Scalable Context-Sensitive Points-to Analysis
Using Multi-dimensional Bloom Filters

Rupesh Nasre¹, Kaushik Rajan², R. Govindarajan¹, and Uday P. Khedker³

¹ Indian Institute of Science, Bangalore, India
² Microsoft Research, Bangalore, India
³ Indian Institute of Technology, Bombay, India

Abstract. Context-sensitive points-to analysis is critical for several program optimizations. However, as the number of contexts grows exponentially, storage requirements for the analysis increase tremendously for large programs, making the analysis non-scalable. We propose a scalable flow-insensitive context-sensitive inclusion-based points-to analysis that uses a specially designed multi-dimensional bloom filter to store the points-to information. Two key observations motivate our proposal: (i) points-to information (between pointer-object and between pointer-pointer) is sparse, and (ii) moving from an exact to an approximate representation of points-to information only leads to reduced precision without affecting correctness of the \((may-points-to)\) analysis. By using an approximate representation a multi-dimensional bloom filter can significantly reduce the memory requirements with a probabilistic bound on loss in precision. Experimental evaluation on SPEC 2000 benchmarks and two large open source programs reveals that with an average storage requirement of 4MB, our approach achieves almost the same precision (98.6%) as the exact implementation. By increasing the average memory to 27MB, it achieves precision up to 99.7% for these benchmarks. Using Mod/Ref analysis as the client, we find that the client analysis is not affected that often even when there is some loss of precision in the points-to representation. We find that the \(\text{NoModRef}\) percentage is within 2% of the exact analysis while requiring 4MB (maximum 15MB) memory and less than 4 minutes on average for the points-to analysis. Another major advantage of our technique is that it allows to trade off precision for memory usage of the analysis.

1 Introduction

Pointer analysis enables many compiler optimization opportunities and remains as one of the most important compiler analyses. For client analyses, both precision and speed of the underlying pointer analysis play a vital role. Several context-insensitive algorithms have been shown to scale well for large programs [1][2][3][4]. However, these algorithms are significantly less precise for real world programs compared to their context-sensitive counterparts [5][6][7][8]. Unfortunately,
context-sensitive pointer analysis improves precision at the cost of high — often unacceptable — storage requirement and analysis time. These large overheads are an artifact of the large number of contexts that a program might have. For example, the SPEC2000 benchmark con has 19K pointers if we do not consider context information but the number increases to 417K pointers if we consider all context-wise pointers. Scaling a context sensitive points-to analysis is therefore a challenging task. Recent research (see Related Work in Section 5) has focused on the scalability aspect of context-sensitive points-to analysis and achieves moderate success in that direction[9][4]. However, the memory requirements are still considerably large. For instance, in [9], most of the larger benchmarks require over 100 MB for points-to analysis. Hence, scalability still remains an issue. Also, none of the current analyses provide a handle to the user to control the memory usage of a points-to analysis. Such a feature will be useful when analyzing a program in a memory constrained environment.

The objective of a context-sensitive points-to analysis is to construct, for each pointer and context, a set containing all the memory locations (pointees) that the pointer can point to in that context. This paper proposes a new way of representing points-to information using a special kind of bloom filter[10] that we call a multi-dimensional bloom filter.

A bloom filter is a compact, and approximate, representation (typically in the form of bit vectors) of a set of elements which trades off some precision for significant savings in memory. It is a lossy representation that can incur false positives, i.e., an element not in the set may be answered to be in the set. However, it does not have false negatives, i.e., no element in the set would be answered as not in the set. To maintain this property, the operations on a bloom filter are restricted so that items can only be added to the set but can never be deleted. Our motivation for using bloom filters for context-sensitive flow-insensitive points to analysis stems from the following three key observations.

- **Conservative static analysis:** As with any other compiler analysis, static points-to analysis tends to be conservative as correctness is an absolute requirement. Thus, in case of static may-points-to analysis, a pointer not pointing to a variable at run time can be considered otherwise, but not vice-versa. As a bloom filter does not have false negatives, a representation that uses bloom filters is safe. A bloom filter can only (falsely) answer that a pointer points to a few extra pointees. This only makes the analysis less precise and does not pose any threat to correctness. Further, as a bloom filter is designed to efficiently trade off precision for space it is an attractive representation to enable scalability of points-to analysis.

- **Sparse points-to information:** The number of pointees that each context-wise pointer (pointer under a given context) actually points to is many orders of magnitude less than both the number of context-wise pointers and the total number of potential pointees. Hence, though the points-to set can potentially be very large, in practice, it is typically small and sparse. A bloom filter is

\[1\] Some modified bloom filter structures[11] have been proposed that can support deletion but they do so at the expense of introducing false negatives.