

Quantifying Uncertainty in Points-To Relations^{*}

Constantino G. Ribeiro and Marcelo Cintra

School of Informatics, University of Edinburgh
c.g.ribeiro@sms.ed.ac.uk;mc@inf.ed.ac.uk

Abstract. For programs that make extensive use of pointers, pointer analysis is often critical for the effectiveness of optimizing compilers and tools for reasoning about program behavior and correctness. Static pointer analysis has been extensively studied and several algorithms have been proposed, but these only provide approximate solutions. As such inaccuracy may hinder further optimizations, it is important to understand how short these algorithms come of providing accurate information about the points-to relations.

This paper attempts to quantify the amount of uncertainty of the points-to relations that remains after a state-of-the-art context- and flow-sensitive pointer analysis algorithm is applied to a collection of programs from two well-known benchmark suites: SPEC integer and MediaBench. This remaining static uncertainty is then compared to the run-time behavior. Unlike previous work that compared run-time behavior against less accurate context- and flow-insensitive algorithms, the goal of this work is to quantify the amount of uncertainty that is intrinsic to the applications and that defeat even the most accurate static analyses.

Experimental results show that often the static pointer analysis is very accurate, but for some benchmarks a significant fraction, up to 25%, of their accesses via pointer de-references cannot be statically fully disambiguated. We find that some 27% of these de-references turn out to access a single memory location at run time, but many do access several different memory locations. We find that the main reasons for this are the use of pointer arithmetic and the fact that some control paths are not taken. The latter is an example of a source of uncertainty that is intrinsic to the application.

1 INTRODUCTION

For programs that make extensive use of pointers, pointer analysis is often critical for the effectiveness of optimizing compilers and tools for reasoning about program behavior and correctness. Without accurate pointer analysis, data accesses through pointer de-references must be assumed to be directed to almost any program data, thus, making it impossible to accurately establish the flow of data. Pointer analysis has been extensively studied and several algorithms have been proposed (e.g., [5, 18, 21, 24, 25], see [9] for a comprehensive list). Completely

^{*} This work was supported in part by EPSRC under grants GR/R65169/01 and GR/S79572/01.

accurate pointer analysis (i.e., uniquely identifying the target of every pointer at every program point) is, in general, undecidable [11]. Thus, such algorithms are only approximate, and usually trade-off efficiency and accuracy. There is no consensus on what the best class of algorithm is and, in fact, many algorithms that theoretically vary significantly in accuracy actually perform very similarly in practice. Nevertheless, it is commonly accepted that context-sensitive and flow-sensitive algorithms are the most accurate ones.

Considering that context- and flow-sensitive pointer analysis represents the best that can be achieved with “general purpose” pointer analysis ¹, it is important to understand how short these algorithms come of providing accurate information about the points-to relations. An understanding of the sources of such uncertainty is critical for devising new heuristics that lead to unsafe but often accurate pointer analyses, such as probabilistic static pointer analyses [10]. Also, an understanding of the inaccuracy of pointer analysis is necessary in order to assess the performance impact that such inaccuracy may have on program optimization (e.g., [3, 4, 7]).

The main contribution of this paper is to systematically quantify the amount of uncertainty in the static may-alias points-to relations for two well-known classes of benchmarks and to compare this to the run-time behavior to quantify what fraction of this uncertainty is actually observed at run time. This paper also attempts to characterize the reasons for the differences between the static and the run-time results. Note that, unlike previous works [13, 15], the goal of this paper is not to quantify the amount of uncertainty arising from analyses that trade-off reduced precision for increased scalability. Instead, the goal of this paper is to quantify the amount of uncertainty that is intrinsic to the applications. By intrinsic we mean uncertainty that comes from the program structure and that would defeat any static analysis technique. For such a study we use, unlike previous work, a flow- and control-sensitive pointer analysis, which is the closest to the limit of what is possible with static analyses only.

More specifically, in this paper we quantitatively evaluate the uncertainty of the points-to relations that remains after a state-of-the-art context- and flow-sensitive pointer analysis algorithm [21] is applied to a collection of benchmarks from the well-known SPEC integer [23] and the MediaBench [12] suites. The static pointer de-references that exhibit uncertain points-to behavior are then instrumented and the actual run-time behavior is quantified. Experimental results show that for most of the benchmarks this static pointer analysis is very accurate, but for some benchmarks a significant fraction, up to 25%, of their accesses via pointer de-references cannot be statically fully disambiguated. We find that some 27% of these de-references turn out to access a single memory location at run time, but many do access several different memory locations. Further analysis shows that the main reasons for this are the use of pointer

¹ Other types of analyses, such as shape analysis (e.g., [6, 22]) may give further information about the behavior of pointers, but they only work for certain classes of applications.

arithmetic and the fact that some control paths are not taken. The latter is an example of a source of uncertainty that is intrinsic to the application.

The rest of the paper is organized as follows: Section 2 briefly overviews pointer analysis and the sources of uncertainty; Section 3 describes our methodology for quantifying uncertainty at compile and run time; Section 4 describes our empirical evaluation methodology; Section 5 presents the experimental results; Section 6 discusses related work; and Section 7 concludes the paper.

2 POINTER ANALYSIS

In this section we present a very brief and informal overview of pointer analysis. The sole goal is to provide a minimal understanding of the problems and, more importantly, the sources of uncertainty.

2.1 Basics

The goal of pointer analysis is to compute for every *program point* the set of memory objects that each pointer may be pointing to. For some simple algorithms there is a one-to-one correspondence between a program point and a source code line, while for context- and flow-sensitive analyses a program point is a source code line augmented with the context and flow information, so that, for instance, the same source code line if reached by two different paths can be treated as two different program points.

The granularity of individual memory objects may vary depending on the capability of the particular algorithm. Powerful algorithms can handle individual scalar variables and individual fields of complex data structures, and can also handle dynamically created memory objects. However, most algorithms treat whole arrays as a single memory object. Following the notation in [21], memory objects that can be individually named are associated with *location sets*, or *locsets* for short.

A common representation for pointers and their target memory locations is based on the notion of *points-to relations*, which are formed by tuples of the form (p, v) , where p is a pointer and v is some location set. These tuples are sometimes referred to as a *points-to relationship* between p and v . More formally, if P and V are the set of pointers and location sets, respectively, then $R \subset P \times V$ is a points-to relation and every tuple $(p, v) \in R$ implies that pointer p may point to location set v , which is represented by $p \rightarrow v$. Note that in languages that allow multiple levels of pointers (i.e., a pointer to a pointer, such as `int **p` in C) pointers can be themselves location sets and $P \subset V$. A common representation for a points-to relation is a *points-to graph*, which is a tuple $G = (N, E)$ of $N = P \cup V$ nodes and $E = R$ edges.

With this representation, the pointer analysis problem then results in computing the points-to graph for every program point. This is done by solving a set of dataflow equations using a fixed point algorithm. The dataflow equations are derived from the pointer manipulation operations allowed in the language.

For instance, the algorithm in [21] assumes the following four *basic pointer assignment operations*:

```
p1 = &p2; // Address-of assignment
p1 = p2;  // Copy assignment
p1 = *p2; // Load assignment
*p1 = p2; // Store assignment
```

where `p1` and `p2` are pointer variables. Note that these do not include pointer arithmetic, which is allowed in some languages such as C, but is not usually supported in existing pointer analysis frameworks.

After the dataflow equations have been solved, the resulting points-to graphs at all program points contain points-to relationships of two types: *definitely* points-to relationships (also known as *must alias*) and *possibly* points-to relationships (also known as *may alias*). A definitely points-to relationship (p, v) at some program point means that at this point p is for certain pointing to location set v . This implies that there is no edge leaving node p in the points-to graph other than the edge (p, v) , or, equivalently, that there is no tuple (p, u) in the points-to relation where $u \neq v$. A possibly points-to relationship (p, v) at some program point means that at this point p may point to location set v , but may also point to at least another different location set u . In this case we say that there is some *uncertainty* or *ambiguity* in the points-to relation.

Finally, changes in the points-to graph after processing some program point can be of two types: *strong updates* and *weak updates*. Strong updates are those that delete all the existing outgoing edges from a pointer p , while weak updates are those that simply add new edges without deleting any of the existing edges. For instance, the update at a program point that contains the assignment `p = &v` is strong as it deletes all edges (p, u) that may have existed before this program point. Note that the assignment `p1 = p2`, where both $p1$ and $p2$ are pointers, is by this definition a strong update (all previous edges from $p1$ are deleted) even if $p1$ is left with several possibly points-to relationships because of the possibly points-to relationships of $p2$. As explained in the next section, weak updates are the source of possibly points-to relationships and they appear due to a few different reasons.

2.2 Uncertainty in Pointer Analysis

Context- and flow-sensitive pointer analysis provides the most accurate static results, but it still cannot fully disambiguate all pointer de-references in many practical situations. Some of the most common reasons for this uncertainty are:

Control flow: A problem occurs when different control paths perform different updates to pointer variables. In this case, without dynamic knowledge of the actual program behavior, the static analysis can only assume that both updates are possible and at the merge point both targets are possible.

Pointer arithmetic: A problem occurs when the value of a pointer is updated through some arithmetic operation. In this case, even if the original target of the pointer is well-known, the final target can only be known if the pointer analysis algorithm has an accurate knowledge of the layout of objects in virtual memory.

Unavailable procedure code: A problem occurs when the original source code of a procedure is not available to the pointer analysis algorithm and the procedure takes a pointer as a parameter. In this case, unless the pointer analysis algorithm has some prior knowledge about the side-effects (or lack thereof) of the called procedure, the static analysis can only assume that after returning from the procedure the pointer may be pointing to any memory object.

Recursive data structures: A problem occurs when pointers are used to link objects from recursive data structures. Usually these form well structured forms such as lists, trees, circular queues, etc. However, traditional pointer analysis is not designed to recognize such structures and end up collapsing several objects into a single memory target. Shape analysis (e.g., [6, 22]) has been specially designed to handle such cases. This paper’s goal is to investigate the accuracy of “general-purpose” pointer analyses and a study of the effects of shape analysis is beyond its scope.

Aggregates: A problem occurs when pointers are used to access internal parts of an aggregate (e.g., an array or a structure) but the pointer analysis assigns a single name for the whole aggregate. In this case, the pointer analysis cannot disambiguate accesses to different parts of the aggregate.

Dynamically allocated objects: A problem occurs when pointers are assigned to dynamically allocated objects that are allocated at the same static code site. In this case most pointer analyses will simply assign a single name to the static memory allocation site and will not be able to disambiguate accesses to the (possibly) multiple objects that are allocated at the site. In fact, many pointer analyses tools are even less accurate and simply assign a single name to the whole heap area, so that even memory objects that are allocated at different static code sites end up being aliased.

3 QUANTIFICATION METHODOLOGY

3.1 Source Code Analysis

To collect the static points-to statistics we modify a context- and flow-sensitive pointer analysis algorithm [21] to count the number of accesses *through a pointer de-reference*, and for each access to count the number of possible target locsets as identified by the points-to graph immediately before the access. When analyzing the source code we count the number of locsets accessed (used or modified) through a pointer de-reference as follows. In these examples assume that p is a pointer variable and that $*^n$ represents n levels of indirection (e.g., $*^2 p$ is equivalent to $**p$).

Indirect use of a variable through a pointer de-reference (e.g., `...=*p`);

This is counted as one *use* via pointer.

Indirect modification of a variable through a pointer de-reference (e.g., `*p=...`);

This is counted as one *modification* via pointer.

Multi-level indirect use of variable through a pointer de-reference (e.g., `...=*np`);

This is counted as n *uses* via pointers. The number of possible target locsets is counted for each de-reference. For instance, in `...=*np`; if `p` may only point to a single target locset, but `*p` may point to two target locsets, then we count one use with a single target and one use with two targets.

Multi-level indirect modification of variable through a pointer de-reference (e.g., `*np=...`);

This is counted as $n - 1$ *uses* via pointers plus one *modification* via pointer. The number of possible target locsets is counted for each de-reference, as described above.

Procedure call (e.g., `foo(..., *p, ...)`);

This is counted as one *use* via pointer. Multi-level indirect uses are counted as described above.

Loops :

Accesses within loops are treated as *one* instance of one of the cases above.

Procedures :

Accesses within procedures are treated as *one* instance *per calling context*.

Also, languages like C allow right-hand-side expressions and boolean expressions to contain assignments, such as `while(*p=a)` or `if(a==(*p=b))`. Obviously, in this case we must appropriately account for the embedded modification.

3.2 Run-time Statistics Collection

To collect the dynamic points-to statistics we further modify the context- and flow-sensitive pointer analysis of [21] to insert additional profiling code just before accesses through a pointer de-reference that are identified as having multiple target locsets. When compiled, this profiling code will record all different run-time memory addresses touched via these pointer de-references and count the number of accesses to each different address.

Each run-time access profiled is given a unique identifier that contains the source code number, so that we can match the run-time accesses with their static access. Note, however, that two mismatches between static and dynamic statistics can happen. First, multiple static accesses identified by the pointer analysis algorithm may map to the same source code line and, thus, to the same run-time counter. This happens because the pointer analysis algorithm separates

static accesses according to their context. Second, not all static accesses may appear at run time if that portion of the code is not executed with the given input data.

4 EVALUATION SETUP

4.1 Applications

To quantify the uncertainty that is intrinsic to static context- and flow-sensitive pointer analysis, we use a subset of the SPEC2000 integer benchmarks [23] and of the MediaBench benchmarks [12] that are written in C ² These applications are representative of the workloads typical of workstations and desktop computing and are well-known for their intense use of pointers in many cases. For the runtime experiments the input sets used are the standard ones provided with each suite (*ref* for SPEC2000). Table 1 shows, for each application, the total number of lines of C code, the total number of location sets and of pointer location sets, the number of source code expressions that are uses through pointer de-references, and the number of source code expressions that are modifications through pointer de-references.

<i>Application</i>	<i>Suite</i>	<i>Lines of Code (KLOC)</i>	<i>Total (Pointer) Location Sets</i>	<i>Pointer Uses</i>	<i>Pointer Modifications</i>
164.gzip	SPEC int	9.1	1,750 (246)	113	43
175.vpr		17	3,959 (649)	960	428
181.mcf		1.9	506 (194)	16	13
186.crafty		12	4,920 (469)	4,716	672
197.parser		12	3,631 (917)	10,587	83
256.bzip2		2.9	887 (85)	4	0
300.twolf		17.5	5,262 (950)	751	79
epic	MediaBench	7.6	397 (105)	37	13
unepic		7.6	531(242)	18	6
mpeg2enc		8.5	2,179 (455)	116	276
mpeg2dec		4.9	1,605 (295)	140	85
g721-enc		1	393 (68)	2	0
g721-dec		1	122 (36)	4	2
gsmencode		5.8	448 (133)	22	0
gsmdecode		5.8	1566 (599)	168	31

Table 1. Application characteristics.

² The benchmarks not included from the suites are those either written in Fortran or C++, which are incompatible with SUIF, or those written in C that did not work with the *original* SPAN package (see Section 4.2 for details on the compilation infrastructure).

4.2 Static Analysis

The statistics collection methodology described in Section 3 was implemented on the SPAN tool [20], which is an add-on to the SUIF compiler [8] that implements the pointer analysis algorithm of [21]. We modified SPAN to record all instances of pointer de-references along with the number of possible targets as identified by SPAN and with the source code line number. The source code line number is useful for identifying instances where SPAN is able to distinguish the different calling contexts of the same source line. Uses and modifications via pointer de-references were counted separately.

Uses and modifications through pointer de-references that may find the pointer uninitialized (according to the SPAN analysis) result in SPAN adding a special location set, called `unk`, to the target set. We decided to count these cases separately. For instance, a pointer de-reference with two possible targets where one of them is `unk` is counted separately from other pointer de-references with two possible targets where both targets are well-defined user objects. The reason for highlighting the ambiguous points-to sets that include `unk` is because this is an important special case that may be treated differently by optimizing compilers and program understanding tools. For instance, an optimizing compiler may choose to ignore the `unk` target when performing an aggressive (possibly unsafe) optimization under the assumption that an actual occurrence of the `unk` target is highly unlikely. On the other hand, a program understanding tool would likely especially flag de-references with possible `unk` targets as they may suggest a bug in the code.

Finally, SPAN creates a single locset *per context* for each dynamic memory allocation call site, and calls these locsets `heap.X`, where `X` is a number that identifies the context. However, it cannot disambiguate further the accesses to different parts of the memory object. Again, we decided to count these cases separately because this is also an important special case. In fact, dynamically allocated memory objects seem to often require specialized analyses [1, 17].

4.3 Profiling Environment

To monitor the actual run-time behavior of static pointer de-references with multiple possible targets we further modified the SPAN tool to add the necessary profiling code. More specifically, at each static de-reference where the pointer may have multiple targets the tool inserts code to record the actual address accessed and to increment a counter per address seen so far. The resulting instrumented code is converted from the SUIF file format (`.spd`) to C code and this code is then compiled for the Intel x86 platform using gcc 3.4.4 and using the `-O2` optimization level.

5 EXPERIMENTAL RESULTS

5.1 Static Pointer Analysis Statistics

We start our study by measuring the amount of uncertainty resulting from the static pointer analysis. Table 2 shows the breakdown of the static accesses through pointer de-references according to the number of possible target memory locations, as given by SPAN. The table presents separate results for uses and modifications. The number of uses and modifications through pointer de-references in this table are often larger than those in Table 1 because of the context-sensitivity of the analysis. This can also be seen from the often great disparity between the number of accesses and the number of source code lines in Table 2. Note that in most cases the number of accesses through pointer de-references is only a small fraction of all static program references.

From this table we can see that the result of the context- and flow-sensitive static analysis of SPAN is fairly accurate and can unambiguously identify the target of the pointer de-references in all accesses for most applications and in more than 90% of the accesses for all but 3 applications. Across the whole suite 81% of all the accesses have a single unambiguously identified target. Nevertheless, for some benchmarks the amount of uncertainty is non-negligible, reaching up to 25% of the accesses for *197.parser*.

Another observation from these results is that often a large fraction of the accesses with multiple possible targets have `unk` as one of the targets (meaning that the pointer may be uninitialized at this point). The exception is *197.parser*. As previously explained, these represent a special case of uncertainty that may be treated differently by an optimizing compiler or a program understanding tool. We do not expect any of these `unk` targets to actually occur at run-time (Section 5.2).

Finally, we also note from these results that there are often many fewer modifications through pointer de-references than there are uses (1731 modifications versus 47230 uses). However, per application these modifications have a relatively larger amount of uncertainty than uses: e.g., 76% of modifications in *197.parser* have multiple possible targets versus 24% of uses.

5.2 Profiling Results

Run-time Uncertainty The first step in quantifying the run-time behavior of the ambiguous pointer de-references is to measure the number of different location sets actually touched by each static reference. Such results can be directly compared to those of Table 2 as these references correspond to those in white text with grey background in that table. Note that since the profiling framework annotates source code lines the run-time accesses reported here correspond to those reported *per source code line* in Table 2. Table 3 shows the breakdown of only those static accesses through pointer de-references that have 2 or more possible target memory locations according to the number of actual target memory locations touched. Again, the two sections of the table correspond to uses

Application	Uses (<i>u</i>) and Modifications (<i>m</i>) with <i>N</i> possible targets (including <i>unk</i> target, including <i>heap</i> target, number of source code lines)			
	<i>N</i> = 1	<i>N</i> = 2	<i>N</i> = 3	<i>N</i> > 3
gzip	u: 277 m: 43	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
vpr	u: 2488 m: 428	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
mcf	u: 67 m: 13	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	6 (0,0,3) 0 (0,0,0)
crafty	u: 4970 m: 479	542 (534,67,59) 47 (45,11,9)	2 (2,2,1) 146 (146,66,13)	119 (0,26,24) 0 (0, 0, 0)
parser	u: 25178 m: 20	241 (241,241,35) 32 (32,32,6)	36 (0,0,11) 0 (0,0,0)	7841 (181,230,259) 31 (9,4,9)
bzip2	u: 119 m: 0	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
twolf	u: 3687 m: 77	6 (6,0,6) 2 (2,0,2)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
epic	u: 156 m: 13	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
unepic	u: 59 m: 6	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
mpeg2enc	u: 395 m: 276	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
mpeg2dec	u: 499 m: 75	8 (8,8,2) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	6 (6,6,1) 10 (10,10,2)
g721-enc	u: 22 m: 0	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
g721-dec	u: 6 m: 2	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
gsmencode	u: 154 m: 0	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)
gsmdecode	u: 346 m: 31	0 (0,0,0) 0 (0,0,0)	0 (0,0,0) 0 (0,0,0)	9 (0,0,9) 0 (0,0,0)

Table 2. Breakdown of static accesses according to the number of possible target memory locations. Results for the source code analysis. The first number (top-left) in each entry is the *total* number of accesses in that category. The numbers in parenthesis are: the number of accesses that have *unk* as one of the targets, the number of accesses that have *heap* as one of the targets, and the number of static source code accesses (as opposed to per-context). For instance, *186.crafty* has 542 uses through pointer dereferences with two possible targets; of these, 534 have *unk* as one of the targets, 67 have *heap* as one of the targets; and these 542 uses appear in only 59 source code lines. The entries in white text with grey background are those that reflect ambiguity in the static analysis and are instrumented for the run-time statistics collection.

and modifications, respectively. Note that some of the static references are not actually executed with the input sets used. Finally, note that in this experiment a static reference is said to touch two or more target memory locations as long as at least two of more of its dynamic instances touch different memory locations.

Application	Uses with N actual targets				Modifications with N actual targets			
	NE	$N = 1$	$N = 2$	$N > 2$	NE	$N = 1$	$N = 2$	$N > 2$
mcf	1	2	0	0	-	-	-	-
crafty	59	1	1	23	17	0	0	5
parser	193	27	0	85	6	1	0	8
twolf	1	5	0	0	2	0	0	0
mpeg2dec	2	0	0	1	1	0	0	1
gsmdecode	0	9	0	0	-	-	-	-

Table 3. Breakdown of static accesses with 2 or more possible target memory locations (Table 2) according to the number of actual target memory locations. Results for the profile analysis. NE stands for static accesses that are not executed. For instance, of the $59+1+24=84$ source code lines with pointer de-references with two or more possible targets in *186.crafty* (Table 2), 59 are not executed, 1 has only a single target at run time, 1 has two targets at run time, and 23 have three or more targets at run time. The entries in white text with grey background are those that reflect actual ambiguity at run time. The entries with light grey background are those where the static ambiguity disappears at run time.

From this table we can see that some (27% of the executed accesses) of the uncertainty of the static analysis disappears at run time and actually a single memory location is accessed. Nevertheless, a significant fraction of the accesses indeed turn out to point to more than one different memory location at run time. The next section discusses in more detail the reasons for the differences between static and dynamic results.

Causes of Uncertainty A closer inspection at the actual outcomes of the ambiguous static references reveals that several factors contribute to the difference between the static and dynamic behaviors. Table 4 shows the causes for this difference and the number of instances of each cause. The references here correspond to all of those in Table 3.

From this table we can see two directions of variation: from fewer possible targets of the static analysis to more actual targets at run time, and from more possible targets of the static analysis to fewer actual targets at run time. There are two major factors affecting those variations. One is the use of pointer arithmetic, which, interestingly, turns out to produce variations in both directions. Another is the fact that some control paths are simply not taken at run time. We also note that many variations come from the use of structures and arrays, which may throw off the static analysis.

<i>Behavior difference</i>		<i>Cause</i>	<i>Number of cases</i>
<i>Static</i>	<i>Actual</i>		
2 or more targets	Not executed	-	282
2 targets (inclusive <i>unk</i>)	Single target	Pointer turns out to be always initialized	6
2 targets	3 or more targets	Pointer arithmetic to index into array-like object	22
		Use of arrays	2
		Use of recursive data structures	5
3 or more targets	Single target	Use of structure fields	2
		Pointer arithmetic to index into array-like object	9
		Control path alternative never taken	28
No change	-	-	95

Table 4. Classification of dynamic accesses according to the difference with respect to the static behavior and according to the cause for the difference

Variations with Input Sets Finally, to assess the sensitivity of the run-time results with respect to input data we repeated some of the experiments with the SPEC benchmarks with the *train* input sets. Significant variability in the run-time points-to behavior with different input data would indicate that techniques that rely on profiling to refine the results of the static analysis are likely to fail. Naturally, the converse is not necessarily true: little variability in the run-time points-to behavior does not guarantee that profiling will work well for all types of feedback-directed analysis. This occurs when the profile-directed analysis is not directly driven by the points-to behavior, but by some other run-time behavior. For instance, probabilistic pointer analysis [10] uses the frequency of path execution to estimate the probability of points-to relations. Nevertheless, little variability in the run-time points-to behavior is a good indication that profile-directed analyses are likely to often work well.

Our experiments show very little to no variability in the run-time behavior of the points-to relations between executions with the *ref* and *train* input sets. A similar result was obtained in [15].

6 RELATED WORK

Pointer analysis has been extensively studied and several algorithms have been proposed. A comprehensive list of pointer analysis research and some discussion on open problems can be found in [9]. The work in [21] introduced the flow- and context-sensitive pointer analysis algorithm that is implemented in the tool we used to gather the static points-to information. Despite not being as fast and efficient as more recent summary-based pointer analysis algorithms (e.g., [18]), it is still reasonably fast and can handle non-trivial programs in practice.

Works that propose and evaluate new pointer analysis algorithms usually present a quantitative breakdown of the number of possible target memory locations. However, this often consists of only the static statistics (quite often just average points-to set sizes) and does not include a quantification of the actual run-time behavior.

Recently, a few works have attempted to investigate the impact of pointer analysis on overall compiler optimization [3, 4, 7]. These works show that pointer analysis is often critical to enable a large number of optimizations. They also indirectly quantify the amount of run-time uncertainty and its impact on optimizations, but they do not provide a full quantification of such run-time uncertainty.

Researchers have since long attempted to extend static analysis with information that better reflects the actual run-time behavior. One such approach is probabilistic static analyses. The work in [19] developed a framework to integrate frequency, or probability, into a class of static dataflow analyses. The framework uses control flow frequencies to compute the probabilities of execution of all possible control flow paths and then uses this information to assign frequencies of occurrence to dataflow facts. This framework has been extended and applied to the problem of pointer analysis in [10], in which case the dataflow facts of interest, for which a frequency of occurrence is sought, are the points-to relationships. That work also quantitatively compared the static points-to results against run-time behavior, but this evaluation was limited to only portions of a few simple benchmarks.

The closest work to ours is [15], which also systematically attempted to quantify the run-time behavior of points-to sets. Our work differs from that work in that we are interested in measuring the gap between the run-time behavior and the best static analysis in order to assess the limitations of purely static analysis and the potential benefits of different extended approaches (e.g., probabilistic points-to heuristics). For this reason we use a context- and flow-sensitive algorithm as the baseline for comparisons, instead of the scalable algorithms used in that work, and we concentrate our evaluation on those references where the static analysis computes points-to sets with more than one element. The work in [13] is similar to [15], but in the context of reference analysis for Java programs, and also used a variation for Java of a context- and flow-insensitive pointer analysis algorithm.

Finally, there has been recently a significant interest in speculative compiler optimizations based on imprecise dataflow and pointer analyses. The cost analyses of such optimizations require a good knowledge of the expected run-time behavior of dataflow and points-to relationships. Our work attempts to systematically evaluate the run-time behavior of points-to relationships and is an important step in the way of developing effective cost analyses for speculative compiler optimizations. Examples of speculative compiler optimizations requiring knowledge of run-time behavior of points-to relationships include: [14], which performs speculative partial redundancy elimination in EPIC architectures with

speculative loads; [2], which performs speculative parallelization; and [16], which performs program slicing for a program understanding tool.

7 CONCLUSIONS

In this paper we attempted to systematically quantify the amount of uncertainty due to may-alias points-to relations for two well-known classes of benchmarks. Unlike previous works [13,15] that consider pointer analysis algorithms that trade-off reduced precision for increased scalability, in this paper we are interested in the amount of uncertainty that is intrinsic to the applications and that defeat even flow- and control-sensitive pointer analysis.

We performed our evaluation applying a state-of-the-art context- and flow-sensitive pointer analysis algorithm [21] to a collection of benchmarks from the well-known SPEC integer [23] and the MediaBench [12] suites. Experimental results show that for most of the benchmarks this static pointer analysis is very accurate, but for some benchmarks a significant fraction, up to 25%, of their accesses via pointer de-references cannot be statically fully disambiguated. We find that some 27% of these de-references turn out to access a single memory location at run time, but many do access several different memory locations. Further analysis shows that the main reasons for this are the use of pointer arithmetic and the fact that some control paths are not taken. These results suggest that some further compiler optimizations may be possible by exploiting the cases where the uncertainty does not appear at run time, but for this to happen it is necessary to improve the handling of pointer arithmetic and to develop probabilistic approaches that capture the actual control flow behavior.

References

1. R. Z. Altucher and W. Landi. “An Extended Form of Must Alias Analysis for Dynamic Allocation.” *Symp. on Principles of Programming Languages*, pages 74-84, January 1995.
2. P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. “Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis.” *Symp. on Principles and Practice of Parallel Programming*, pages 25-36, June 2003.
3. B. Cheng and W.-M. Hwu. “Modular Interprocedural Pointer Analysis using Access Paths: Design, Implementation, and Evaluation.” *Conf. on Programming Language Design and Implementation*, pages 57-69, June 2000.
4. M. Das, B. Liblit, M. Fähndrich, and J. Rehof. “Estimating the Impact of Scalable Pointer Analysis on Optimization.” *Intl. Static Analysis Symp.*, pages 260-278, July 2001.
5. M. Emami, R. Ghiya, and L. J. Hendren. “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers.” *Conf. on Programming Language Design and Implementation*, pages 242-256, June 1994.
6. R. Ghiya and L. J. Hendren. “Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C.” *Symp. on Principles of Programming Languages*, pages 1-15, January 1996.

7. R. Ghiya, D. Lavery, and D. Sehr. "On the Importance of Points-To Analysis and Other Memory Disambiguation Methods for C Programs." *Conf. on Programming Language Design and Implementation*, pages 47-58, June 2001.
8. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S-W. Liao, E. Bugnion, and M. Lam. "Maximizing Multiprocessor Performance with the SUIF Compiler." *IEEE Computer*, Vol. 29, No. 12, pages 84-89, December 1996.
9. M. Hind. "Pointer Analysis: Haven't We Solved This Problem Yet?" *Wksp. on Program Analysis for Software Tools and Engineering*, pages 54-61, June 2001.
10. Y.-S. Hwang, P.-S. Chen, J. K. Lee, and R. D.-C. Ju. "Probabilistic Points-to Analysis." *Intl. Wksp on Languages and Compilers for Parallel Computing*, pages 290-305, August 2001.
11. W. Landi. "Undecidability of Static Analysis." *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 4, pages 323-337, December 1992.
12. C. Lee, M. Potkonjak, W. H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." *Intl. Symp. on Microarchitecture*, pages 330-335, December 1997.
13. D. Liang, M. Pennings, and M. J. Harrold. "Evaluating the Precision of Static Reference Analysis Using Profiling." *Intl. Symp. on Software Testing and Analysis*, pages 22-32, July 2002.
14. J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. "A Compiler Framework for Speculative Analysis and Optimizations." *Conf. on Programming Language Design and Implementation*, pages 289-299, June 2003.
15. M. Mock, M. Das, C. Chambers, and S. J. Eggers. "Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization." *Wksp. on Program Analysis for Software Tools and Engineering*, pages 66-72, June 2001.
16. M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. "Improving Program Slicing with Dynamic Points-to Data." *Intl. Symp. on Foundations of Software Engineering*, pages 71-80, November 2002.
17. E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. "Importance of Heap Specialization in Pointer Analysis." *Wksp. on Program Analysis for Software Tools and Engineering*, pages 43-48, June 2004.
18. E. M. Nystrom, H.-S. Kim, and W.-M. Hwu. "Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis." *Intl. Static Analysis Symp.*, pages 165-180, August 2004.
19. G. Ramalingam. "Data Flow Frequency Analysis." *Conf. on Programming Language Design and Implementation*, pages 267-277, May 1996.
20. R. Rugina and M. Rinard. "Span: A shape and Pointer Analysis Package." Technical report, M.I.T. LCSTM-581, June 1998.
21. R. Rugina and M. Rinard. "Pointer Analysis for Multithreaded Programs." *Conf. on Programming Language Design and Implementation*, pages 77-90, May 1999.
22. M. Sagiv, T. Reps, and R. Wilhelm. "Parametric Shape Analysis via 3-valued Logic." *Symp. on Principles of Programming Languages*, pages 105-118, January 1999.
23. Standard Performance Evaluation Corporation. www.spec.org/cpu2000
24. B. Steensgaard. "Points-to Analysis in Almost Linear Time." *Symp. on Principles of Programming Languages*, pages 32-41, January 1996.
25. R. P. Wilson and M. S. Lam. "Efficient Context-Sensitive Pointer Analysis for C Programs." *Conf. on Programming Language Design and Implementation*, pages 1-12, June 1995.