Compacting Points-To Sets through Object Clustering

MOHAMAD BARBAR, University of Technology Sydney, Australia and CSIRO's Data61, Australia
YULEI SUI, University of Technology Sydney, Australia

Inclusion-based set constraint solving is the most popular technique for whole-program points-to analysis whereby an analysis is typically formulated as repeatedly resolving constraints between points-to sets of program variables. The set union operation is central to this process. The number of points-to sets can grow as analyses become more precise and input programs become larger, resulting in more time spent performing unions and more space used storing these points-to sets. Most existing approaches focus on improving scalability of precise points-to analyses from an algorithmic perspective and there has been less research into improving the data structures behind the analyses.

Bit-vectors as one of the more popular data structures have been used in several mainstream analysis frameworks to represent points-to sets. To store memory objects in bit-vectors, objects need to mapped to integral identifiers. We observe that this object-to-identifier mapping is critical for a compact points-to set representation and the set union operation. If objects in the same points-to sets (co-pointees) are not given numerically close identifiers, points-to resolution can cost significantly more space and time. Without data on the unpredictable points-to relations which would be discovered by the analysis, an ideal mapping is extremely challenging.

In this paper, we present a new approach to inclusion-based analysis by compacting points-to sets through object clustering. Inspired by recent staged analysis where an auxiliary analysis produces results approximating a more precise main analysis, we formulate points-to set compaction as an optimisation problem solved by integer programming using constraints generated from the auxiliary analysis's results in order to produce an effective mapping. We then develop a more approximate mapping, yet much more efficiently, using hierarchical clustering to compact bit-vectors. We also develop an improved representation of bit-vectors (called core bit-vectors) to fully take advantage of the newly produced mapping. Our approach requires no algorithmic change to the points-to analysis. We evaluate our object clustering on flow-sensitive points-to analysis using 8 open-source programs (>3.1 million lines of LLVM instructions) and our results show that our approach can successfully improve the analysis with an up to 1.83× speed up and an up to 4.05× reduction in memory usage.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → *Data compression.*

Additional Key Words and Phrases: staged points-to analysis, points-to sets, bit-vectors, hierarchical clustering

ACM Reference Format:
Mohamad Barbar and Yulei Sui. 2021. Compacting Points-To Sets through Object Clustering. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 159 (October 2021), 27 pages. https://doi.org/10.1145/3485547

1 INTRODUCTION

Points-to analysis is a static program analysis which determines the possible memory objects each pointer may point to at runtime. Ideally an analysis is sound, meaning no objects are missing

Authors' addresses: Mohamad Barbar, mohamad.barbar@student.uts.edu.au, University of Technology Sydney, Sydney, NSW, Australia and CSIRO's Data61, Sydney, NSW, Australia; Yulei Sui, University of Technology Sydney, Sydney, NSW, Australia.

2475-1421/2021/10-ART159
https://doi.org/10.1145/3485547

from points-to sets, and precise, meaning no spurious objects appear in points-to sets, but this is exceedingly difficult. Typically, for whole-program analyses, we build off a sound base, subject to some soundiness [Livshits et al. 2015], and gradually add precision as we discover new ways to scale the resulting more expensive analyses.

In recent years there has been a strong focus on improving scalability of precise whole-program points-to analyses from an algorithmic perspective by, for example, solving constraints more efficiently [Hardekopf and Lin 2007a; Lei and Sui 2019; Pereira and Berlin 2009], merging pointers with equivalent points-to sets [Barbar et al. 2021; Hardekopf and Lin 2007b, 2011; Smaragdakis et al. 2013], selectively choosing precision [Lhoták and Chung 2011; Smaragdakis et al. 2011], parallelisation [Liu et al. 2019; Méndez-Lojo et al. 2010], and applying machine learning [Jeon et al. 2018, 2020], amongst other techniques. On the other hand, there has been less research into improving the data structures behind points-to analysis [Berndl et al. 2003; Bravenboer and Smaragdakis 2009; Hardekopf and Lin 2009; Lhoták and Hendren 2003] which is what we focus on in this work.

Points-to analysis operates on points-to sets and makes heavy use of set operations, most notably the union operation. The set operations performed become more expensive and grow in number as analyses become more precise and as input programs become larger (adding large numbers of memory objects and pointers). This is in addition to the space requirements to store these sets increasing. In terms of computing and representing these sets, a few data structures have been used, including (sparse) bit-vectors or bit-sets [Hardekopf and Lin 2009, 2011; Lhoták and Hendren 2003], binary decision diagrams (BDDs) [Berndl et al. 2003; Whaley 2007], and various explicit representations like B-trees [Bravenboer and Smaragdakis 2009] and hash-based sets [Lhoták and Hendren 2003].

Bit-vectors, as one of the more popular data structures for points-to analysis, have been widely used for solving constraints for high-level points-to analysis algorithms in several frameworks such as Soot [Lhoták and Hendren 2003], WALA [WALA 2021], and LLVM-based static analysis tools [Hardekopf and Lin 2011; Schubert et al. 2019; Sui and Xue 2016b]. As studied by previous literature, bit-vectors have been shown to be more efficient than hash-based sets and sorted arrays [Lhoták and Hendren 2003], and BDDs [Hardekopf and Lin 2009]. BDD-based points-to analysis often requires variable reordering to be efficient but this is not necessarily easy to exploit and creating efficient BDDs can be extremely time consuming. Thus the benefits may not outweigh using explicit representations such as B-trees [Bravenboer and Smaragdakis 2009]. Furthermore, non-trivial algorithmic changes are required [Berndl et al. 2003; Zhu and Calman 2004]. By contrast, bit-vectors are much more easily integrated into points-to analyses since bit-vectors act like standard (integral) sets. Without any algorithmic change, only a bijective mapping of abstract memory objects to integral identifiers is required and points-to sets would be made up of the identifiers which map to the pointed-to objects.

Bit-vectors are contiguous arrays of words (the native size which the instruction set architecture (ISA) operates on). Each bit in a word represents an integral identifier and a set bit means that identifier is in the data structure. Concretely, the first bit of the first word represents 0, the second bit of the first word represents 1, and so on. For example, given a word size of 4 bits ($\langle \times\times\times\times \rangle$ where each × represents a bit of the word), as a bit-vector, the set $\{0, 3, 8, 9\}$ can be represented with an array of three words:

$$[\,\langle 1001\rangle, \langle 0000\rangle, \langle 1100\rangle\,].$$
$$\{\;\;0\quad\;3\qquad\qquad\qquad 8\,9\qquad\}$$

In this instance, the bits representing 0, 3, 8, and 9, are set to 1, and the remaining bits are 0. This efficiency in representing identifiers using bits, rather than entire words differentiates bit-vectors from explicitly represented sets. Bit-vectors are more efficient than their explicit counterparts in space in cases where redundant (zero-)words are few. Here, we use 3 words, rather than 4 as in an explicitly represented data structure like an array or hash-based set, to represent the 4 identifiers. Operations on bit-vectors are also efficient; the union operator, for example, becomes a linear bitwise-or between every pair of words in the same position in two bit-vectors. Furthermore, bit-vectors can very trivially take advantage of spatial locality and vectorisation.

Unfortunately, in points-to analysis, zero-words can often occur. Even, in our example above, the second (zero-)word is completely redundant. Worse still, consider the amount of redundant zero-words required for a very plausible points-to set like $\{2, 5000\}$ which would make such a representation less practical. Sparse bit-vectors can solve this problem by omitting non-zero words and linking words through a linked list. Each non-zero word is paired with an offset representing the integral identifier which the first bit of that word refers to. Our example above would become

$$\{\, 0\ \langle 1001 \rangle \,\} \rightarrow \{\, 8\ \langle 1100 \rangle \,\} \rightarrow \mathit{nil},$$

thus avoiding maintaining the zero-word. Despite removing redundant zero-words, sparse bit-vectors introduce other problems like the loss of spatial locality and vectorisation opportunity, and the need for extra metadata in the way of links and offsets. We observe that the downsides of using either type of bit-vector for points-to analysis can be substantially minimised if the objects in the same points-to sets (or co-pointees) have numerically close identifiers in the object-to-identifier mapping. That is, we want to make full use of each word, i.e., reduce the number of zero-**bits**, to improve both types of bit-vectors. For example, the objects being mapped to 0, 3, 8, and 9, would be ideally mapped as 0, 1, 2, 3, requiring only a single word in a sparse or contiguous bit-vector.

Developing such an ideal mapping is worthwhile since it can produce more compact points-to sets, improving both sparse and contiguous bit-vectors, and boosting the performance of any points-to analysis built upon them. However, points-to relations are unpredictable making it extremely challenging to predict an ideal object-to-identifier mapping that accommodates every pointer's points-to set whereby objects within the same points-to set have numerically close identifiers.

We require some indication of the points-to relations which will be discovered by the analysis. Inspired by recent advances in staged analysis (where an expensive main analysis is staged by a cheap auxiliary analysis), this paper *clusters pointees* such that co-pointees are given numerically close identifiers by leveraging the freely available points-to sets produced by the already-run auxiliary analysis. We evaluate our approach on a staged flow-sensitive analysis [Hardekopf and Lin 2011], and despite the auxiliary analysis being an over-approximation of the main analysis, we find that a mapping that works well for the auxiliary analysis also works well for the main analysis. The result is that bit-vectors (and sparse bit-vectors) are smaller, using far fewer words, making representation more compact and operations faster, improving the main analysis in space and time.

First, we formulate our approach as an integer programming problem, optimising for compact points-to sets, by generating constraints from the auxiliary analysis's results in order to produce our mapping. However, integer programming is expensive, so we explore agglomerative clustering, or bottom-up hierarchical clustering, a more approximate and highly efficient solution. Popular for data mining [Maimon and Rokach 2005], agglomerative clustering, helps us produce a good mapping with negligible overhead. Additionally, we also develop the core bit-vector, an improved version of the contiguous bit-vector, to fully take advantage of the mappings produced by our clustering approach. We evaluate our object clustering on a precise flow-sensitive points-to analysis using 8 open-source programs. Our results show that our approach can successfully reduce the number of words required for sparse and contiguous bit-vectors by up to 6.75× (4.94× on average) and up to 13.03× (5.66× on average), respectively. Analysis time improves by up to 1.83× (1.62× on average) and up to 1.74× (1.35× on average), and memory usage is reduced by up to 3.04× (2.43× on average) and up to 4.05× (2.35× on average) for sparse and contiguous bit-vectors respectively.

In summary, our contributions are:

- The clustering of pointees using hierarchical clustering upon the data produced by an auxiliary analysis to reduce the time and space overhead of storing and operating on points-to sets.
- A formulation of points-to set compaction as an optimisation problem solved by integer programming.
- The core bit-vector, an improved contiguous bit-vector.
- An evaluation of our approach using a staged flow-sensitive analysis on 8 programs which shows that our approach significantly reduces required words and finds up to 1.83× improvement in time and up to 4.05× improvement in space.

2 BACKGROUND AND MOTIVATION

We first introduce some brief background on inclusion-based points-to analysis, both flow-insensitive and -sensitive, and how staging works in Section 2.1. We then motivate the importance of our work in Section 2.2 by presenting our observations on the prevalence of union operations for precise flow-sensitive analysis, which is also the main target analysis for evaluating our approach.

2.1 Inclusion-Based Points-To Analysis

Inclusion-based set constraint solving is the most commonly used technique for whole-program points-to analysis. An analysis is typically formalised as a set of inference rules which generate constraints between points-to sets based on various program statements. The constraints are resolved by performing set unions until a fixed-point is reached.

In this work, we use points-to analysis for C/C++ to evaluate our approach. Based on partial-SSA form [Lattner and Adve 2004], following previous C/C++ points-to analyses [Balatsouras and Smaragdakis 2016; Hardekopf and Lin 2009, 2011; Kuderski et al. 2019; Lhoták and Chung 2011; Sui and Xue 2016a], we perform the analysis on an LLVM-like instruction set, where the set of all variables $\mathcal{V} = \mathcal{O} \cup \mathcal{P}$ is made up of two sets, $\mathcal{O}$, which represents all possible abstract objects and their fields (*address-taken variables*), and $\mathcal{P}$, which represents all stack and global pointers (*top-level pointers*). Top-level pointers are explicit in contrast to address-taken variables which are implicit and accessed indirectly at Store and Load instructions through top-level pointers.

Given $p, q \in \mathcal{P}$ and $o \in \mathcal{O}$, there are five basic instructions: Alloc $p = alloc_o$ which represents allocation of memory (abstracted into a memory object o), Copy $p = q$ which represents assignment between two top-level pointers, Field $p = \&q \rightarrow f_i$ which denotes access of the i-th field (f_i) of the object which q points to, Load $p = *q$ which represent assignment from a dereferenced pointer, and Store $*p = q$ which represents assignment to a dereferenced pointer. Given this set of instructions, Figure 1 presents the inference rules of a field-sensitive and flow-insensitive points-to analysis, commonly referred to as Andersen's analysis [Andersen 1994]. Each instruction (aside from the simpler Alloc and Field instructions) is analysed by solving a set constraint between two points-to sets, e.g., $pt(q) \subseteq pt(p)$, which represents the *inclusion* of points-to set $pt(q)$ in $pt(p)$. The size of a points-to set will increase monotonically until a fixed-point is reached. A Load or Store instruction can derive many more set unions based on the newly discovered points-to targets of the dereferenced pointer during iterative constraint solving.

A more precise analysis implies more set union operations. Figure 2 gives the rules of a typical inclusion-based flow-sensitive analysis. Unlike flow-insensitive analysis which computes a single points-to set solution for each variable across the whole program, flow-sensitive analysis promises more precise points-to results by computing and maintaining the points-to sets of each object

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

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

Fig. 1. Inference rules for a field-sensitive and flow-insensitive inclusion-based points-to analysis.

o before $(pt[\overline{\ell}](o))$ and after $(pt[\overline{\ell}](o))$ each instruction ℓ. This greatly increases the number of points-to sets and the number of constraints generated. Note that we do not need to associate a program-point with top-level pointers since they are defined only once in LLVM's partial-SSA form [Lattner and Adve 2004]. Another source of precision is the `[SU/WU]` rule, which along with the *kill* function, gives us standard strong updates for singleton objects [Lhoták and Chung 2011]. Finally, through the `[CONTROL-FLOW]` rule, the points-to sets of memory objects are propagated along control-flow edges ($\ell \longrightarrow \ell'$) in the control-flow graph (CFG), thus introducing many new union operations that a flow-insensitive analysis does not perform.

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

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

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

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

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

State-of-the-art flow-sensitive analysis is performed as a *staged analysis* [Barbar et al. 2021; Hardekopf and Lin 2011], which builds a def-use graph, a sparse representation of data-flow built upon the control-flow graph. Points-to sets are then propagated on pre-computed def-use edges rather than control-flow edges to reduce the number of points-to set propagations along control-flow. This technique uses a less precise but cheap auxiliary analysis (e.g., Andersen's analysis) which over-approximates the more precise main flow-sensitive analysis to bootstrap that analysis. The main insight is that since the points-to results of the auxiliary analysis will always be more conservative ($pt(p)$ in the auxiliary analysis is a superset of $pt(p)$ in the main analysis), these results can be used to determine properties of the main analysis like a sound def-use graph. Despite staging, increasingly precise analyses still remain very costly for their increase in union operations and stored points-to sets. In the case of flow-sensitive points-to analyses, the main analysis can be orders of magnitude slower than the flow-insensitive analysis used to stage it (standard Andersen's analysis), though it is still much faster than an ordinary unstaged flow-sensitive analysis.

Table 1. Statistics for program points, variables and set union operations for flow-sensitive analysis.

Program	Program Points	Top-Level Variables	Memory Objects	Average PTS	Largest PTS	SFS Unions
dhcpcd	57 168	63 196	3701	223.55	265	90 785 902
gawk	279 931	141 136	4784	434.81	811	2 275 388 148
bash	254 314	149 070	4339	244.57	324	531 039 266
mutt	360 535	178 147	7169	379.47	1238	1 347 064 278
lynx	579 285	237 252	7917	198.02	1129	4 829 162 478
xpdf	440 418	388 859	19 101	87.78	1590	12 423 325 697
ruby	795 643	670 649	20 235	2.68	1726	13 502 095 022
keepassxc	572 001	604 363	37 671	171.70	365	879 430 055

2.2 Prevalence of Union Operations

The union operation is central to whole-program points-to analysis. The time spent performing union operations to analyse a program depends on the size of the program and the precision of the points-to analysis. A larger program usually contains more objects and thus has larger and more points-to sets, incurring more overhead per union and increasing the number of unions. A more precise analysis can have many more union operations than an imprecise one, for example flow-sensitive analysis computes points-to sets at each program point while its flow-insensitive counterpart only computes a single solution for each variable.

To illustrate these concerns, we ran a flow-sensitive inclusion-based analysis staged by a flow-insensitive Andersen's analysis (SFS [Hardekopf and Lin 2011] implemented in SVF [Sui and Xue 2016b]) on 8 open source programs and recorded the number of unions required in the main stage. We include programs across a wide range of domains, including email, programming languages, and password management. The exact details of the algorithm, setup, and programs are listed in Section 5. We show the number of program points, top-level pointers, and abstract memory objects each program contains as well as the the average and maximum points-to set sizes, and the number of unions each analysis performs in practice in Table 1. From Table 1, it can be gleaned that operating on points-to sets is not trivial considering how large they can be and how many times the union operation is invoked. Analysing `xpdf` and `ruby`, for example, require around 13 billion unions each, and so any improvement to the union operation can have a considerable impact.

3 REPRESENTING POINTS-TO SETS AS BIT-VECTORS

In this section, we describe the internal workings of contiguous and sparse bit-vectors, and their performance issues when performing a union operation between two points-to sets whose element identifiers are not numerically close as well excess memory costs they may incur. These problems are amplified by the presence of a large number of memory objects as in Table 1.

3.1 Contiguous and Sparse Bit-Vectors

Contiguous bit-vectors are implemented as arrays of words. In the best-case where words are used to the fullest, a bit-vector can represent a set a of i integers within the range 0 to n in $\lceil \frac{i}{W} \rceil$ words, where W is the word size, in bits, of the ISA. In the worst case, that same set would be represented in $\lceil \frac{n}{W} \rceil$ words. Bitwise operations upon the entire bit-vector, like *or* and *not*, could be implemented by performing the operation on individual words (and the corresponding word of the other bit-vector operand in the case of binary operations). Fortunately, many bitwise operations map directly to set operations, like bitwise-or being an analogue to the union operation on a set of integers. Such operations are amenable to vectorisation and can take advantage of the spatial-locality afforded by the contiguous nature of the words. Concretely, the union, as the most important such operation in points-to analysis, for example, of two bit-vectors of size n made up of words $w_1, ..., w_n$ and $w_1', ..., w_n'$, respectively, would be given by,

$$\begin{array}{ccccccc} & [& w_1, & w_2, & \ldots, & w_n &] \\ | & [& w_1', & w_2', & \ldots, & w_n' &] \\ \hline & [& w_1 \mid w_1', & w_2 \mid w_2', & \ldots, & w_n \mid w_n' &]. \end{array}$$

If one bit-vector is longer than the other such that the longer bit-vector contains m more words, the shorter bit-vector can be thought of as being padded with m zero-words since in the resulting bit-vector, the last m words would be verbatim the last m words of the longer bit-vector. It should be noted that some implementations require bit-vectors to be sized the same (like in C++'s `std::bitset` [ISO/IEC 2017]) whilst others like that implemented by LLVM [LLVM-BV 2021] strip trailing zero-words by maintaining the length of the bit-vector as extra metadata. When we use the term bit-vector or standard bit-vector, we refer to the latter and only implicitly represent the length.

To use bit-vectors as points-to sets, we need to assign abstract memory objects, our pointees, to integral identifiers. For example, consider we have 10000 memory objects (o_0 to o_{9999}), a quantity not foreign when analysing typical programs, and that the points-to sets of some pointers p and q, at some point during the analysis, are

$$\begin{aligned} pt(p) &= \{o_1, o_{4500}, o_{9999}\}, \text{ and} \\ pt(q) &= \{o_1, o_4, o_8\}. \end{aligned}$$

And let us consider a naive mapping of each object to its subscript such that we have $o_n \mapsto n$. As bit-vectors, our two points-to sets would be represented as

$$\begin{aligned} pt(p) &= [\ \langle 01_100\rangle, \langle 0000\rangle, \ldots, \langle 0000\rangle, \langle 1_{4500}000\rangle, \langle 0000\rangle, \ldots, \langle 0000\rangle, \langle 0001_{9999}\rangle\ ], \text{ and} \\ pt(q) &= [\ \langle 01_100\rangle, \langle 1_4000\rangle, \langle 1_8000\rangle\ ], \end{aligned}$$

where the word size W equals 4 and we subscript set bits with the identifier they correspond to for readability. We see that $pt(p)$ is impractically large, requiring 2500 words, or 10000 bits, for just 3 pointees. This is precisely the worst case of $\lceil \frac{n}{W} \rceil = \lceil \frac{10000}{4} \rceil$ where the best case would be $\lceil \frac{3}{4} \rceil$. $pt(q)$ is more reasonable despite still being imperfect, requiring 3 words, or 12 bits, for 3 pointees, despite the 3 pointees being able to fit into 1 word ($\lceil \frac{3}{4} \rceil = 1$). Furthermore, if we were to include $pt(p)$ in $pt(q)$ ($pt(p) \subseteq pt(q)$), $pt(q)$ would grow to be the same size as $pt(p)$, making us then represent both $pt(p)$ and $pt(q)$ with 20000 bits in total for 5 pointees (whose corresponding set bits are spread amongst just 5 words). This becomes wasteful and impractical for representing a points-to set in which identifiers are not numerically close.

Sparse bit-vectors aim to overcome the abundance of zero-words by only storing words which have at least one set bit. Since there are now "holes" in the data structure, we can no longer use contiguous memory and need to mark each word with an offset indicating the integral value which the first bit in that word represents. Such offset/word pairs are often linked together in a linked list to aid arbitrary insertion which is also necessary for inplace union operations.

With the redundant zero-words which can appear in contiguous bit-vectors removed, sparse bit-vectors can represent $pt(p)$ and $pt(q)$ as

$$\begin{aligned} pt(p) &= \{\ 0\ \langle 01_100\rangle\ \} \rightarrow \{\ 4500\ \langle 1_{4500}000\rangle\ \} \rightarrow \{\ 9996\ \langle 0001_{9999}\rangle\ \} \rightarrow \mathit{nil}, \text{ and} \\ pt(q) &= \{\ 0\ \langle 01_100\rangle\ \} \rightarrow \{\ 4\ \langle 1_4000\rangle\ \} \rightarrow \{\ 8\ \langle 1_8000\rangle\ \} \rightarrow \mathit{nil}, \end{aligned}$$

thus requiring 3 words per bit-vector (and some extra metadata in the form of links and offsets). Representation of $pt(p)$ has improved drastically, but that of $pt(q)$ has become worse when we account for the extra metadata in the way of offsets and links. However, if we were to fulfil our $pt(p) \subseteq pt(q)$ constraint, then the resulting $pt(q)$'s representation has improved as it would only require 5 words (and some metadata) to represent, not 2500 words. Of the disadvantages of representing reasonably sized bit-vectors as sparse bit-vectors is the loss of vectorisation, spatial locality, the extra logic required to perform operations, and the extra metadata required. The union operation, presented in Algorithm 1, is more complicated and is now riddled with conditionals (e.g., lines 3 and 6), indirect references (e.g., lines 5 and 8), and heap allocations for nodes (e.g., line 4). Thus, in practice, sparse bit-vectors can be less efficient than their contiguous counterpart in addition to the extra memory metadata can take up in cases where there would not have been many zero-words.

Algorithm 1: Union of two sparse bit-vectors.

```
   Input  :  $s_1, s_2$ – sparse bit-vectors (linked lists of
                  $\{offset, word, next\}$ structures).
   Output :  $u$ – union of $s_1$ and $s_2$.
 1 $c_1 \leftarrow s_1$; $c_2 \leftarrow s_2$;
 2 while $c_1 \neq nil \wedge c_2 \neq nil$ do
 3     if $c_1.offset = c_2.offset$ then
 4         $u$.append(new { $c_1.offset$ $\langle c_1.word \mid c_2.word \rangle$ });
 5         $c_1 \leftarrow c_1.next$; $c_2 \leftarrow c_2.next$;
 6     else if $c_1.offset < c_2.offset$ then
 7         $u$.append(new { $c_1.offset$ $\langle c_1.word \rangle$ });
 8         $c_1 \leftarrow c_1.next$;
 9     else
10         $u$.append(new { $c_2.offset$ $\langle c_2.word \rangle$ });
11         $c_2 \leftarrow c_2.next$;
12     end
13 end
   // $c_1$ or $c_2$ may not yet be $nil$; append remainder.
14 while $c_1 \neq nil$ do
15     $u$.append(new { $c_1.offset$ $\langle c_1.word \rangle$ });
16     $c_1 \leftarrow c_1.next$;
17 end
18 while $c_2 \neq nil$ do
19     $u$.append(new { $c_2.offset$ $\langle c_2.word \rangle$ });
20     $c_2 \leftarrow c_2.next$;
21 end
```

The disadvantages of using both contiguous and sparse bit-vectors are amplified by a poor object-to-identifier mapping. Due to the unpredictability of points-to relations, an object-to-identifier mapping better than ensuring that the n memory objects of a program are given identifiers 0 to n is difficult. If we assume that the objects present in $pt(p)$ and $pt(q)$ are not present in any other points-to set it would be ideal to remap these objects to identifiers which are close to each other. For example, consider the mapping

$$o_1 \mapsto 0 \quad o_{4500} \mapsto 1 \quad o_{9999} \mapsto 2 \quad o_4 \mapsto 3 \quad o_8 \mapsto 4,$$

and the subsequent representation as bit-vectors rather than sparse bit-vectors:

$$pt(p) = [\,\langle 1_0\,1_1\,1_2\,0\rangle\,], \text{and}$$
$$pt(q) = [\,\langle 1_0\,0\,0\,1_3\rangle, \langle 1_4\,0\,0\,0\rangle\,].$$

Now, we have represented $pt(p)$ with a single word, as a bit-vector, and $pt(q)$ with 2 words, also as a bit-vector. After fulfilling the $pt(p) \subseteq pt(q)$ constraint, the result would remain compact ($pt(q) = [\,\langle 1_0\,1_1\,1_2\,1_3\rangle, \langle 1_4\,0\,0\,0\rangle\,]$), and in fact would be a best-case representation, yielding much better performance.

However, in reality, these objects may occur in multiple points-to sets and have requirements of closeness to many other objects such that achieving an ideal mapping is far more complicated. To address the above challenges, we explore techniques like integer programming and hierarchical clustering which can find a superior mapping by leveraging the over-approximate points-to data produced by the auxiliary analysis. But first, we briefly describe an improved contiguous bit-vector that can better take advantage of a better object-to-identifier mapping.

3.2 Core Bit-Vector

Let us take a look at the *core bit-vector*, our improved version of standard bit-vector. It has the compact representation like a contiguous bit-vector (little metadata) while reducing many zero-words by stripping leading zero-words. Together with sparse bit-vectors, core bit-vectors will be used for evaluating our object-to-identifier mappings for points-to analysis in Section 5.

Consider $W = 4$ and a points-to set with 5 objects whose identifers map to 9995, 9996, 9997, 9998, and 9999. These identifiers would ideally fit within two words. As a sparse bit-vector, this points-to set is

$$\{\,9992\ \langle 0\,0\,0\,1_{9995}\rangle\,\} \to \{\,9996\ \langle 1_{9996}\,1_{9997}\,1_{9998}\,1_{9999}\rangle\,\} \to nil,$$

using just two words and associated metadata. However, as a bit-vector, this points-to set would look like

$$[\,\langle 0000\rangle, \ldots, \langle 0000\rangle, \langle 0\,0\,0\,1_{9995}\rangle, \langle 1_{9996}\,1_{9997}\,1_{9998}\,1_{9999}\rangle\,],$$

which contains 2500 words with only two meaningful words. The problem is that bit-vectors have an explicit representation (of zero) for the integers 0 to the first non-zero bit, no matter how far that may be.

To remedy this we use *core bit-vectors* which use an offset similar to sparse bit-vectors to state the bit of the first word and strips leading and trailing zero words (to maintain only the “core”). However, unlike a sparse bit-vector, zero words may occur between the first and last (non-zero by definition) words. For example, the points-to set above would be represented as

$$\{\,9992\ [\,\langle 0\,0\,0\,1_{9995}\rangle, \langle 1_{9996}\,1_{9997}\,1_{9998}\,1_{9999}\rangle\,]\,\}$$

saving us 2498 words. If we were to insert identifier 9988, it would become

$$\{\,9988\ [\,\langle 1_{9988}\,0\,0\,0\rangle, \langle 0\,0\,0\,1_{9995}\rangle, \langle 1_{9996}\,1_{9997}\,1_{9998}\,1_{9999}\rangle\,]\,\}$$

which uses the same number of words as an equivalent sparse bit-vector but has the benefit of storing words contiguously and with less metadata – the offset and the (implicit) length – therefore potentially saving both space and time. Algorithm 2 details the union operation of two core bit-vectors. In contrast to Algorithm 1, operations are all performed on contiguous arrays, rather than across indirect links, and loops do not contain conditionals, making the process amenable to vectorisation. When the offset is set to 0, the data structure acts as a standard contiguous bit-vector.

Algorithm 2: Union of two core bit-vectors.

```
Input  :  $c_1$, $c_2$ – core bit-vectors ({offset, [words]}).
Output :  $u$ – union of $c_1$ and $c_2$.
// Determine which input starts "earlier" and which "later".
if $c_1.offset < c_2.offset$ then
    $e \leftarrow c_1$; $l \leftarrow c_2$;
else
    $e \leftarrow c_2$; $l \leftarrow c_1$;
end
$u.offset \leftarrow e.offset$;
$ei \leftarrow 0$; // Index into earlier input.
while $ei < \left\lfloor \frac{l.offset}{\mathcal{W}} \right\rfloor - \left\lfloor \frac{e.offset}{\mathcal{W}} \right\rfloor$ do
    if $ei < e.words.length$ then
        // Append words in $e$ which are not in $l$.
        $u$.append($e.words[ei]$);
    else
        // Nothing left in $e$; add zero-words till start of $l$.
        $u$.append(0);
    end
    $ei \leftarrow ei + 1$;
end
$li \leftarrow 0$; // Index into later input.
// Append union of words common to both inputs.
while $ei < e.words.length \land li < l.words.length$ do
    $u$.append($e.words[ei] \mid l.words[li]$);
    $ei \leftarrow ei + 1$; $li \leftarrow li + 1$;
end
// $ei$ or $li$ may not have reached end; append remainder.
while $ei < e.words.length$ do
    $u$.append($e.words[ei]$);
    $ei \leftarrow ei + 1$;
end
while $li < l.words.length$ do
    $u$.append($l.words[li]$);
    $li \leftarrow li + 1$;
end
```

4 COMPACTING POINTS-TO SETS

Assigning memory objects to identifiers based on points-to relations can be framed as an optimisation problem, making integer programming (IP) well suited. Unfortunately, the resulting integer programs are computationally expensive to solve, but the formulation helps clarify the problem and its optimal solution. We first formulate compacting points-to sets as an optimisation problem solved by IP in Section 4.1. Then, we present a more approximate and much more efficient approach in Section 4.2 where we provide a brief introduction to hierarchical clustering with an example which clusters items located at 2D coordinates, then detail how we can apply this technique to objects in points-to sets, and conclude with two simple techniques to improve efficiency and the resulting mapping.

4.1 Integer Programming Formulation

It is hard to determine a good identifier mapping before any points-to analysis is performed as it is unknown which objects are pointed to by the same pointers and should thus be assigned close identifiers. Inspired by staged analysis [Hardekopf and Lin 2011], we use an auxiliary analysis as a good indication of which objects may occur in the same points-to sets, given the insight that the points-to sets of auxiliary analysis always represent supersets of those in the main phase. Therefore, creating a mapping for the auxiliary analysis should create a good mapping for the main analysis, at the very least, better than a random mapping. All of our variables below are integers.

We let the constant $\mathcal{W}$ be the word size of the ISA (in bits), and for each points-to set $P = \{o_{x_1}, o_{x_2}, \ldots, o_{x_n}\}$ produced by the auxiliary analysis, we have the following known variables:

(1) n is the number of objects in P.
(2) $w = \lceil \frac{n}{\mathcal{W}} \rceil$ is the minimum number of words required for a bit-vector representation of P in the, potentially impossible overall, best case scenario.

And the following unknown variables:

(1) m_{x_i}, where $1 \leq i \leq n$, is the identifier object o_{x_i} will be assigned to in our new mapping.
(2) f is some offset multiplier for where the identifiers, m_{x_i}, start.

Our goal is to produce mappings from o_{x_i} to m_{x_i} such that the least possible number of words can represent all points-to sets (as close as possible to each w), thus reducing both space and time overhead with a more compact bit-vector representation and a more efficient bit-wise union operation. Ideally, we will be using core bit-vectors. In the best case scenario, each pair of objects in each points-to set are assigned identifiers within w of each other, that is, $\frac{\left|m_{x_i} - m_{x_j}\right|}{\mathcal{W}} < w$ for $1 \leq i \leq n$ and $1 \leq j \leq n$, with the minimum m_{x_i} assigned to a multiple of $\mathcal{W}$, i.e., $f \cdot \mathcal{W}$. That is, the minimum m_{x_i} is aligned on a word boundary. Aligning to $\mathcal{W}$ is important to avoid co-pointees being mapped to identifiers within w of each other, but needlessly crossing word boundaries. For example, given $\mathcal{W} = 4$ and $pt = \{o_1, o_2\}$, it would be unoptimal to assign o_1 to 7 and o_2 to 8 despite that $\frac{|m_1 - m_2|}{\mathcal{W}} = \frac{|7-8|}{4} < 1$ since we would require 2 words in a bit-vector instead of 1 (which can be achieved with various assignments like that of o_1 to 2 and o_2 to 3, for example). This produces the best possible allocation because it ensures points-to sets, as bit-vectors, are as small as they can possibly be (of size w), if the problem allows.

We need to determine the best possible values for m and f to achieve this. As an integer program, we can encode these requirements as the following constraints, for each points-to set $P = \{o_{x_1}, o_{x_2}, \ldots, o_{x_n}\}$:

$$\begin{aligned} m_{x_i} &\geq f_P \cdot \mathcal{W} \\ m_{x_i} &< f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W} \\ f_P &\geq 0, \end{aligned} \tag{C1}$$

which ensures that all objects in a points-to set are mapped to identifiers between some starting offset and within the minimum number of words required to represent that points-to set. We also require that objects are uniquely mapped to identifers, so for all pair of identifiers m_i, m_j, where $i \neq j$,

$$\left|m_i - m_j\right| > 0, \tag{C2}$$

which can be expressed as the constraints

$$\begin{aligned} m_i - m_j &< L \cdot b_{ij}, \\ m_i - m_j &> -L + L \cdot b_{ij} \\ b_{ij} &\geq 0 \\ b_{ij} &\leq 1, \end{aligned} \tag{C3}$$

where L is set to be larger than any possible identifier, and b_{ij} are boolean variables (as forced by the constraints upon them). This construct asserts that $m_i - m_j$ must never be zero, i.e., that they are not equal. When b_{ij} is 0, the first two constraints ensure the difference is in the open range $(-L, 0)$, and when set to 1, they ensure the difference is in the open range $(0, L)$.

Unfortunately, such an integer program may be impossible to solve due to the pigeonhole principle. Consider that an object o (which we will map to m) occurs in $\mathcal{W}$ points-to sets, none of which contain common elements aside from o, each of which with $w = 1$ and $n > 1$. It would not be possible for the m corresponding to o to be assigned within $\mathcal{W}$ of all of the identifiers assigned to o's co-pointees in the same word. Rather, it can only be in the same word as $\mathcal{W} - 1$ of its co-pointees. The $\mathcal{W}$th co-pointee must be, at best, in a word before or after that which m falls into. To remedy this, we provide a "tolerance" for each points-to set to extend the range of identifiers the objects of a points-to set can fall in. Thus we introduce another unknown for each points-to set:

(3) t is a tolerance multiplier for where the identifiers assigned to objects of a points-to set can end.

Thus, for each pointee o_{x_i} in points-to set $P = \{o_{x_1}, o_{x_2}, \ldots, o_{x_n}\}$, we reframe our constraints as (changes underlined):

$$\begin{aligned} m_{x_i} &\geq f_P \cdot \mathcal{W} \\ m_{x_i} &< f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W} \underline{+ t_P \cdot \mathcal{W}} \\ f_P &\geq 0 \\ t &\geq 0. \end{aligned} \tag{C4}$$

Again, the first constraint ensures that m_{x_i} begins aligned at some offset into the identifier space ($f_P \cdot \mathcal{W}$). The second ensures that it does not extend w words away from the offset ($f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W}$) as before, but now we allow for some tolerance ($t_P \cdot \mathcal{W}$). As before we must also ensure that the identifiers are unique (Constraints (C2)/(C3)). This set of constraints becomes trivial to solve with absurdly large tolerances and circumvents our goal of small points-to sets. Thus, our objective is to minimise the tolerances as they introduce extra words than the (potentially impossible) ideal. That is, we minimise $t_{P_1} + \cdots + t_{P_n}$, where n is the number of points-to sets we take into consideration, which gives us the least required number of words in a core bit-vector[1].

Proof. For each points-to set P, representing P as a core bit-vector uses at best w_P words. The constraints encode an identifier mapping which makes each points-to set P fit into w_P words, **except** that a tolerance for the number of words is allowed. Optimising for minimum tolerance multipliers (i.e. the sum of t_P for all points-to sets P, where each tolerance t_P would then be used in $t_P \cdot W$) ensures that each points-to set is represented with as few words as possible, thus producing an ideal mapping. □

Finally, as mentioned previously, this is too computationally expensive to solve. There are a few ways to make the integer programs produced cheaper to solve, of them is the regioning we

[1] It is important to note that this minimal representation is for the points-to sets produced by the auxiliary analysis which may differ from that produced by the main stage.

introduce to improve clustering in Section 4.4 which allows groups of objects to be considered separately, but regardless, IP as a solution to this problem simply does not scale with current techniques for large programs. The remainder of this paper is dedicated to solutions which are more approximate but far more scalable.

4.2 Hierarchical Clustering

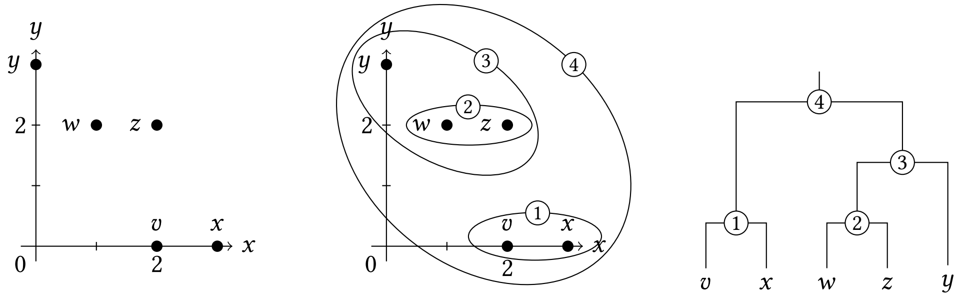
Our aim is to assign pointees which appear in points-to sets together to identifiers which are numerically close together. We find clustering to be a more practical way to achieve this goal. Clustering is the process of grouping close or similar items together given a set of items with some measure of similarity or distance between them like the Jaccard Index for sets [Jaccard 1912] or the Euclidean distance for coordinates.

There is no notion of a "perfect clustering", rather success is determined depending on the problem for which clustering is employed to solve. Hierarchical clustering can be agglomerative (bottom-up) or divisive (top-down) [Maimon and Rokach 2005]. An agglomerative approach starts with clusters of a single item and successively merges clusters until there remains only one cluster, and a divisive approach starts with a single cluster, divides it, and then successively divides clusters until only single-item clusters remain. We focus on the former due to the availability of an extremely performant C++ implementation [Müllner 2013]. Both approaches are hierarchical in that the resulting clusters are made up of smaller clusters. The resulting hierarchy is represented by a dendrogram, a tree where child nodes are the clusters which form parent nodes, also clusters [Maimon and Rokach 2005]. Leaf nodes are the individual objects themselves.

To show how agglomerative clustering works, we walk through a simple example. Consider some items located at coordinates: v at $(2, 0)$, w at $(1, 2)$, x at $(3, 0)$, y at $(0, 3)$, and z at $(2, 2)$, as in Figure 3a, and that we want to cluster these items according to their closeness using the Euclidean distance as a measure for that. The process is shown on the same plane in Figure 3b. For our example, we set the linkage criterion as the *single* linkage criterion [Murtagh 1983] such that the distance between two clusters is the minimum distance between any two items in each of those clusters.

To begin, each item is considered to be part of its own cluster, and we search for the closest two clusters (according to the linkage criterion) and merge them. The pairs v and x, and w and z both have the shortest Euclidean distance between them (1.0), so we randomly pick a pair, say v and x, and merge them as cluster ①. Then we merge the other pair, w and z, as cluster ② since they are now the two closest clusters. The next two closest clusters are now the cluster containing w and z (cluster ②) and the single-item cluster containing y, since w and y have a distance of $\sqrt{(0-1)^2 + (3-2)^2} = 1.41$ between them. They are merged as cluster ③. Now we have only two clusters remaining, so we merge them as ④ since they are obviously the closest two clusters, and thus agglomerative clustering is complete. The dendrogram in Figure 3c shows the resulting clusters for our example. The entire dendrogram represents one cluster, but if we cut it, we can form multiple clusters. For example, if we cut the dendrogram one level down (i.e., one step backwards from the end of the clustering process), we have two clusters, one with v and x, and one with w, y, and z. From this, we can infer that according to our definition of "close" (the linkage criterion and closeness measure), v and x are close and w, y, and z are close, relatively.

4.2.1 Linkage Criteria. In our example, determining the distance between two single-item clusters is straightforward – simply calculate the Euclidean distance – but determining the distance between two clusters where at least one contains more than one object was more complicated and we required a *linkage criterion*. Being that clustering is an approximate process, multiple linkage criteria exist to optimise for different data types. In this work, we focus on three, single-linkage,



(a) Items on a plane.

(b) Agglomerative clustering process with the Euclidean distance. The numbers on the clusters represent the order in which they are created.

(c) Resulting dendrogram.

Fig. 3. Agglomerative clustering process (b) of coordinate data (a) and the resulting dendrogram (c).

complete-linkage, and average-linkage, partly because of the existence of extremely performant implementations of them [Müllner 2013] and partly because of how common they are. Clustering with any of these linkage criteria can be implemented in $O(n^2)$ time in the worst-case given n items to cluster [Maimon and Rokach 2005; Müllner 2013]. For our data, clustering is very fast and so we perform clustering with each of these criteria and choose the one which produces the most promising mapping on-the-fly.

Single-Linkage. The distance between two clusters under the single-linkage criterion is the smallest distance between any member of one cluster to any member of the other [Murtagh 1983]. That is, assuming we have a distance function (for our problem, we use the distance defined in Definition 4.1) for two objects, $d : O \times O \mapsto \mathbb{R}$, the distance between clusters C_1 and C_2, is

$$min\{d(o_1, o_2) \mid o_1 \in C_1, o_2 \in C_2\}.$$

Complete-Linkage. The distance between two clusters under the complete-linkage criterion is the largest distance between any member of one cluster to any member of the other [Murtagh 1983]. This is similar to the single-linkage critera, except that it, very harshly, blocks clusters whose members are not *all* close. Formally, with the same d and clusters C_1 and C_2 as before, this is given by

$$max\{d(o_1, o_2) \mid o_1 \in C_1, o_2 \in C_2\}.$$

Average-Linkage. Unlike single- and complete-linkage, the distance calculated with average-linkage considers all members of both clusters [Murtagh 1983]. The distance given is the average distance from any member of one cluster to any member of the other cluster. For clusters C_1 and C_2 and our distance function d, this is given by

$$\frac{1}{|C_1|\,|C_2|} \sum_{o_1 \in C_1} \sum_{o_2 \in C_2} d(o_1, o_2).$$

4.3 Clustering Objects

With hierarchical clustering, we can cluster objects in such a way that co-pointees of smaller points-to sets are clustered together early, and hence are regarded as "closer" and thus more suitable for close identifiers. A simple linear scan of points-to sets, and assigning identifiers during the scan, is not sufficient because relationships between pointees are not transitive.

For example, consider the two points-to sets $\{o_a, o_b, o_c, o_d, o_e\}$ and $\{o_a, o_f\}$. If we were to perform a linear scan assigning identifiers sequentially starting from the first points-to set, we would produce the mapping

$$o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2 \quad o_d \mapsto 3 \quad o_e \mapsto 4 \quad o_f \mapsto 5,$$

which forces the first points-to set into two words (which is ideal), but unfortunately, also forces the second points-to set into two words (which is not ideal; it can be represented in one word). Clustering better understands relationships between pointees allowing us to produce a better mapping.

To cluster objects, we require a more suitable measure of distance between two objects as we define below.

Definition 4.1. (Object distance). The distance between two objects is the minimum number of words required to represent any points-to set in which both objects appear in as a bit-vector.

This definition encodes how far apart, word-wise, two objects would be in the theoretical best case scenario (which may be impossible; recall our discussion on the pigeonhole principle and tolerances in Section 4.1). We want the clustering process to approximate this theoretical limit as closely as possible.

Revisiting our example, the object distance between o_a and o_f, $d(o_a, o_f)$, is 1 since o_a and o_f appear in two points-to sets, one of which can be represented in 2 words and the other in 1 word, and $min(1, 2) = 1$. $d(o_b, o_f) = d(o_c, o_f) = d(o_e, o_f) = \infty$ since these objects never appear in the same points-to set. The remaining object pairs have a distance of 2 because they appear together only in the first points-to set which requires two words to represent in the ideal case.

With this definition, we can build a distance matrix by calculating the distance for each pair of objects. This can be done by filling the distance matrix with ∞, scanning all points-to sets, and updating the distance between each pair of objects in a points-to set if that points-to set requires fewer words to represent as a bit-vector than is recorded in the distance matrix. For each points-to set, this is a quadratic process in the number of objects in that points-to set. In practice, we find this process extremely fast because we only need to consider unique points-to sets since duplicate points-to obviously require the same number of words to represent. Our experience is that many points-to sets are duplicates (recall that the auxiliary analysis is a fast yet imprecise analysis).

With a distance matrix, clustering can be performed to obtain a dendrogram. We are not interested in cutting the dendrogram, rather, we are interested in the relative prioritisation of object pairs that the clustering algorithm found, that is, which objects are closer to each other compared to other objects. For example, in the previous scenario, the clustering algorithm would quickly determine that o_a and o_f must be close together, before determining the closeness between o_a, o_b, o_c, o_d, and o_e. With a counter starting from 0, we can assign objects to identifiers by visiting objects depth-first in the dendrogram. The depth-first search ensures that the clusters within a cluster are visited before visiting adjacent, less related clusters. That is, within each cluster with more than one object (including the large cluster of all objects), the identifiers for the objects in that cluster are sequential. This is very fast being linear in the size of the dendrogram which has $2n - 1$ nodes, where n is the number of objects.

4.4 More Efficient Region-Based Clustering

A distance matrix implemented as a literal matrix, rather than as a hash table for example, is more efficient. Since the distance matrix is symmetrical (the distance of o to o' is the same as the distance

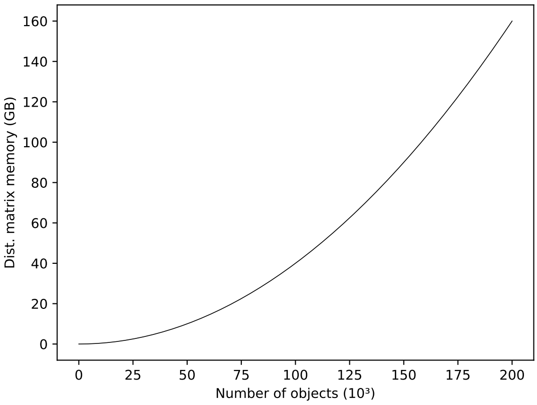


Fig. 4. Memory used to represent various (condensed) distance matrices.

from o' to o), a condensed distance matrix which is a linear array forming the upper triangle of the matrix [Müllner 2013] is sufficient. The number of elements in such a matrix would be $\frac{n(n-1)}{2}$ where n is the number of objects we are clustering. In other words, the number of elements, and hence memory usage, grows quadratically and quickly becomes problematic. In `fastcluster`, the hierarchical clustering implementation we use, this data structure is made up of `doubles` which are 64 bits or 8 bytes.

To show why this is a problem, Figure 4 shows the memory required to represent the distance matrix for various amounts of objects up to 200000. Importantly, the distance matrix is required only for clustering and can be freed immediately afterwards. So if the main phase analysis uses more memory than the distance matrix, saving memory is pointless. Otherwise, the distance matrix drives up the maximum resident set size, which may be impractical.

To remedy this, we group objects into independent *regions*, and perform clustering on a per region basis without sacrificing precision. The distance matrix is actually sparse, in that many objects do not share a meaningful relationship, rather they can be allocated distant identifiers without any effect because they never appear in a points-to set together. For example, in the points-to sets $\{o_a, o_b\}$ and $\{o_c, o_d\}$, the objects o_a and o_c do not share any meaningful relation, and no matter how distant their mapped identifiers are, bit-vector representations will not be adversely affected. To region objects, from the points-to sets of the auxiliary analysis, we create an undirected graph where there exists a path between objects o and o' if they occur together in a points-to set, and determine the connected components of the graph[2]. Objects in the same connected component share a relationship, either directly, or transitively, and the difference in the identifiers allocated can be meaningful and so they should be clustered together.

Since objects in distinct regions do not share a relationship, we can build individual distance matrices for objects within the same region and cluster those objects separately from the objects in other regions. Rather than starting our identifier counter from 0, we simply continue the counter from where it left off when assigning identifiers for the objects in the previous clustered region. With this, we can save memory with smaller distance matrices, and time by sequentially clustering regions since clustering grows non-linearly. Furthermore, we can concurrently cluster separate regions, but in our experience, clustering is fast enough for our problem that this is not worth

[2] We are careful to say "exists a path" rather than "exists an edge" as it makes the graph smaller without losing meaning for our purpose. Practically, for each points-to set, an edge can be added from the "first" object in a points-to set to every other object in that points-to set, thus trying to add a linear number of edges, not a quadratic number, per points-to set.

Table 2. Number of memory objects, regions, non-trivial regions, objects in non-trivial objects (with the proportion of total objects they make up), and the number of objects in the largest region for our benchmark programs.

Program	Memory Objects	Regions	Non-Trivial Regions	Non-Trivial Region Objects	Largest Region
dhcpcd	3699	2903	1	569 (15.38%)	569
gawk	4784	3061	1	1138 (23.79%)	1138
bash	4339	3247	2	644 (14.84%)	553
mutt	7186	4265	1	2434 (33.87%)	2434
lynx	7917	4177	3	1949 (24.61%)	1699
xpdf	19 101	14 328	1	3468 (18.16%)	3468
ruby	20 235	9073	25	9331 (46.11%)	5069
keepassxc	37 671	28 468	9	5224 (13.87%)	4026

the implementation complexity. If concurrent clustering is foregone, we can maintain a single distance matrix at a time, further saving memory. For example, if n objects are split into two equal sized regions, we would require two distance matrices of $\frac{\frac{n}{2}(\frac{n}{2}-1)}{2}$ distances, and, if not clustered concurrently, at separate times, so only one such distance matrix is required before being freed.

Another benefit of regioning is that certain regions can be assigned identifiers randomly, that is, without any clustering. In regions with fewer than $\mathcal{W}$ objects, all identifier allocations are equivalent since any number of objects $\mathcal{W} - n$ where $n < \mathcal{W}$ require a full word to represent (modulo some concerns around the crossing of word boundaries as will be discussed in Section 4.5). Clustering these objects in such small regions brings no benefit so they can be assigned any identifiers within $\mathcal{W}$ of each other immediately. We call these regions *trivial regions*.

For our benchmark programs, Table 2 shows the number of objects, the number of regions, the number of non-trivial regions (those with more than $\mathcal{W} = 64$ objects), the number of objects in non-trivial regions, and the size of the largest region. Without regioning, it would be as though we had a single region with all the objects. We see that trivial regions dominate which greatly reduces the clustering workload. In 4 benchmarks, there only exists a single non-trivial region, and in all those instances, that region contained 15% to 35% of all objects. Only one benchmark, `ruby`, had a significant number of non-trivial regions (25), and the objects in those regions made up about half of all objects. The greatest proportion of objects in non-trivial regions compared to the total number of objects was for `ruby` at almost a half followed by `mutt` at about a third, and the smallest proportion was `keepassxc` at almost 14% closely followed by `dhcpcd` at a little over 15%.

4.5 Word-Aligned Identifier Mapping

We can produce better object-to-identifier mappings by considering the regions defined above and ensuring that objects from different regions are never assigned to identifiers within the same word. Consider two regions of objects, R_1 which contains o_a, o_b, and o_c, and R_2 which contains o_d, o_e, and o_f, and that we assign identifiers as

$$R_1 : o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2, \text{and}$$
$$R_2 : o_d \mapsto 3 \quad o_e \mapsto 4 \quad o_f \mapsto 5.$$

As a core-bit vector (though this problem applies equally to sparse bit-vectors), all points to sets with objects from R_1 (non-empty subsets of $\{o_a, o_b, o_c\}$) take the form

$$\{\, 0 \; [\, \langle \times\times\times 0 \rangle \,] \,\},$$

where each × can be 1 or 0. This is perfect in that we have achieved the theoretical limit of always representing the points-to sets in R_1 with a single word (modulo metadata like offsets or links). On the other hand, under this object identifier mapping, points-to sets with objects from R_2 – non-empty subsets of $\{o_d, o_e, o_f\}$ – can take one of 3 forms:

$$\begin{aligned} \{\, 0 \; [\, \langle 0001 \rangle \,] \,\}, & \qquad \{o_d\} \\ \{\, 4 \; [\, \langle \times\times 00 \rangle \,] \,\}, \text{or} & \qquad \{o_e\}, \{o_f\}, \{o_e, o_f\} \\ \{\, 0 \; [\, \langle 0001 \rangle, \langle \times\times 00 \rangle \,] \,\}. & \qquad \{o_d, o_e\}, \{o_d, o_f\}, \{o_d, o_e, o_f\} \end{aligned}$$

That is, to represent the non-empty subsets of $\{o_d, o_e, o_f\}$, we would require either 1 or 2 words rather than the theoretical 1 word which is in fact possible. The problem is that regions are sharing words for their objects with other regions despite there never being a points-to set with objects from both regions (by definition), resulting in not all of them making ideal use of the identifiers in a word (R_2 in our case). Though R_1 and R_2 are trivial regions, this problem applies to non-trivial regions too.

We can avoid this by aligning identifiers to words when we begin allocating identifiers to each region. Aligning to words for each region would mean that the identifiers for the objects in a region begin at an identifier divisible by $\mathcal{W}$. Returning to our example above, performing word-aligned identifier assignment would result in a mapping akin to

$$\begin{aligned} R_1 : o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2, \text{and} \\ R_2 : o_d \mapsto 4 \quad o_e \mapsto 5 \quad o_f \mapsto 6. \end{aligned}$$

We have given the same mapping to the objects in R_1 since it was word-aligned, starting at 0, a multiple of $\mathcal{W} = 4$, but began the identifiers in R_2 from 4, another multiple of $\mathcal{W} = 4$. Now we can represent all points-to sets with objects from R_1 as

$$\{\, 0 \; [\, \langle \times\times\times 0 \rangle \,] \,\},$$

as before, and all points-to sets with objects from R_2 as

$$\{\, 4 \; [\, \langle \times\times\times 0 \rangle \,] \,\},$$

thus achieving our goal of representing all such points-to sets with the theoretically fewest possible number of words (1).

Word-aligning identifier assignment leaves gaps. For example, in the mapping defined in the previous paragraph, no object maps to identifier 3. This is inconsequential, since all points-to sets only contain objects from the region they are a part of, and so these gaps are not "wasted". Concretely, in our example, there is no points-to set containing one or more objects from R_1 (assigned from 0 to 2) that would contain 4 objects in total and thus benefit from that fourth object being assigned the identifier 3. Furthermore, with core bit-vectors (and sparse bit-vectors), any excess leading zero-words caused by word alignment would be inconsequential. Though the clustering is performed on the results of the auxiliary analysis, this holds true for the main phase analysis too since each points-to set produced by the main-phase analysis is a subset of a points-to set produced by the auxiliary analysis. In other words, the main phase cannot produce larger regions.

5 EVALUATION

This section describes our implementation in SVF, the programs we use to evaluate our approach, and then details the results obtained by applying our approach to staged flow-sensitive analysis (SFS) [Hardekopf and Lin 2011] using sparse and core bit-vectors.

5.1 Implementation

We have implemented our approach in open source points-to analysis framework SVF [Sui and Xue 2016b] using `fastcluster` [Müllner 2013]. We have made the artifact publicly available [Barbar and Sui 2021][3]. For the sparse bit-vector we use the well-crafted implementation in LLVM [LLVM-SBV 2021]. LLVM's sparse bit-vector is actually an unrolled doubly linked list, and we use the default setting of using 128-bit "words" (two native words on our test machine). Thus, there may be wasted words if only 64 bits of the 128 bits are used, and excess memory usage since a singly linked list is sufficient for points-to analysis. We implement the core bit-vector using an STL `vector` as the array part. As for the standard bit-vectors used, they are our core bit-vectors set with an offset of 0 to simulate a bit-vector which strips trailing zeroes but not leading zeroes.

In theory the remapping process would be to perform clustering and an object-to-identifier mapping and then change the identifiers used for objects everywhere such that the old mapping becomes non-existent to the analysis. This would be a fast linear scan over the data structures used by SVF. However, drastic architectural changes to SVF would be required, so we decided to give our points-to sets each a pointer to the map such that objects have an "internal" identifier in the points-to set and an "external" identifier. External identifiers correspond to the original naive mapping and internal identifiers correspond to the mapping produced from clustering. Objects are then stored in points-to sets as the internal identifier, and returned (e.g., during iteration) as the external identifier. In short, only the points-to sets know about the new mapping. This introduces some slight overhead that in an ideal situation would not be incurred, so we have applied this overhead when not clustering too for a fair comparison. Otherwise, algorithms remain unchanged.

We use SVF's implementation of SFS as described in Section V of the original paper [Hardekopf and Lin 2011] staged by SVF's Andersen's analysis implementation boosted by the wave propagation constraint solving technique [Pereira and Berlin 2009]. Since we simply produce a better representation of points-to sets, points-to analysis results remain the same.

5.2 Benchmarks

We have chosen 8 real-world programs as benchmarks from amongst open source projects in a range of domains like email, networking, and language implementations. Table 3 lists the benchmarks' versions, bitcode sizes, and lines of LLVM instructions (>3.1 million) along with a short description. We count the number of LLVM lines of instructions by disassembling the bitcode (which is stripped of debug information) and running the result through `wc -l`. All programs are C programs, except `xpdf` and `keepassxc` which are written in C++. These two programs are also graphical and utilise the Qt framework. Through crux-bitcode [crux-bc 2021], programs were built with `-O3` using WLLVM [WLLVM 2021], a Clang wrapper which combines individual LLVM bitcode files into a single bitcode file suitable for whole program analysis.

5.3 Experiments

Our experiments are conducted on a machine running 64-bit Ubuntu 18.04.2 LTS with an Intel Xeon Gold 6132 processor at 2.60 GHz. Being that this processor is an x86_64 processor, we have a

[3] Our approach is a part of upstream SVF. More information is available at https://github.com/SVF-tools/SVF/wiki/Object-Clustering.

Table 3. Benchmark versions, bitcode sizes in MiB, lines of LLVM instructions, and descriptions.

Benchmark	Version	Size	LOI	Description
`dhcpcd`	9.3.4	1.2 MiB	82 939	Dynamic host configuration protocol client
`gawk`	5.1.0	2.4 MiB	179 805	GNU AWK; text filtering language interpreter
`bash`	5.0.18	2.6 MiB	196 168	Bourne Again Shell; Unix shell
`mutt`	2.0.3	3.2 MiB	224 500	Text-based email client
`lynx`	2.8.9	5.1 MiB	286 991	Text-based web browser
`xpdf`	4.03	7.6 MiB	494 764	PDF viewer
`ruby`	2.7.2	13 MiB	864 114	Ruby programming language interpreter
`keepassxc`	2.6.2	15 MiB	828 669	Password manager

native word size of 64 bits. We limited our experiments to 100 GB of memory and 24 hours of time. To measure time we use C's `clock` function on the main phase (including clustering time where relevant), and to measure memory we take the maximum resident set size reported by GNU's `time` for the entire execution. In this section, we describe the reduction in words required for sparse and core bit-vectors under various linkage criteria, and the time and memory usage of SFS when using our new mappings. All analyses are run 3 times and all values in the tables in this section represent median values.

5.3.1 Linkage Criteria. For sparse and core bit-vectors respectively, Tables 4 and 5 show the number of words used to represent all points-to sets in SFS in the theoretical best case (potentially impossible), the original naive mapping, and after producing a mapping through clustering with single-, complete-, and average-linkage criteria. By best case we mean the theoretical limit of representing each points-to set in $\lceil \frac{n}{W} \rceil$ words where n is the size of the points-to set. For our programs, the general trend is that single-linkage overwhelmingly produces the best points-to set representations, occasionally average-linkage, and never complete-linkage. Rather than always choosing single linkage, we have decided to perform the clustering three times with all the linkage criteria and choose the one which produces the best result for the auxiliary analysis. In every case, the linkage criterion which produced the representation with the fewest words for the auxiliary analysis (Andersen's) also did so for the main analysis (SFS). Despite creating three mappings, total overhead remained negligible (relatively). At worst, `xpdf` took about 10 seconds to perform this entire process of clustering and evaluation, followed by `keepassxc` which took 2.5 seconds. `dhcpcd` took the least time at less than a fifth of a second. Thus, this approach of performing clustering and mapping with each linkage criterion and choosing the best result is promising and adds marginal overhead.

Clustering never achieves a theoretically ideal mapping in our benchmarks, but does get close at times and always presents a drastic difference from the original mapping. Without the ability to achieve a practically ideal mapping (described in Section 4.1), we do not know how much room for improvement there is. In other words, it is possible the pigeonhole principle is playing a significant role in preventing us from achieving a better mapping, rather than the approximate nature of hierarchical mapping. On average, we see a reduction in word count by 4.94× for sparse bit-vectors and 5.66× for core bit-vectors, with a maximum of 6.75× and 13.03×, respectively.

It should be noted that this not only improves the time and memory required to perform the main analysis (as is shown in Sections 5.3.2 and 5.3.3), but points-to analysis clients benefit too since they have more compact points-to sets to work with. For example, alias queries, implemented

Table 4. Number of words required for sparse bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	24 726 044	**4 991 412**	6 635 746	6 635 082	4.95×
gawk	58 007 460	429 843 096	**82 989 102**	132 528 508	99 502 900	5.18×
bash	26 586 881	289 532 162	**42 914 256**	42 914 700	53 173 774	6.75×
mutt	51 298 142	490 533 016	**102 662 924**	145 767 682	160 026 830	4.78×
lynx	133 664 618	965 029 738	267 599 228	319 144 056	**215 831 960**	4.47×
xpdf	731 879 787	3 943 399 840	1 463 766 422	1 527 354 270	**1 400 196 420**	2.82×
ruby	320 059 196	3 920 937 120	888 289 648	889 035 364	**764 713 778**	5.13×
keepassxc	13 770 856	315 331 268	**47 619 934**	74 384 858	67 691 754	6.62×
Geo. Mean						4.94×

Table 5. Number of words required for core bit-vectors theoretically, in the original naive mapping, and under mappings produced using various linkage criteria. The reduction compares the result predicted to be best (in bold) against the original.

Benchmark	Theoretical	Original	Single	Complete	Average	Reduction
dhcpcd	3 317 195	23 911 465	**4 961 417**	6 605 816	5 784 023	4.82 ×
gawk	58 007 460	429 739 789	**82 783 110**	140 588 641	148 836 214	5.19 ×
bash	26 586 881	295 168 808	**31 731 607**	36 861 568	47 120 912	9.30 ×
mutt	51 298 142	548 971 273	**87 213 543**	260 457 927	259 746 461	6.29 ×
lynx	133 664 618	1 015 676 964	**237 113 529**	289 849 510	302 122 259	4.28 ×
xpdf	731 879 787	4 197 513 654	1 558 434 196	1 558 496 134	**1 526 729 185**	2.75 ×
ruby	320 059 196	6 600 730 356	**1 405 659 097**	2 514 836 137	2 186 425 117	4.70 ×
keepassxc	13 770 856	1 399 786 369	**107 456 539**	134 257 502	120 881 288	13.03×
Geo. Mean						5.66 ×

as an intersection test, would become cheaper, and iteration over points-to sets would also become cheaper with the improved spatial locality.

5.3.2 Time. We compare the baseline using sparse bit-vectors with the analysis using the new mapping using sparse bit-vectors, and similarly for standard bit-vectors against the new mapping using core bit-vectors. For analysis time, we see an average speed up of 1.62× and 1.35×, respectively, over the baseline as shown in Table 6. We believe this time improvement would become more drastic as analyses grow either through larger benchmarks, more precision, or both. Generally using sparse bit-vectors with a clustered mapping is faster than using core bit-vectors unclustered, except in the case of xpdf. In Tables 4 and 5, xpdf saw the least improvement in word count reduction. Finally, for all programs except keepassxc, the CBVs with a clustered mapping are faster than their SBV counterparts. Notably, keepassxc is the program with the largest difference in word count between sparse and core bit-vectors at over 2× more for core bit-vectors. Still, these two times (and likewise for some other benchmarks) lie within a margin of error. We discuss difficulties in clustering effectively more in the next section.

5.3.3 Memory. As for memory usage, we see a more noticeable impact as in Table 7. With our new mappings, we use over 2.43× and 2.35× less memory than the baselines on average. The baseline

Table 6. Time to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in seconds. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	**Unclustered**			**Clustered**		**Improvement**	
	SBV	**BV**	**CBV**	**SBV**	**CBV**	**SBV/SBV**	**BV/CBV**
dhcpcd	66.50	59.09	58.40	52.93	52.87	1.25×	1.12×
gawk	1417.51	1002.50	971.71	773.34	655.85	1.83×	1.53×
bash	307.88	218.72	228.09	188.46	175.58	1.63×	1.25×
mutt	666.08	517.56	464.38	396.15	360.53	1.68×	1.44×
lynx	3058.57	2531.57	2383.98	1889.04	1880.56	1.62×	1.35×
xpdf	OOM	10 836.70	10 518.00	10 904.40	9636.46	-	1.12×
ruby	OOM	OOM	7451.52	6159.25	5985.72	-	-
keepassxc	1084.50	1093.42	1049.59	624.52	628.30	1.74×	1.74×
Geo. Mean						1.62×	1.35×

could not analyse ruby within the allocated 100 GB. At most we see an improvement of 4.05× (keepassxc) and at worst an improvement of 1.35× (the smallest program, dhcpcd).

With the original unclustered mapping, we see that at times moving from standard bit-vectors to core bit-vectors hardly improves memory usage, notably, xpdf, for example. Being that it is large and contains many memory objects we would expect it to improve more. The most likely reason is that the points-to sets in those benchmarks have large swathes of zero-words that are not at the start. In other words, with few leading zero-words, core bit-vectors would not have much to strip compared to the standard bit-vector which maintains leading zero-words. This is proven by the significant improvement achieved for xpdf by using a better mapping (2.41×).

The general trend is that core bit-vectors require less memory than their sparse counterparts due to the extra metadata required for sparse bit-vectors. However, this is only true when the number of words required is negligibly (relatively) similar, and this is not the case for ruby and keepassxc where we see a small regression moving from sparse bit-vectors to core bit-vectors. Notably, in terms of absolute numbers, these programs have a largest region larger than each of the other programs. Larger regions strongly suggest more complex relations due to more objects being related, whether directly (Definition 4.1) or transitively. In a larger region, a single poor choice during clustering in mapping of an object to an identifier is more costly in terms of adding redundant zero-words than in smaller regions, especially for contiguous bit-vectors. This regression is extremely slight despite the large difference in required words for core bit-vectors but there is potential for the rift to grow as benchmarks become larger. In these cases, sparse bit-vectors remain viable, and in either case, our benchmarks have always shown improvement over using unclustered mappings.

It is possible that some programs are simply too difficult to cluster effectively for contiguous bit-vectors like the core bit-vector. Especially, when more objects are directly related ($d(o_1, o_2) \neq \infty$) or transitively related, the pigeonhole principle becomes more prominent. This large number of relations can occur simply because of complex points-to relations or specifically because the auxiliary analysis poorly handles the input program and introduces relations which do not exist in the more precise main analysis. An auxiliary analysis compared to the main analysis may, for example, poorly handle key indirect/virtual calls. Since the integer programming approach does

Table 7. Memory required to run SFS using unclustered identifiers with sparse bit-vectors (SBV), bit-vectors (BV), and core bit-vectors (CBV) and using clustered identifiers with SBVs and CBVs, in gigabytes. Using unclustered identifiers with SBVs and BVs forms the baseline. The best result achieved by the baseline and by our approach are compared. OOM and OOT mean the analysis exhausted available memory and time, respectively.

Benchmark	**Unclustered**			**Clustered**		**Improvement**	
	SBV	**BV**	**CBV**	**SBV**	**CBV**	**SBV/SBV**	**BV/CBV**
dhcpcd	1.20	0.92	0.89	0.74	0.68	1.62×	1.35×
gawk	12.76	8.02	7.76	4.63	3.67	2.75×	2.19×
bash	9.00	4.94	5.06	3.23	2.67	2.79×	1.86×
mutt	14.56	11.68	11.42	5.47	4.56	2.66×	2.56×
lynx	29.07	21.83	19.23	11.51	9.38	2.53×	2.33×
xpdf	OOM	72.08	68.62	42.47	29.91	≥2.35×	2.41×
ruby	OOM	OOM	86.91	32.95	34.67	≥3.04×	≥2.88×
keepassxc	12.41	25.19	24.31	6.15	6.22	2.02×	4.05×
Geo. Mean						≥2.43×	≥2.35×

not practically scale to produce an actual ideal mapping (for the auxiliary analysis's points-to sets, let alone those for the main analysis), we do not know how close the mapping produced by clustering for core bit-vectors actually is to the best possible case (rather than what is theoretically, but unlikely, possible), or how close the sparse bit-vector result is (despite its excess metadata). In either case, our mapping provides significant memory reduction as it stands. Overall, on average (geometric mean), as core bit-vectors and as sparse bit-vectors, our mappings use around 85% and 90% more words, respectively, than the potentially impossible theoretical best listed in Tables 4 and 5. However, as benchmarks grow, so does this difference.

6 RELATED WORK

Algorithmic-level improvement for points-to analysis. Whole program points-to analyses have been studied extensively in the past few decades. Inclusion-based set-constraint solving [Andersen 1994] has been widely used for Andersen's flow-insensitive analysis and among many other precision dimensions including field-sensitivity, flow-sensitivity and context-sensitivity [Barbar et al. 2020; Hardekopf and Lin 2011; Lhoták and Chung 2011; Pearce et al. 2003, 2007; Smaragdakis et al. 2011]. Resolving points-to sets in inclusion-based analysis is formulated as a set-constraint or set-union problem by computing a dynamic transitive closure on top of the constraint graph of a program. As the analysis becomes more precise, the number of constraints increases and the cost of each union does too. The majority of work on whole-program analysis focuses on improving the analysis at the algorithmic level to reduce overhead of inclusion-based analysis [Fähndrich et al. 1998; Hardekopf and Lin 2007a; Heintze and Tardieu 2001; Pearce et al. 2003; Pereira and Berlin 2009]. For example, Fähndrich et al. [1998] introduced a partial online cycle elimination while processing complex constraints (e.g., Load/Store) and demonstrated that cycle detection is important for scaling inclusion-based points-to analysis and Heintze and Tardieu [2001] proposed a new field-based Andersen's analysis that can analyse large-scale programs. Later, a series of techniques, such as lazy cycle detection [Hardekopf and Lin 2007a], offline variable substitution [Rountev and Chandra 2000], and optimisations on constraint propagation [Pereira and Berlin 2009] have also been proposed to accelerate inclusion-based analysis at the algorithmic level.

Points-To Set Data Structure. The points-to set is key to realising points-to analysis. Data structures for implementing points-to sets include bit-vectors, BDDs, and explicit representations like B-trees and hash-based sets. BDD-based points-to analysis often requires variable reordering to be efficient [Bravenboer and Smaragdakis 2009] and/or algorithmic changes to the analysis [Berndl et al. 2003; Zhu and Calman 2004]. Creating efficient BDDs can be extremely time consuming and the benefits may not outweigh using explicit representations [Bravenboer and Smaragdakis 2009].

Bit-vectors have been commonly used for solving constraints in mainstream frameworks such as Soot [Lhoták and Hendren 2003], WALA[WALA 2021], and LLVM-based static analysis tools [Hardekopf and Lin 2011; Schubert et al. 2019; Sui and Xue 2016b]. Bit-vectors need an object-to-identifier mapping without requiring algorithmic changes for points-to analysis, and despite mappings previously being basic, have been shown more efficient than hash-based sets and sorted arrays [Lhoták and Hendren 2003], and BDDs Hardekopf and Lin [2009]. As a more sophisticated mapping for Java-based analyses, Toussi and Khademzadeh [2013] have suggested assigning objects of the same type (or with a subtyping relation) to consecutive identifiers. They also discuss a bit-vector (the ranged bit-vector) similar to the core bit-vector but uses a static over-approximated offset and length such that a pointer of type T only needs to consider type T objects (or subtype of T objects) in its points-to set. While type filtering for strongly-typed languages like Java can improve *both* performance and precision [Lhoták and Hendren 2003], this is not the case for weakly-typed languages like C and C++. In points-to analysis for C and C++, objects do not have a set type, typing objects may require heap cloning, and pointers may point to objects of any type [Balatsouras and Smaragdakis 2016; Barbar et al. 2020]. This incurs (sometimes very significant) overhead and type filtering may involve assumptions that are not suitable for all practitioners [Barbar et al. 2020]. In this paper, we make one step forward by exploring object clustering to generally create a better mapping in staged analyses and boost the performance of bit-vectors to save both time and space.

Staged Analysis. Recently, staged analysis [Fink et al. 2008; Hardekopf and Lin 2011; Oh et al. 2012; Tavares et al. 2014], which uses pre-computed points-to information to bootstrap a later more precise analysis, has been leveraged to scale precise points-to. The insight is that the points-to results of the auxiliary analysis will always be more conservative (e.g., $pt(p)$ in the auxiliary analysis is a superset of $pt(p)$ in the main analysis) [Hardekopf and Lin 2011]. These results can then be used to determine properties of the main analysis such as building a sound def-use graph to perform a sparse analysis for both top-level and address-taken variables [Chow et al. 1996; Hardekopf and Lin 2011; Sui et al. 2012]. Our work, inspired by staged analysis, also aims to leverage the already computed auxiliary points-to information to perform efficient and effective object clustering, yielding more compact points-to sets for precise points-to analyses like flow-sensitive points-to analysis.

7 CONCLUSION

This paper presents a new object clustering approach to compacting points-to sets for whole-program points-to analysis. We first formulate compacting points-to sets as an optimisation problem solved by integer programming, and then develop a more approximate yet much more efficient hierarchical clustering approach to compact bit-vector-based points-to sets. These techniques require no high-level algorithmic changes to the points-to analyses. We also develop an improved bit-vector to better take advantage of our clustering approach. We have evaluated our approach using a staged flow-sensitive analysis on 8 benchmarks. Our approach can successfully reduce the number of words required for sparse and contiguous bit-vectors by up to 6.75× (4.94× on average) and up to 13.03× (5.66× on average), respectively. Analysis time improves by up to 1.83× (1.62× on average) and up to 1.74× (1.35× on average), and memory usage is reduced by up to 3.04× (2.43× on average) and up to 4.05× (2.35× on average) for sparse and contiguous bit-vectors respectively.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their reviews and suggestions and we thank Joakim Skarding for his useful insights on optimisation and integer programming. For most of the duration of this project, the first author was supported by a PhD scholarship funded by CSIRO's Data61. This research is supported by Australian Research Grants DP200101328 and DP210101348.

REFERENCES

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation. University of Copenhagen, Denmark.

George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *International Static Analysis Symposium (SAS '16).* Springer, Germany, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

Mohamad Barbar and Yulei Sui. 2021. *Compacting Points-To Sets through Object Clustering (Artifact).* Zenodo. https://doi.org/10.5281/zenodo.5507442

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-Sensitive Type-Based Heap Cloning. In *34th European Conference on Object-Oriented Programming (ECOOP '18, Vol. 166).* Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 24:1–24:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.24

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21).* IEEE Computer Society, USA, 222–235. https://doi.org/10.1109/CGO51591.2021.9370334

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to Analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03).* ACM, USA, 103–114. https://doi.org/10.1145/781131.781144

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* ACM, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. 1996. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96).* Springer, Germany, 253–267. https://doi.org/10.1007/3-540-61053-7_66

crux-bc 2021. crux-bitcode. https://github.com/mbarbar/crux-bitcode. Last accessed on 14 September 2021.

Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98).* ACM, USA, 85–96. https://doi.org/10.1145/277650.277667

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering and Methodology* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

Ben Hardekopf and Calvin Lin. 2007a. 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 (PLDI '07).* ACM, USA, 290–299. https://doi.org/10.1145/1250734.1250767

Ben Hardekopf and Calvin Lin. 2007b. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *International Static Analysis Symposium (SAS '07).* Springer, Germany, 265–280. https://doi.org/10.1007/978-3-540-74061-2_17

Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09).* ACM, USA, 226–238. https://doi.org/10.1145/1480881.1480911

Ben Hardekopf and Calvin Lin. 2011. 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 (CGO '11).* IEEE Computer Society, USA, 289–298. https://doi.org/10.1109/CGO.2011.5764696

Nevin Heintze and Olivier Tardieu. 2001. 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 (PLDI '01).* ACM, USA, 254––263. https://doi.org/10.1145/378795.378855

ISO/IEC. 2017. *ISO/IEC 14882:2017 — Programming languages — C++* (fifth ed.). International Organization for Standardization, Switzerland. 1605 pages.

Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New Phytologist* 11, 2 (1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276510

Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 179 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428247

Jakub Kuderski, Jorge A Navas, and Arie Gurfinkel. 2019. Unification-based Pointer Analysis without Oversharing. In *2019 Formal Methods in Computer Aided Design (FMCAD '19)*. IEEE Computer Society, USA, 37–45. https://doi.org/10.23919/FMCAD.2019.8894275

Chris Lattner and Vikram Adve. 2004. 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 (CGO '04)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

Yuxiang Lei and Yulei Sui. 2019. Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis. In *International Static Analysis Symposium (SAS '19)*. Springer, Germany, 27–47. https://doi.org/10.1007/978-3-030-32304-2_3

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, USA, 3–16. https://doi.org/10.1145/1926385.1926389

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC '03)*. Springer, Germany, 153–169.

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Transactions on Programming Languages and Systems* 41, 1, Article 6 (March 2019), 31 pages. https://doi.org/10.1145/3293606

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

LLVM-BV 2021. https://llvm.org/doxygen/BitVector_8h_source.html. Last accessed on 16 April 2021.

LLVM-SBV 2021. https://llvm.org/doxygen/SparseBitVector_8h_source.html. Last accessed on 16 April 2021.

Oded Maimon and Lior Rokach. 2005. *Data Mining and Knowledge Discovery Handbook* (1st ed.). Springer, USA. https://doi.org/10.1007/b107408

Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-Based Points-to Analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, USA, 428–443. https://doi.org/10.1145/1869459.1869495

Fionn Murtagh. 1983. A Survey of Recent Advances in Hierarchical Clustering Algorithms. *Comput. J.* 26, 4 (11 1983), 354–359. https://doi.org/10.1093/comjnl/26.4.354

Daniel Müllner. 2013. fastcluster: Fast Hierarchical, Agglomerative Clustering Routines for R and Python. *Journal of Statistical Software, Articles* 53, 9 (2013), 1–18. https://doi.org/10.18637/jss.v053.i09

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, USA, 229––238. https://doi.org/10.1145/2254064.2254092

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, USA, 3–12. https://doi.org/10.1109/SCAM.2003.1238026

David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (Nov. 2007), 4:1–4:42. https://doi.org/10.1145/1290520.1290524

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. Wave Propagation and Deep Propagation for Pointer Analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, USA, 126–135. https://doi.org/10.1109/CGO.2009.9

Atanas Rountev and Satish Chandra. 2000. Off-Line Variable Substitution for Scaling Points-to Analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, USA, 47—-56. https://doi.org/10.1145/349299.349310

Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: an inter-procedural static analysis framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '19)*. Springer, Germany, 393–410. https://doi.org/10.1007/978-3-030-17465-1_22

Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-Based Pre-Processing for Points-to Analysis. In *Proceedings of the 28th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '13)*. ACM, USA, 253—-270. https://doi.org/10.1145/2509136.2509524

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, USA, 17–30. https://doi.org/10.1145/1926385.1926390

Yulei Sui and Jingling Xue. 2016a. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, USA, 460–473. https://doi.org/10.1145/2950290.2950296

Yulei Sui and Jingling Xue. 2016b. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, USA, 265–266. https://doi.org/10.1145/2892208.2892235

Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, USA, 254–264. https://doi.org/10.1145/2338965.2336784

André Tavares, Benoit Boissinot, Fernando Pereira, and Fabrice Rastello. 2014. Parameterized Construction of Program Representations for Sparse Dataflow Analyses. In *Proceedings of the 23rd International Conference on Compiler Construction*. Springer, Germany, 18–39. https://doi.org/10.1007/978-3-642-54807-9_2

Hamid A Toussi and Ahmed Khademzadeh. 2013. Improving Bit-Vector Representation of Points-To Sets Using Class Hierarchy. *International Journal of Computer Theory and Engineering* 5, 3 (2013), 494–499. https://doi.org/10.7763/IJCTE.2013.V5.736

WALA 2021. The T. J. Watson Libraries for Analysis (WALA). http://wala.sf.net/. Last accessed on 14 September 2021.

John Whaley. 2007. *Context-Sensitive Pointer Analysis Using Binary Decision Diagrams*. Ph.D. Dissertation. Stanford University, USA.

WLLVM 2021. Whole Program LLVM in Go. https://github.com/SRI-CSL/gllvm. Last accessed on 14 September 2021.

Jianwen Zhu and Silvian Calman. 2004. Symbolic Pointer Analysis Revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, USA, 145–157. https://doi.org/10.1145/996841.996860