) | 808 308 (51.13%) | 749 564 (47.42%) | 1 580 751 |
| `emacs` | 92 677 (4.61%) | 809 938 (40.27%) | 1 108 850 (55.13%) | 2 011 465 |
| `git` | 124 333 (9.03%) | 684 809 (49.73%) | 567 897 (41.24%) | 1 377 039 |
| `kakoune` | 86 364 (8.72%) | 394 225 (39.81%) | 509 693 (51.47%) | 990 282 |
| `ruby` | 11 090 (3.15%) | 195 495 (55.47%) | 145 827 (41.38%) | 352 412 |
| `squid` | 55 792 (3.23%) | 796 024 (46.09%) | 875 241 (50.68%) | 1 727 057 |
| `wireshark` | 47 856 (3.02%) | 592 647 (37.34%) | 946 580 (59.64%) | 1 587 083 |
| **Geo. Mean** | – (2.98%) | – (48.40%) | – (44.24%) | |

collection of the global pool can further save memory and eliminate memory usage regressions, which we would like to explore in the future.

Table 4 lists the union operations performed by the Andersen's analysis and categorises them as concrete (unique) unions, property unions, or lookups. When we preemptively memoise, we count such an operation as a property operation. We see that in every program, less than 10% of unions are concrete unions, meaning the remainder are either property unions, and thus trivial, or duplicates of a non-property union. In fact, we only see more than 5% for two programs, `git` and `kakoune`. In most programs, the number of property unions and lookups are roughly even. It is interesting to note that despite the small number of unions (compared to more precise analyses, as will be seen in the next section), hash consing has produced a noticeable speedup.

**SFS** Table 5 shows the time taken and memory used by SFS with and without hash consing. For time, we see a very similar to trend to that of Andersen's analysis. Unexpectedly, considering how many more constraints flow-sensitive analysis can produce, we see a lower geometric mean of 1.69×. This can be explained by the lack of analysis timing data for 9 programs without hash consing, i.e., the baseline, because those analyses exceeded the allocated 100 GB of memory, and thus we cannot draw a time comparison. If sufficient memory resources were available, we would expect to see a much larger average improvement as

Table 5: Time taken (seconds) and memory usage (GB) for SFS with and without hash consing, followed by the time and memory difference of the two approaches. OOM means the analysis exhausted the allocated 100 GB of memory.

| Program | Baseline | | Hash consed | | Time diff. | Memory diff. |
|---|---|---|---|---|---|---|
| | Time | Memory | Time | Memory | | |
| `dhcpcd` | 77.27 | 1.08 | 73.04 | 0.66 | 1.06× | 1.65× |
| `nsd` | 113.39 | 2.97 | 75.76 | 0.74 | 1.50× | 4.02× |
| `tmux` | 280.09 | 3.33 | 212.25 | 1.14 | 1.32× | 2.93× |
| `gawk` | 1526.61 | 12.13 | 685.78 | 2.42 | 2.23× | 5.02× |
| `bash` | 337.01 | 8.55 | 165.28 | 1.51 | 2.04× | 5.65× |
| `mutt` | 797.92 | 13.95 | 400.08 | 2.15 | 1.99× | 6.49× |
| `lynx` | 3256.47 | 26.71 | 1594.90 | 3.65 | 2.04× | 7.32× |
| `xpdf` | OOM | OOM | 7210.36 | 6.44 | – | ≥15.52× |
| `python3` | OOM | OOM | 23534.00 | 16.72 | – | ≥5.98× |
| `svn` | OOM | OOM | 14000.10 | 22.61 | – | ≥4.42× |
| `emacs` | OOM | OOM | 51367.00 | 44.50 | – | ≥2.25× |
| `git` | OOM | OOM | 49264.50 | 39.59 | – | ≥2.53× |
| `kakoune` | OOM | OOM | 12845.40 | 9.49 | – | ≥10.53× |
| `ruby` | OOM | OOM | 4250.19 | 9.77 | – | ≥10.24× |
| `squid` | OOM | OOM | 72733.50 | 37.53 | – | ≥2.66× |
| `wireshark` | OOM | OOM | 24820.20 | 14.50 | – | ≥6.90× |
| **Geo. Mean** | | | | | 1.69× | ≥4.93× |

these 9 benchmarks are the largest and would be likely improve most. This can be gleaned from the data in Table 6 which shows the union type breakdown for SFS. We see that the number of unions is very high giving much room for our approach to improve time. Concrete unions never exceed 1% when using hash consing, thus hash consing and memoisation have improved over 99% of unions for our programs. We also see that, compared to Andersen's analysis, a larger proportion of unions have become property unions rather than lookups.

As for memory usage in Table 5, we see a significant improvement with a geometric mean reduction of over 4.93×, and at most over 15.52× (`xpdf`). Hash consing brings memory requirements to a level acceptable for commodity hardware: of the 9 programs which exceeded the allocated 100 GB in the baseline analysis, 6 now require less than 32 GB, and all suffice with less than 64 GB. Even though our implementation does not include garbage collection of unnecessary intermediate points-to sets in the global pool, our approach still shows significant memory reduction for more precise analyses like SFS. With garbage collection we expect to see an even greater improvement in memory usage.

**Effect of Preemptive Memoisation** For our programs, preemptive memoisation generally does not have a discernible effect on time. This is because preemptive memoisation reduces the number of concrete unions *after* the application of our techniques (i.e., after our other techniques have made the most

Table 6: Number of union operations which are concrete operations, property operations (and their proportion), lookups into the union operations table (and their proportion), and the total number of unions for SFS using hash consing.

| Program | Concrete | Property | Lookup | Total |
|---|---|---|---|---|
| dhcpcd | 858 019 (0.95%) | 60 000 495 (66.20%) | 29 772 284 (32.85%) | 90 630 798 |
| nsd | 106 236 (0.06%) | 131 385 659 (70.44%) | 55 032 002 (29.50%) | 186 523 897 |
| tmux | 265 726 (0.05%) | 515 202 000 (89.33%) | 61 282 554 (10.63%) | 576 750 280 |
| gawk | 2 568 240 (0.11%) | 1 674 478 472 (72.11%) | 645 178 369 (27.78%) | 2 322 225 081 |
| bash | 27 701 (0.01%) | 435 565 965 (84.61%) | 79 195 559 (15.38%) | 514 789 225 |
| mutt | 319 829 (0.02%) | 1 033 079 848 (78.53%) | 282 194 806 (21.45%) | 1 315 594 483 |
| lynx | 788 833 (0.02%) | 3 836 871 871 (79.72%) | 975 346 188 (20.26%) | 4 813 006 892 |
| xpdf | 2 375 069 (0.02%) | 9 475 061 599 (76.93%) | 2 838 361 665 (23.05%) | 12 315 798 333 |
| python3 | 1 125 561 (0.00%) | 27 494 110 299 (83.29%) | 5 516 560 498 (16.71%) | 33 011 796 358 |
| svn | 9 536 154 (0.04%) | 15 950 564 702 (73.53%) | 5 731 542 295 (26.42%) | 21 691 643 151 |
| emacs | 40 525 287 (0.04%) | 62 746 471 959 (67.68%) | 29 925 669 621 (32.28%) | 92 712 666 867 |
| git | 15 868 477 (0.03%) | 36 002 062 086 (75.28%) | 11 805 253 473 (24.69%) | 47 823 184 036 |
| kakoune | 833 730 (0.00%) | 21 708 874 709 (81.56%) | 4 907 142 103 (18.44%) | 26 616 850 542 |
| ruby | 1 219 328 (0.01%) | 11 142 763 302 (83.49%) | 2 202 254 328 (16.50%) | 13 346 236 958 |
| squid | 3 080 598 (0.00%) | 117 192 828 125 (85.99%) | 19 097 056 263 (14.01%) | 136 292 964 986 |
| wireshark | 9 219 867 (0.06%) | 7 534 330 653 (50.97%) | 7 237 949 555 (48.97%) | 14 781 500 075 |
| **Geo. Mean** | – (0.02%) | – (75.61%) | – (22.10%) | |

expensive operations cheaper, like transforming $N$ occurrences of a particularly expensive union into one concrete union followed by $N - 1$ lookups). That is, it reduces the number of the already reduced concrete unions (second column of Tables 4 and 6). Regardless, we notice that the number of concrete unions does meaningfully shrink. For example, for Andersen's and SFS respectively, we see about 2500 and 1 million fewer for `svn`, about 10 000 and 500 000 fewer for `squid`, and about 1000 and 7 million fewer for `emacs`. This indicates that as input programs grow and the difference in concrete unions starts to have a noticeable effect on time (e.g., when points-to sets become unreasonably large), the role preemptive memoisation plays can become more significant. As expected, we see a slight increase in memory usage due to storing more operations in the operations table (each entry taking 12 bytes, modulo any table overhead).

## 5 Related Work

**Inclusion-based Points-to Analysis** The study of inclusion-based points-to analysis has a long history [1, 6, 13, 20, 26, 29, 35, 36, 39, 44, 46]. Resolving points-to relations in inclusion-based analysis is formalised as a set-constraint problem often solved by using a constraint graph of a program. To boost the performance of points-to analysis, most existing efforts focus on improving the analysis at the algorithmic level (e.g., via developing more efficient constraint solvers [35, 36,

44]) or simplifying the constraint graph (e.g., cycle elimination [13, 20], variable substitution [21, 39], or selective precision [30, 41]).

Despite these efforts, redundant and duplicate points-to sets and operations still exist and can not be completely tamed by existing techniques. Pushing the boundaries through algorithmic changes to the points-to analysis may produce increasingly diminishing returns on performance. Unlike previous approaches which simplify constraints or make algorithmic changes to points-to analysis, the goal of this work is to improve underlying data structures.

**Data Structures for Points-to Analysis** There has been a handful of work on using and developing data structures, particularly through better representing points-to sets, for efficient points-to analysis. For computing and representing points-to sets, several data structures have been used including binary decision diagrams (BDDs) [5, 52], bit-vectors [22, 23, 31] and explicit representations such as B-trees [8] and hash-based sets [31]. BDD-based points-to analysis often requires expensive variable reordering to be efficient. Thus the benefits may not outweigh using explicit representations [8]. Moreover, they often require algorithmic changes to the points-to analysis [5, 54], introducing extra implementation complexity. Bit-vectors as arguably the most popular data structure to represent points-to sets having been used in mainstream frameworks such as Soot [31], WALA [50], and LLVM-based static analysis tools [23, 40, 47]. Bit-vectors have been shown more efficient than hash-based sets and sorted arrays [31], and BDDs [22]. In this paper, we demonstrate that our hash consed points-to sets work well on top of LLVM's sparse bit-vectors to boost the performance of state-of-the-art flow-insensitive and flow-sensitive points-to analyses.

**Hash Consing for Static Analysis** In unpublished work [25], Heintze described splitting points-to sets into two parts: a unique part (called an overflow list) and a shared part. The shared part can be described as hash consing and thus implements a finer-grained hash consing since it does this on subsets rather than entire sets. However, no memoisation is performed, and doing so would be less effective due to the overflow list where, for example, two sets may be equivalent but not share any parts (i.e. the unique parts are different and the shared parts are different). The data structure is also much more difficult to implement whereas what we have presented can be retrofitted onto most set-like data structures exposing necessary operations (largely the set union operation). An implementation is available in Soot [43] as the `SharedHybridSet`.

Hash consing has also more generally been explored for static analysis to represent, for example, memory maps and program states [12, 33], invocation graphs [11], subtrees [3], and constants [27], with success. Static analyses are ripe for hash consing and memoisation because they are by nature approximations designed to capture a *class* of runtime data and so contain many duplicate data structures, operations, or both. We believe this work is the first to apply hash consing to the base aspects of points-to analysis, i.e. points-to sets and

their unions, describe extra optimisations, and show why points-to analysis is perfectly suited for hash consing.

## 6 Conclusion

This paper uses hash consed points-to sets to produce a persistent data structure to reduce duplicate points-to sets, saving space, and memoise union operations, saving time, without any high-level algorithmic changes. Hash consing can effectively handle duplication during points-to resolution by representing points-to sets once and referring to such representations through references. Our approach can speed up duplicate union operations through efficient memoisation and operand comparisons. We have evaluated our approach using 16 real-world C/C++ programs (>9.5 million lines of LLVM instructions). We observe an average memory reduction of $\geq$4.93× (up to $\geq$15.52×) in staged flow-sensitive analysis (SFS) and an average speed up of 1.69× (up to 2.23×). We also observe a speed up in state-of-the-art Andersen's analysis of 1.85× on average (up to 3.21×) while using roughly the same amount of memory.

## Acknowledgements

We thank the reviewers for their suggestions on improving this work. The first author is supported by a PhD scholarship funded by CSIRO's Data61. This research is supported by Australian Research Grants DP200101328 and DP210101348.

## References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, University of Copenhagen, Denmark (1994)
2. Balatsouras, G., Smaragdakis, Y.: Structure-sensitive points-to analysis for C and C++. In: International Static Analysis Symposium. pp. 84–104. SAS '16, Springer, Germany (2016). https://doi.org/10.1007/978-3-662-53413-7_5
3. Ball, T., Rajamani, S.K.: Bebop: A path-sensitive interprocedural dataflow engine. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. pp. 97–103. PASTE '01, ACM, USA (2001). https://doi.org/10.1145/379605.379690
4. Barbar, M., Sui, Y., Chen, S.: Object versioning for flow-sensitive pointer analysis. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization. pp. 222–235. CGO '21, IEEE Computer Society, USA (2021). https://doi.org/10.1109/CGO51591.2021.9370334
5. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation. pp. 103–114. PLDI '03, ACM, USA (2003). https://doi.org/10.1145/781131.781144
6. Blackshear, S., Chang, B.Y.E., Sankaranarayanan, S., Sridharan, M.: The flow-insensitive precision of Andersen's analysis in practice. In: Proceedings of the 18th International Conference on Static Analysis. pp. 60–76. SAS '11, Springer, Germany (2011). https://doi.org/10.1007/978-3-642-23702-7_9

7. Braibant, T., Jourdan, J.H., Monniaux, D.: Implementing and reasoning about hash-consed data structures in Coq. Journal of Automated Reasoning **53**(3), 271–304 (2014). https://doi.org/10.1007/s10817-014-9306-0
8. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications. pp. 243–262. OOPSLA '09, ACM, USA (2009). https://doi.org/10.1145/1640089.1640108
9. Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y.: MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX Security Symposium. pp. 2325–2342. USENIX Security '20 (2020)
10. Cheng, X., Wang, H., Hua, J., Xu, G., Sui, Y.: DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Transactions on Software Engineering and Methodology (TOSEM) **30**(3), 1–33 (2021). https://doi.org/10.1145/3436877
11. Choi, W., Choe, K.M.: Cycle elimination for invocation graph-based context-sensitive pointer analysis. Information and Software Technology **53**(8), 818–833 (2011). https://doi.org/10.1016/j.infsof.2011.03.003
12. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods. pp. 233–247. SEFM '12, Springer, Germany (2012). https://doi.org/10.1007/978-3-642-33826-7_16
13. Fähndrich, M., Foster, J.S., Su, Z., Aiken, A.: Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. pp. 85–96. PLDI '98, ACM, USA (1998). https://doi.org/10.1145/277650.277667
14. Fan, X., Sui, Y., Liao, X., Xue, J.: Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 329–340. ISSTA '17, ACM, USA (2017). https://doi.org/10.1145/3092703.3092729
15. Farkhani, R.M., Jafari, S., Arshad, S., Robertson, W., Kirda, E., Okhravi, H.: On the effectiveness of type-based control flow integrity. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 28–39. ACSAC '18, ACM, USA (2018). https://doi.org/10.1145/3274694.3274739
16. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Proceedings of the 2006 Workshop on ML. pp. 12–19. ML '06, ACM, USA (2006). https://doi.org/10.1145/1159876.1159880
17. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. ACM Transactions on Software Engineering Methodology **17**(2) (May 2008). https://doi.org/10.1145/1348250.1348255
18. Gosling, J., Joy, B., Steele, G.L., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edn. (2014)
19. Goubault, J.: Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. Journées Francophones des Langages Applicatifs pp. 222–238 (1994)
20. Hardekopf, B., Lin, C.: The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 290–299. PLDI '07, ACM, USA (2007). https://doi.org/10.1145/1250734.1250767

21. Hardekopf, B., Lin, C.: Exploiting pointer and location equivalence to optimize pointer analysis. In: International Static Analysis Symposium. pp. 265–280. SAS '07, Springer, Germany (2007). https://doi.org/10.1007/978-3-540-74061-2_17
22. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 226–238. POPL '09, ACM, USA (2009). https://doi.org/10.1145/1480881.1480911
23. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 289–298. CGO '11, IEEE Computer Society, USA (2011). https://doi.org/10.1109/CGO.2011.5764696
24. Hash consing. https://en.wikipedia.org/wiki/Hash_consing (2020)
25. Heintze, N.: Analysis of large code bases: The compile-link-analyze model (1999), http://web.archive.org/web/20050513012825/http://cm.bell-labs.com/cm/cs/who/nch/cla.ps, unpublished
26. Heintze, N., Tardieu, O.: Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 254—263. PLDI '01, ACM, USA (2001). https://doi.org/10.1145/378795.378855
27. Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T., Monfort, V., Pichardie, D., Turpin, T.: Sawja: Static analysis workshop for Java. In: Formal Verification of Object-Oriented Software. pp. 92–106. Springer, Germany (2011). https://doi.org/10.1007/978-3-642-18070-5_7
28. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization. p. 75. CGO '04, IEEE Computer Society, USA (2004). https://doi.org/10.1109/CGO.2004.1281665
29. Lei, Y., Sui, Y.: Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In: International Static Analysis Symposium. pp. 27–47. SAS '19, Springer, Germany (2019). https://doi.org/10.1007/978-3-030-32304-2_3
30. Lhoták, O., Chung, K.C.A.: Points-to analysis with efficient strong updates. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 3–16. POPL '11, ACM, USA (2011). https://doi.org/10.1145/1926385.1926389
31. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using SPARK. In: Proceedings of the 12th International Conference on Compiler Construction. pp. 153–169. CC '03, Springer, Germany (2003)
32. Livshits, V.B., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in C programs. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 317–326. ESEC/FSE '11, ACM, USA (2003). https://doi.org/10.1145/940071.940114
33. Manevich, R., Ramalingam, G., Field, J., Goyal, D., Sagiv, M.: Compactly representing first-order structures for static analysis. In: Proceedings of the 9th International Symposium on Static Analysis. pp. 196–212. SAS '02, Springer, Germany (2002). https://doi.org/10.5555/647171.716101
34. Pearce, D.J., Kelly, P.H., Hankin, C.: Online cycle detection and difference propagation for pointer analysis. In: Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation. pp. 3–12. SCAM '03, IEEE Computer Society, USA (2003). https://doi.org/10.1109/SCAM.2003.1238026

35. Pearce, D.J., Kelly, P.H., Hankin, C.: Efficient field-sensitive pointer analysis of C. ACM Transactions on Programming Languages and Systems **30**(1), 4:1–4:42 (Nov 2007). https://doi.org/10.1145/1290520.1290524
36. Pereira, F.M.Q., Berlin, D.: Wave propagation and deep propagation for pointer analysis. In: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 126–135. CGO '09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/CGO.2009.9
37. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for race detection. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 320–331. PLDI '06, ACM, USA (2006). https://doi.org/10.1145/1133981.1134019
38. What is Referential Transparency? https://www.sitepoint.com/what-is-referential-transparency (2017)
39. Rountev, A., Chandra, S.: Off-line variable substitution for scaling points-to analysis. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 47—-56. PLDI '00, ACM, USA (2000). https://doi.org/10.1145/349299.349310
40. Schubert, P.D., Hermann, B., Bodden, E.: PhASAR: An inter-procedural static analysis framework for C/C++. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 393–410. TACAS '19 (2019)
41. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 17–30. POPL '11, ACM, USA (2011)
42. Sondergaard, H., Sestoft, P.: Referential transparency, definiteness and unfoldability. Acta Informatica **27**(6), 505–517 (Jan 1990)
43. Soot. https://github.com/soot-oss/soot (2021)
44. Sridharan, M., Fink, S.J.: The complexity of Andersen's analysis in practice. In: Proceedings of the 16th International Symposium on Static Analysis. pp. 205–221. SAS '09, Springer, Germany (2009). https://doi.org/10.1007/978-3-642-03237-0_15
45. Sui, Y., Cheng, X., Zhang, G., Wang, H.: Flow2Vec: Value-flow-based precise code embedding. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–27 (2020). https://doi.org/10.1145/3428301
46. Sui, Y., Xue, J.: On-demand strong update analysis via value-flow refinement. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 460–473. FSE '16, ACM, USA (2016). https://doi.org/10.1145/2950290.2950296
47. Sui, Y., Xue, J.: SVF: Interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction. pp. 265–266. CC '16, ACM, USA (2016). https://doi.org/10.1145/2892208.2892235
48. Trabish, D., Kapus, T., Rinetzky, N., Cadar, C.: Past-sensitive pointer analysis for symbolic execution. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 197–208. ESEC/FSE '20, ACM, USA (2020). https://doi.org/10.1145/3368089.3409698
49. Trabish, D., Mattavelli, A., Rinetzky, N., Cadar, C.: Chopped symbolic execution. In: Proceedings of the 40th International Conference on Software Engineering. pp. 350–360. ICSE '18, ACM, USA (2018). https://doi.org/10.1145/3180155.3180251
50. The T. J. Watson libraries for analysis (WALA). http://wala.sf.net/ (2021)

51. Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 999–1010. ICSE '20, ACM, USA (2020). https://doi.org/10.1145/3377811.3380386
52. Whaley, J.: Context-Sensitive Pointer Analysis Using Binary Decision Diagrams. Ph.D. thesis, Stanford University, USA (2007)
53. Yan, H., Sui, Y., Chen, S., Xue, J.: Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: Proceedings of the 40th International Conference on Software Engineering. pp. 327–337. ICSE '18, ACM, USA (2018). https://doi.org/10.1145/3180155.3180178
54. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. pp. 145–157. PLDI '04, ACM, USA (2004). https://doi.org/10.1145/996841.996860