Scalable Context-Sensitive Points-To Analysis using Multi-Dimensional Bloom Filters.

Rupesh Nasre[1], Kaushik Rajan[2], R. Govindarajan[1], Uday P. Khedker[3].

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

nasre@csa.iisc.ernet.in, kaushik@msr.microsoft.com,
govind@serc.iisc.ernet.in, uday@cse.iitb.ac.in

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 upto 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 *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 *eon* 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[1]. 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

[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.

potentially be very large, in practice, it is typically small and sparse. A bloom filter is ideally suited to represent data of this kind. When the set is sparse, a bloom filter can significantly reduce the memory requirement with a probabilistically low bound on loss in precision.
- ***Monotonic data flow analysis:*** As long as the underlying analysis uses a monotonic iterative data flow analysis, the size of the points-to set can only increase monotonically. This makes a bloom filter a suitable choice as monotonicity guarantees that there is no need to support deletions.

The above observations make a bloom filter a promising candidate for representing points-to information. However, using the bloom filter as originally proposed in [10] is not efficient for a context sensitive analysis. We therefore extend the basic bloom filter to a multi-dimensional bloom filter (*multibloom*) to enable efficient storage and manipulation of context aware points-to information. The added dimensions correspond to *pointers*, *calling contexts*, and *hash functions*. The bloom filter is extended along the first two dimensions (pointers and calling contexts) to support all the common pointer manipulation operations ($p = q$, $p = \&q$, $p = *q$ and $*p = q$) and the query operation $DoAlias(p, q)$ efficiently. The third dimension (hash functions) is essential to control loss in precision. We theoretically show and empirically observe that larger the number of hash functions, lower is the loss in precision. In effect, multibloom significantly reduces the memory requirement with a very low probabilistically bound loss in precision. The compact representation of points-to information allows the context sensitive analysis to scale well with the program size.

The major contributions of this paper are:

- We propose a multi-dimensional bloom filter (*multibloom*) that can compactly represent the points-to information with almost no loss in precision.
- Using extended bloom filter operations, we develop a context-sensitive flow-insensitive points-to analysis for C programs in the LLVM compilation infrastructure.
- We show that by using multibloom, a user can control the total memory requirement of a compiler analysis, unlike in most other analyses.
- We demonstrate the effectiveness of multibloom through experimental evaluation on 16 SPEC 2000 benchmarks and 2 real world applications. With less than 4MB memory on average (maximum 15MB), multibloom achieves more than 98% precision, taking less than 4 minutes per benchmark on average.
- We also evaluate precision of a client Mod/Ref analysis. We find that using *multibloom*, the *NoModRef* percentage is within 1.3% of the exact analysis while requiring 4MB memory and 4 minutes on average for the points-to analysis.

2 Background

General purpose languages like C pose many challenges to the compiler community. Use of pointers hinders many compiler optimizations. Pointers with multiple

```
void main() {
  S1: p1 = f(&x);
      p3 = p1;
  S2: p2 = f(&y);
}

int *f(int *a) {
      int *u, *v;
  S3: u = g(&x);
  S4: v = g(&y);
      return a;
}

int *g(int *b) {
      return b;
}
```

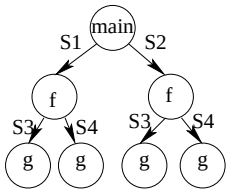

Fig. 1. Example program and its invocation graph.

indirections, pointers to functions, etc. only add to these challenges. For analyzing such complicated programs, however, it is sufficient to assume that all pointer statements in the program are represented using one of the four basic forms: address-of assignment ($p = \&q$), copy assignment ($p = q$), load assignment ($p = *q$) and store assignment ($*p = q$)[12] (we describe how these statements are handled by our analysis in Section 3). Our analysis handles all aspects of C (including recursion), except variable number of arguments.

2.1 Context-Sensitive Points-to Analysis

A context-sensitive points-to analysis distinguishes between various calling contexts of a program and thus, is able to more accurately determine the points-to information compared to the context-insensitive version [5]. This precision, however, comes at a price: storing the number of contexts, which is huge in a large C program. Consider the example program and its invocation graph shown in Figure 1. The invocation graph shows that for different contexts, function f has 2 instances and function g has 4 instances. The number of distinct paths from *main* to the leaf nodes in the graph is equal to the number of different contexts the program has. In general, the number of contexts in a program can be exponential in terms of the number of functions. For instance, the number of methods in the open source program *pmd* is 1971, but it has 10^{23} context-sensitive paths[9]. Therefore, for a context-sensitive points-to analysis, the number of points-to tuples can be exponential (in the number of functions in the program). The exponential blow up in the number of contexts, typically results in an exponential blow up in the storage requirement for exact representation of context-wise points-to tuples.

Reducing the storage requirements of a context-sensitive points-to analysis has attracted much research in pointer analysis. Several novel approaches have been proposed for scalable pointer analyses (see Section 5 for related work). Despite these advances, absolute values of memory and time required are substantially high. For instance, in [9], all the benchmarks having more than 10K methods (*columba*, *gantt*, *jxplorer*, *jedit*, *gruntspud*) require over 100MB of memory. For the benchmarks we evaluate, we find that the number of pointers increases by 1 or 2 orders of magnitude if we track them in a context-wise manner. So it is possible that the memory and time requirements of a context-sensitive

analysis will be a few orders of magnitude higher than a context insensitive analysis.

Our goal, in this paper, is to reduce this storage and execution time requirement of a context-sensitive points-to analysis. This is achieved by using a variant of bloom filter, which sacrifices a small amount of precision. As we shall see in the next subsection, once the user fixes the size of a bloom filter, he/she can estimate a probabilistic bound on the loss in precision as a function of the average number of pointees of a pointer (in a given context).

2.2 Bloom Filter

A bloom filter is a probabilistic data structure used to store a set of elements and test the membership of a given element[10]. In its simplest form, a bloom filter is an array of N bits. An element e belonging to the set is represented by setting the kth bit to 1, where $h(e) = k$ and h is the hash function mapping element e to k^{th} bit. For instance, if the hash function is $h_1(e) = (3 * e + 5)\%N$, and if $N = 10$, then for elements $e = 13$ and 100, the bits 4 and 5 are set. Membership of an element e is tested by using the same hash function. Note that element 3 also hashes to the same location as 13. This introduces false positives, as the membership query would return *true* for element 3 even if it is not inserted. Note, however, that there is no possibility of false negatives, since we never reset any bit.

The false positive rate can be reduced drastically by using multiple hash functions. Thus, if we use two hash functions for the above example, with $h_2(e) = (\lfloor e/2 \rfloor + 9)\%N$, then the elements $e = 13$, 100 get hashed to bits 5, 9. Note that a membership query to 3 would return *false* as location 0 (corresponding to $h_2(3)$) is 0, even though location 4 (corresponding to $h_1(3)$) is set. Thus, using multiple hash functions the false positives can be reduced.

The false positive rate P for a bloom filter of size N bits after n elements are added to the filter with d hash functions is given by Equation 1 (from [10]).

$$P = \frac{(1/2)^d}{(1 - \frac{nd}{N})} \tag{1}$$

This is under the assumption that the individual hash functions are *random* and different hash functions are *independent*. Unlike traditional data structures used in points-to analysis[5][8], time to insert elements in a bloom filter and to check for their membership is independent of the number of elements in the filter.

3 Points-to Analysis using Bloom Filters

A points-to tuple $\langle p, c, x \rangle$ represents a pointer p pointing to variable x in calling context c. A context is defined by a sequence of functions and their call-sites. A naive implementation stores context-sensitive points-to tuples in a bloom filter by hashing the tuple $\langle p, c, x \rangle$ and setting that bit in the bloom filter. This simple

operation takes care of statements only of the form $p = \&x$. Other pointer statements, like $p = q$, $p = *q$, and $*p = q$ require additional care. For example, for handling $p = q$ type of statements, the points-to set of q has to be copied to p. While bloom filter is very effective for existential queries, it is inefficient for universal queries like "*what is the points-to set of pointer p under context c?*".

One way to solve this problem is to keep track of the set of all pointees (objects). This way, the query $FindPointsTo(p, c)$ to find the points-to set for a pointer p under context c is answered by checking the bits that are set for each of the pointees. Although this is possible in theory, it requires storing all possible pointees, making it storage inefficient. Further, going through all of them every time to process a $p = q$ operation makes this strategy time inefficient. Further complications arise if we want to support a context-sensitive *DoAlias* query. Therefore, we propose an alternative design that has more dimensions than a conventional bloom filter in order to support the pointer operations.

3.1 Multi-Dimensional Bloom Filter

Our proposed multi-dimensional bloom filter (multibloom) is a generalization of the basic bloom filter introduced in Section 2.2. It has 4 dimensions, one each for pointers, contexts, hash functions and a bit vector along the fourth dimension. It is represented as $mb[P][C][D][B]$. The configuration of a multibloom is specified by a 7-tuple $\langle P, C, D, B, M_p, M_c, H\rangle$ where P is the number of entries for pointers, C is the number of entries for contexts, D is the number of hash functions, B is the bit-vector size for each hash function, M_p is the function mapping pointers, M_c is the function mapping contexts and H is the family of hash functions. The first 4 entries (P, C, D, B) denote the number of unique values that can be taken along each dimension. For example $C = 16$ would mean that the multibloom has space for storing the pointee set for 16 contexts in which a pointer is accessed. We will have to map every context of a given pointer to one among 16 entries. The total size of the structure is $Size = P \times\ C \times\ D \times\ B$. Functions M_p and M_c map the pointer p and context c to integers $Pidx$ and $Cidx$ in the range $[0, P-1]$ and $[0, C-1]$ respectively. A family of hash functions $H{=}(h_1, h_2, \cdots, h_D)$ map the pointee x to D integers $Hidx_1, Hidx_2, \cdots, Hidx_D$ respectively. These play the same role as the hash functions in Section 2.2.

Given a points-to tuple $\langle p, c, x\rangle$, it is entered into the multibloom as follows. $Pidx$, $Cidx$ and $(Hidx_1, Hidx_2, \cdots, Hidx_D)$ are obtained using M_p, M_c and H respectively. The tuple is added to multibloom by setting the following D bits:

$$mb\,[Pidx]\,[Cidx]\,[i]\,[Hidx_i] = 1,\ \forall i \in [1, D]$$

Extending Bloom Filter Operations for $p = q$ Statement: While processing $p = q$ type of statement under context c, all we need to do is to find the $B \times D$ source bits from the multibloom that correspond to pointer q under the context c and bitwise-OR it with the $B \times D$ destination bits corresponding to pointer p under context c. This logically copies the pointees of q on to p without having to universally quantify all the pointees that q points to. The pseudo-code is given in Algorithm 1.

Algorithm 1 Handling statement $p = q$ under context c in multibloom, with D hash functions and a B bit vector

$Pidx_{src} = M_p[q]$, $Cidx_{src} = M_c[c]$
$Pidx_{dst} = M_p[p]$, $Cidx_{dst} = M_c[c]$
for $i = 1$ to D **do**
  for $j = 1$ to B **do**
    $mb[Pidx_{dst}][Cidx_{dst}][i][j] = mb[Pidx_{dst}][Cidx_{dst}][i][j]$
      $\vee\ mb[Pidx_{src}][Cidx_{src}][i][j]$
  end for
end for

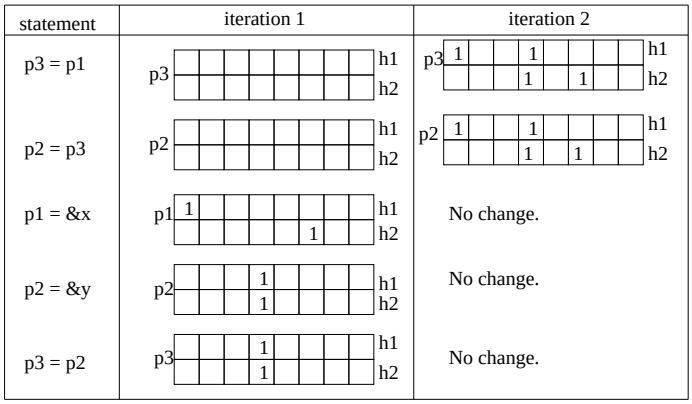

Fig. 2. Example program to illustrate points-to analysis using bloom filters. First column shows the program statements. Later columns show the state of bloom filters for different pointers after successive iterations over constraints until a fix-point is reached.

Example: Consider the program fragment given in the first column of Figure 2. Consider a multibloom with configuration

$$\langle P, C, D, B, M_p, M_c, H \rangle = \langle 3, 1, 2, 8, I, C_0, (h_1, h_2) \rangle$$

The map M_p is an identity function I that returns a different value for $p1$, $p2$ and $p3$. The two hash functions h_1 and h_2 are defined as $h_1(x) = 0,\ h_2(x) = 5,\ h_1(y) = 3\ and\ h_2(y) = 3$. C_0 maps every context to entry 0, since $C = 1$. As there is only one entry for context and each statement modifies one pointer, we illustrate the multibloom as 3 bloom filters. For clarity, we depict the multibloom as multiple 2-dimensional arrays in Figure 2. Initially, all the bits in the buckets of each pointer are set to 0. The state of bloom filters after every iteration (the analysis is flow-insensitive) for the example code is shown in Figure 2.

Extending Bloom Filter Operations for $*p = q$ and $p = *q$: There are two ways to handle statements of the form $*p = q$ and $p = *q$. One way is to extend the above strategy by adding more dimensions to the multibloom. This is extensible to multiple levels of indirection. This strategy would add more dimensions to our 4-dimensional bloom filter, one for each pointer dereference.

Clearly, this adds to storage and analysis time requirements. The second way is to conservatively assume that a pointer to a pointer points to the universal set of pointees and process the statement conservatively. The number of pointers to pointers is much less in programs compared to single-level pointers. Therefore, depending on the application, one may be willing to lose some precision by this conservative estimate. To obtain a good balance of storage requirement, analysis time and precision, we employ a combination of the above two techniques. We extend multibloom for two-level pointers (**p) and use the conservative strategy (universal set of pointees) for higher-level pointers (***p, ****p and so on). The conservative strategy results in little precision loss considering that less than 1% of all dynamic pointer statements contain more than two levels of pointer indirections (obtained empirically).

Extending multibloom for two-level pointers makes it look like $mb[P][S][C][D][B]$ where S is the number of entries for pointers that are pointees of a two-level pointer. For single-level pointers, S is 1. For two-level pointers S is configurable. For higher-level pointers S is 1 and an additional bit is set to indicate that the pointer points to the universal set of pointees.

To handle load statement $p = *q$ where p is a single-level pointer and q is a two-level pointer, all the cubes $mb[Q][i]$ (i.e., $C \times D \times B$ bits) corresponding to pointer q, $\forall i = 1..S$ are bitwise-ORed to get a resultant cube. Note that $S = 1$ for the result, i.e., the result is for a single-level pointer. This cube is then bitwise-ORed with that of p, i.e., with $mb[P][1]$. This makes p point to the pointees pointed to by all pointers pointed to by q.

To handle store statement $*q = p$ where p is a single-level pointer and q is a two-level pointer, the cube $mb[P][1]$ of p is bitwise-ORed with each cube $mb[Q][i]$ of q, $\forall i = 1..S$. It makes each pointer pointed to by q point to the pointees pointed to by p.

Handling context-sensitive load/store statements requires a modification to address-of assignment $p = \&q$. If p is a two-level pointer, then to process the address-of statement in context c, $D \times B$ bits of q are bitwise-ORed with $D \times B$ bits of p in the appropriate hash entry for q (see example below).

For mapping a pointer onto the range $1..S$, we need a mapping function M_s. The multibloom configuration is thus extended to include S and M_s.

Example: Consider the program fragment given in the first column of Figure 3. Consider a multibloom with configuration

$$\langle P, S, C, D, B, M_p, M_s, M_c, H \rangle = \langle 5, 2, 1, 1, 8, I, h_s, -, (h) \rangle$$

The map M_p is an identity function I that returns a different value for $p1$ through $p5$ The hash function h is defined as $h(x) = 1$ *and* $h(y) = 4$. The mapping function h_s is defined as $h_s(p1) = 1$ *and* $h_s(p2) = 2$. Initially, all bits in the buckets for each pointer are set to 0. The state of bloom filters after each statement is processed is shown in the second column of Figure 3. Third column describes the multibloom operation. Note that the above strategy of using an additional dimension for two-level pointers can be extended to include more dimensions to accommodate higher-level pointers.

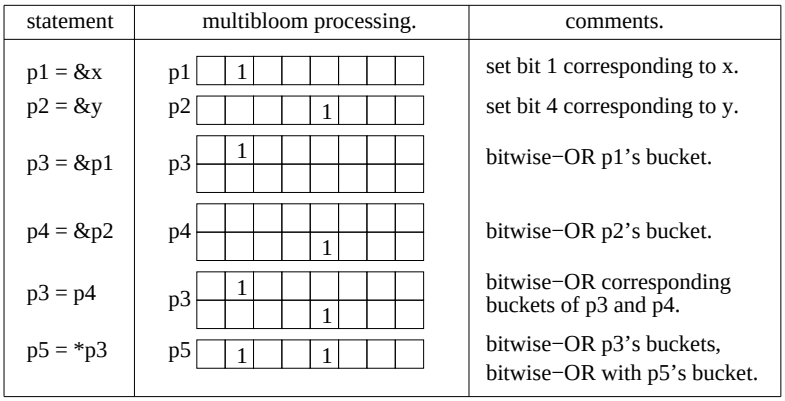

Fig. 3. Example program to illustrate handling load/store statements. First column shows the program statements. Second column shows the bloom filter state after each statement is processed. Third column describes the multibloom operation.

Storage Requirement of Multibloom: A quick analysis explains why multibloom is space efficient. Consider the SPEC 2000 benchmark *parser* which has about 10K pointers and an average of 3 pointees per context-wise pointer, on an average about 16 contexts per pointer, and around 20% two- or higher-level runtime pointer-statements. Consider a multibloom with $P = 10\text{K}$, $S = 5$, $C = 8$, $D = 8$ and $B = 50$. The total memory requirement for the multibloom is $10\text{K} \times (0.2 \times 5 + 0.8 \times 1) \times 8 \times 8 \times 50$ bits $= 4.32\text{MB}$. This is much less than what a typical analysis would require, which is at least a few tens of megabytes for a program having 10K pointers.

To measure the false positive rate we will now try to map the values back from a 4-dimensional multibloom to a 2-dimensional bloom filter so that we can apply Equation 1. As there are 16 contexts on an average per pointer and $C = 8$, on average 2 contexts would map to a given context bin. Therefore the number of entries per bloom filter would be twice the average number of pointees per context-wise pointer. Now assuming the representation across pointers is more or less uniform, we can use the equation with $N = B \times D = 400$, $d = D = 8$, $n = 3 \times 2 = 6$ (average number of contexts per bin multiplied by average number of pointees per context-wise pointer). This gives a false positive rate of 0.5% per Equation 1. In practice we find that the loss in precision is not perceivable at all. The *NoAlias* percentage, a metric used in [13] (explained in Section 4), in this case for the approximate representation is exactly the same as that for an exact representation which takes significantly higher amounts of memory.

3.2 Querying the Multibloom

The ultimate goal of alias analysis is to answer whether two pointers p and q alias with each other either in a specific calling context or in a context-insensitive manner. We describe below how multibloom can be used to answer these queries.

Algorithm 2 Handling context-sensitive $DoAlias(q_1, q_2, c)$

```
Pidx_q1 = M_p[q_1], Cidx_q1 = M_c[c]
Pidx_q2 = M_p[q_2], Cidx_q2 = M_c[c]
for i = 1 to D do
  hasPointee = false
  for j = 1 to B do
    if mb[Pidx_q1][Cidx_q1][i][j] == mb[Pidx_q2][Cidx_q2][i][j] == 1 then
      hasPointee = true
      break
    end if
  end for
  if hasPointee == false then
    return NoAlias
  end if
end for
return MayAlias
```

Algorithm 3 Handling context-insensitive $DoAlias(q_1, q_2)$

```
for c = 1 to C do
  if DoAlias(q_1, q_2, c) == MayAlias then
    return MayAlias
  end if
end for
return NoAlias
```

Context-Sensitive Query A context-sensitive query is of type $DoAlias(q_1, q_2, c)$. To answer this query we need to first extract the $B \times D$ bit sets that belong to q_1 and q_2 under the context c. For each hash function the algorithm needs to determine if the corresponding bit vectors have at least one common bit with the value 1. If no such bit exists for any one hash function, then q_1 and q_2 do not alias. The pseudo-code is given in Algorithm 2. Note that this procedure is for single level pointers. In case q_1 and q_2 are higher-level pointers, the outermost for-loop of the procedure needs to be run for each value of s where $s \in [1..S]$.

Context-Insensitive Query A context-insensitive query will be of type $DoAlias(q_1, q_2)$. The query is answered by iterating over all possible values of the context c and calling the context-sensitive version of DoAlias: $DoAlias(q_1, q_2, c)$. Only if under no context do q_1 and q_2 alias, it concludes that there is no alias. The pseudo-code is shown in Algorithm 3.

4 Experimental Evaluation.

4.1 Implementation Details and Experimental Setup

All our implementation is done in the LLVM compiler infrastructure[13] and the analysis is run as a post linking phase. We implement two points-to analyses, one

which has an exact representation (without false positives) of the points-to set and the other uses our proposed multiblooom representation. For an exact representation we store pointees per context for a pointer using STL vectors[14]. Both versions are implemented by extending Andersen's algorithm [15] for context-sensitivity. They are flow-insensitive and field-insensitive implementations that use an invocation graph based approach. Each aggregate (like arrays and structures) is represented using a single memory location. Neither version implements optimizations like offline variable substitution[16].

Bench -mark	KLOC	Total Inst	Pointer Inst	No. of Fns
gcc	222.185	328425	119384	1829
perlbmk	81.442	143848	52924	1067
vortex	67.216	75458	16114	963
eon	17.679	126866	43617	1723
httpd	125.877	220552	104962	2339
sendmail	113.264	171413	57424	1005
parser	11.394	35814	11872	356
gap	71.367	118715	39484	877
vpr	17.731	25851	6575	228
crafty	20.657	28743	3467	136
mesa	59.255	96919	26076	1040
ammp	13.486	26199	6516	211
twolf	20.461	49507	15820	215
gzip	8.618	8434	991	90
bzip2	4.650	4832	759	90
mcf	2.414	2969	1080	42
equake	1.515	3029	985	40
art	1.272	1977	386	43

Table 1. Benchmark characteristics

Bench -mark	Memory (KB)		Precision	
	S=1	S=5	S=1	S=5
gcc	220973	302117.00	84.2	85.3
perlbmk	99346.9	143662.00	89.3	90.6
vortex	44756.4	62471.00	91	91.5
eon	108936	131552.00	96.3	96.8
httpd	221633	233586.00	92.8	93.2
sendmail	122310	127776.00	90.2	90.4
parser	23511.4	43093.10	97	98
gap	74914.8	84551.70	96.7	97.4
vpr	15066.4	23676.60	93.6	94.2
crafty	10223.9	10891.20	96.9	97.6
mesa	50389.7	55066.90	99.2	99.4
ammp	12735.8	15282.90	99.1	99.2
twolf	29037.2	33663.10	99.1	99.3
gzip	2807	3005.9	90.6	90.9
bzip2	2128.51	2333.82	87.7	88
mcf	2122.09	3758.17	94.5	94.5
equake	2245.6	3971.50	97.6	97.7
art	1090.72	1693.82	88.6	88.6

Table 2. Sensitivity to parameter S.

We evaluate performance over 16 C/C++ SPEC 2000 benchmarks and two large open source programs: *httpd* and *sendmail*. Their characteristics are given in Table 1. *KLOC* is the number of Kilo lines of code, *Total Inst* is the total number of static LLVM instructions, *Pointer Inst* is the number of static pointer-type LLVM instructions and *No. of Fns* is the number of functions in the benchmark. The LLVM intermediate representations of SPEC 2000 benchmarks and open source programs were run using *opt* tool of LLVM on an Intel Xeon machine with 2GHz clock, 4MB L2 cache and 3GB RAM. To quantify the loss in precision with a multibloom implementation, we use the *NoAlias* percentage metric used in LLVM. It is calculated by making a set of alias queries for all pairs of pointer variables within each function in a program and counting the number of queries that return *NoAlias*. Larger the *NoAlias* percentage, more precise is the analysis (upper bounded by the precision of the exact analysis).

We evaluate the performance of a multibloom for many different configurations and compare it with the exact implementation. In all evaluated configurations we allow the first dimension (P) to be equal to the number of unique pointers. We empirically found that the number of entries S for pointers pointed

Bench -mark	Precision (*NoAlias* %)					Memory (KB)				
		multibloom					multibloom			
	exact	4-4-10 tiny	8-8-10 small	8-12-50 medium	8-16-100 large	exact	4-4-10 tiny	8-8-10 small	8-12-50 medium	8-16-100 large
gcc	OOM	71.8	79.6	83.4	85.3	OOM	3956	15445	113577	302117
perlbmk	OOM	75.3	85.0	89.3	90.6	OOM	1881	7345	54008	143662
vortex	OOM	85.7	90.1	91.2	91.5	OOM	818	3194	23486	62471
eon	96.8	81.5	88.9	94.3	96.8	385284	3059	11942	87814	233586
httpd	93.2	90.1	92.1	92.9	93.2	225513	1673	6533	48036	127776
sendmail	90.4	85.6	88.2	90.3	90.4	197383	1723	6726	49455	131552
parser	98.0	65.8	97.3	97.9	98.0	121588	565	2204	16201	43094
gap	97.5	88.2	93.5	96.7	97.4	97863	1107	4323	31786	84552
vpr	94.2	85.9	93.9	94.1	94.2	50210	310	1211	8901	23677
crafty	97.6	97.1	97.6	97.6	97.6	15986	143	557	4095	10892
mesa	99.4	89.6	96.6	99.1	99.4	8261	721	2816	20702	55067
ammp	99.2	98.4	99.0	99.2	99.2	5844	201	782	5746	15283
twolf	99.3	96.7	99.1	99.3	99.3	1594	441	1721	12656	33664
gzip	90.9	88.8	90.5	90.8	90.9	1447	42	164	1205	3205
bzip2	88.0	84.8	88.0	88.0	88.0	519	31	120	878	2334
mcf	94.5	91.3	94.3	94.5	94.5	220	50	193	1413	3759
equake	97.7	96.9	97.7	97.7	97.7	161	52	204	1494	3972
art	88.6	86.6	88.4	88.6	88.6	42	23	87	637	1694

Table 3. Precision (*NoAlias* %) vs Memory (in KB). *OOM* means *Out Of Memory*.

to by two-level pointers gives a good trade off between memory and precision for $S = 5$. The hash family H, the context mapper M_c and the pointer-location mapper M_s are derived from the in-built pseudo random number generator. Many different combinations were tried for the other three dimensions: $C = (4, 8, 16)$, $B = (10, 20, 50, 100)$ and $D = (4, 8, 12, 16)$. From now on, when we report the results, we refer to the multibloom configuration by the tuple $(C-D-B)$. Below we report the results for select configurations that showed interesting behavior.

4.2 Tradeoff between Precision, Memory and Analysis Time

In Tables 3-4 we report the precision, time and memory requirements for various benchmarks. We compare 4 different multibloom configurations namely *tiny t* (4-4-10), *small s* (8-8-10), *medium m* (8-12-50) and *large l* (8-16-100) with *exact* which does not have any false positives.

Three out of the 18 benchmarks run out of memory when we run an exact analysis, highlighting the need for a scalable context-sensitive points-to analysis. All the multibloom configurations ran to completion successfully for these three benchmarks. The *tiny* configuration indicates significant reduction in both memory and analysis time. The memory requirement is three orders less, while the access time is reduced to about one-fourth for all benchmarks which take at least 20 seconds. The precision (in terms of *NoAlias* percentage) is within 7% for *tiny* of an exact analysis on average. At the other end, *medium* and *large* config-

Bench -mark	Precision (*NoAlias %*)					Time (s)				
		multibloom					multibloom			
	exact	t	s	m	l	exact	t	s	m	l
gcc	OOM	71.8	79.6	83.4	85.3	OOM.	791.705	3250.627	10237.702	27291.303
perlbmk	OOM	75.3	85.0	89.3	90.6	OOM.	76.277	235.207	2632.044	5429.385
vortex	OOM	85.7	90.1	91.2	91.5	OOM.	95.934	296.995	1998.501	4950.321
eon	96.8	81.5	88.9	94.3	96.8	231.166	39.138	118.947	1241.602	2639.796
httpd	93.2	90.1	92.1	92.9	93.2	17.445	7.180	15.277	52.793	127.503
sendmail	90.4	85.6	88.2	90.3	90.4	5.956	3.772	6.272	25.346	65.889
parser	98.0	65.8	97.3	97.9	98.0	55.359	9.469	31.166	145.777	353.382
gap	97.5	88.2	93.5	96.7	97.4	144.181	5.444	17.469	152.102	419.392
vpr	94.2	85.9	93.9	94.1	94.2	29.702	5.104	18.085	88.826	211.065
crafty	97.6	97.1	97.6	97.6	97.6	20.469	2.636	9.069	46.899	109.115
mesa	99.4	89.6	96.6	99.1	99.4	1.472	1.384	2.632	10.041	23.721
ammp	99.2	98.4	99.0	99.2	99.2	1.120	1.008	2.592	15.185	38.018
twolf	99.3	96.7	99.1	99.3	99.3	0.596	0.656	1.152	5.132	12.433
gzip	90.9	88.8	90.5	90.8	90.9	0.348	0.192	0.372	1.808	4.372
bzip2	88.0	84.8	88.0	88.0	88.0	0.148	0.144	0.284	1.348	3.288
mcf	94.5	91.3	94.3	94.5	94.5	0.112	0.332	0.820	5.036	12.677
equake	97.7	96.9	97.7	97.7	97.7	0.224	0.104	0.236	1.104	2.652
art	88.6	86.6	88.4	88.6	88.6	0.168	0.164	0.408	2.404	6.132

Table 4. Precision (*NoAlias* %) vs Time (in sec). *OOM* means *Out Of Memory*. *t* is *tiny*, *s* is *small*, *m* is *medium* and *l* is *large* configuration.

urations achieve full precision for all the benchmarks with significant savings in memory requirement for those requiring at least 15MB memory. However, this comes at a price in terms of analysis time. Thus *medium* and *large* are good configuration to use if precision is an absolute requirement. Even for the larger benchmarks they will lead to termination as they still provide a compact storage.

The *small* configuration proves to be an excellent trade off point. It achieves a good precision (within 1.5%) on average and achieves more than 10-fold memory reduction for benchmarks requiring more than 10MB memory for an exact analysis. It takes around the same amount of time on benchmarks that terminate with exact analysis. It should be noted that for smaller benchmarks (*mesa*, *ammp*, *twolf*, *gzip*, *bzip2*, *mcf*, *equake* and *art*) the configuration *small* requires more time than *exact* configuration. However, for larger benchmarks we see significant improvements in analysis time using bloom filter. One unique advantage of using multibloom is the user-control over various parameters to trade off precision for memory or vice versa. To reduce memory requirement for *medium* and *large*, we experimented with smaller values of S. The results for $S = 1$ versus $S = 5$ are given in Table 2 (memory in KB and precision as *NoAlias* percentage). We observe that with at most 1% reduction in average precision, we can obtain around 18% reduction in average memory requirement. In summary, a multibloom representation guarantees a compact storage representation for context-sensitive points-to analysis and allows the user to pick the configuration depending on whether analysis time or accuracy is more desirable.

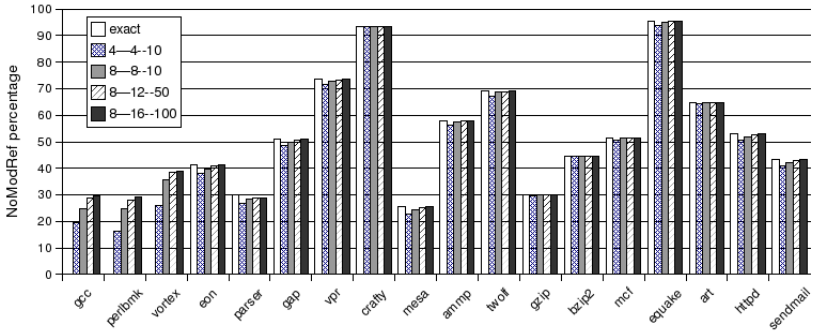

Fig. 4. Mod/Ref client analysis.

4.3 Mod/Ref Analysis as a Client to Points-to Analysis

Next we analyze how the loss in precision in the points-to analysis due to false positives affect the client analyses. We use the Mod/Ref analysis as the client of our multibloom based points-to analysis. For a query $GetModRef(callsite, pointer)$, the Mod/Ref analysis checks whether *callsite* reads or modifies the memory pointed to by *pointer*. It has four outcomes: (i) *NoModRef*: *call-site* does not read from or write to memory pointed to by *pointer*, (ii) *Ref*: *call-site* reads from the memory pointed to by *pointer*, (iii) *Mod*: *call-site* writes to (and does not read from) the memory pointed to by *pointer*, and (iv) *ModRef*: *call-site* reads from and writes to the memory pointed to by *pointer*. $ModRef$ is most conservative and should be returned when it is not possible to establish otherwise for a safe analysis. The more precise an approximate points-to analysis the more often will it answer *NoModRef* (upper bounded by an *exact analysis*). Figure 4 shows percentage of queries answered *NoModRef* by the analysis. From the figure, it can be seen that the *NoModRef* percentage with multibloom is 96.9% of the exact analysis even with a *tiny* configuration. For *small* configuration, it improves further to 98.7%. This shows that a client analysis is hardly affected due to loss in precision by using an approximate representation, while still enjoying the benefits of reduced memory and time requirements.

An important aspect of using multibloom is the provision of selecting a configuration on need basis. For more precise analysis, one can trade off memory and speed requirements by choosing larger values for C, D and B. For scalable analyses, one can reduce these values trading off some precision.

5 Related Work.

Many scalable pointer analysis algorithms are context- and flow-insensitive [1]. As scalability became an important factor with increasing code size, interesting mechanisms were introduced to approximate the precision of a full blown

context-sensitive and flow-sensitive analysis. [17] proposed *one level flow* to improve precision of context-insensitive, flow-insensitive analyses, still maintaining the scalability. Later, several inclusion-based scalable analyses were proposed [2][3][4], based on some novel data structures for points-to analysis like BDD. Similar to ours, several context-sensitive but flow-insensitive analyses have been recently proposed. Since inclusion-based analyses are costly, several unification-based algorithms were introduced, trading off precision for speed [1], [18]. Several context-sensitive algorithms proposed earlier [5][6][7][8] are flow-sensitive. Flow-sensitivity adds to precision but typically makes the analysis non-scalable. The idea of *bootstrapping* [19] enables context- and flow-sensitive algorithms to scale.

Various enhancements have also been made to the original Andersen's inclusion-based algorithm: online cycle elimination[20] to break dependence cycles on the fly, offline variable substitution[16] to reduce the number of pointers tracked during the analysis, location equivalence[21] and semi-sparse flow-sensitivity[22]. These enhancements are orthogonal to the usage of bloom filters. One can implement a points-to analysis with, for instance, online cycle elimination with points-to tuples stored in bloom filters and enjoy combined benefits.

Several novel data structures have been used in the last decade to scale points-to analysis, like ROBDD[2][23][9], ZBDD[24]. These data structures store exact representation of the points-to information and have no false positives. In contrast, bloom filters are useful for storing information in an approximate way. Also, our multibloom filter approach provides the user to control the memory requirement with a probabilistic lower bound on the loss in precision. Optimistic results for pointer analysis hint that bloom filters would be very useful for other compiler analyses as well.

6 Conclusions

In this paper we propose the use of multi-dimensional bloom filter for storing points-to information. The proposed representation, though, may introduce false positives, significantly reduces the memory requirement and provides a probabilistic lower bound on loss of precision. As our multibloom representation introduces only false positives, but no false negatives, it ensures safety for (may-)points-to analysis. We demonstrate the effectiveness of multibloom on 16 SPEC 2000 benchmarks and 2 real world applications. With average 4MB memory, multibloom achieves almost the same (98.6%) precision as the exact analysis taking about average 4 minutes per benchmark. Using Mod/Ref analysis as the client, we find that the client analysis is not affected that often even with some loss of precision in points-to representation. Our approach, for the first time, provides user a control on the memory requirement, yet giving a probabilistic lower bound on the loss in precision. As a future work, it would be interesting to see the effect of approximation introduced using bloom filters with the approximations introduced in control-flow analyses such as kCFA or in unification of contexts.

References

1. B. Steensgaard, "Points-to analysis in almost linear time," in *POPL*, 1996.
2. M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in *PLDI*, 2003.
3. N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: A million lines of C code in a second," in *PLDI*, 2001.
4. O. Lhotak and L. Hendren, "Scaling Java points-to analysis using spark," in *CC*, 2003.
5. M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *PLDI*, 1994.
6. W. Landi, B. G. Ryder, and S. Zhang, "Interprocedural modification side effect analysis with pointer aliasing," in *PLDI*, 1993.
7. J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs," in *OOPSLA*, 1999.
8. R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *PLDI*, 1995.
9. J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI*, 2004.
10. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," in *Communications of the ACM 13(7):422426*, 1970.
11. L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," in *SIGCOMM*, 1998.
12. R. Rugina and M. Rinard, "Pointer analysis for multithreaded programs," in *PLDI*, 1999.
13. http://llvm.org, "The LLVM compiler infrastructure,"
14. http://en.wikipedia.org/wiki/Standard_Template_Library, "Standard Template Library,"
15. L. O. Andersen, "Program analysis and specialization for the C programming language," in *PhD Thesis*, 1994.
16. A. Rountev and S. Chandra, "Offline variable substitution for scaling points-to analysis," in *PLDI*, 2000.
17. M. Das, "Unification-based pointer analysis with directional assignments," in *PLDI*, 2000.
18. M. Fahndrich, J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints," in *PLDI*, 2000.
19. V. Kahlon, "Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis," in *PLDI*, 2008.
20. M. Faehndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *PLDI*, 1998.
21. B. Hardekopf and C. Lin, "Exploiting pointer and location equivalence to optimize pointer analysis," in *SAS*, 2007.
22. B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *POPL*, 2009.
23. J. Zhu and S. Calman, "Symbolic pointer analysis revisited," in *PLDI*, 2004.
24. O. Lhotak, S. Curial, and J. N. Amaral, "Using ZBDDs in points-to analysis," in *LCPC*, 2007.