View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Edinburgh Research Archive

Understanding Uncertainty in Static Pointer Analysis

Constantino Goncalves Ribeiro

Master of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2008

Abstract

For programs that make extensive use of pointers, pointer analysis is often critical for the effectiveness of optimising compilers and tools for reasoning about program behaviour 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 optimisations, it is important to understand how short these algorithms come of providing accurate information about the points-to relations.

This thesis 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 behaviour. Unlike previous work that compared run-time behaviour 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.

In a first step to quantify the uncertainties, a compiler framework was proposed and implemented. It is based on the SUIF1 research compiler framework and the SPAN pointer analysis package. This framework was then used to collect extensive data from the static points-to analysis. It was also used to drive a profiled execution of the programs in order to collect the real run-time points-to data. Finally, the static and the run-time data were compared.

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.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Marcelo Cintra, for his support and help throughout all my study and his example and dedication as supervisor, teaching me and show me to be a researcher, and for making. I want to thank my second supervisor, Dr. Michael O' Boyle for his critical advices during my study and panels. I would also like to thank everyone in our research group for sharing our common interests and discussions.

I'd like to thank my friends Dr. Ernesto Neto and Dr. Jose Carlos for the valuable counsels and opinions. Finally I would like to thank my wife, Julia for her patience and support even not been with me during these years in UK.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- Constantino G. Ribeiro and Marcelo Cintra. ”Quantifying Uncertainty in Points-To Relations.”*International Workshop on Languages and Compilers for Parallel Computing*”, pages 190-204, November 2006.

Table of Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions of this Thesis 3
1.3 Thesis Structure 4

2 Background **5**
2.1 Compiler Optimisations 5
2.2 Pre-requisites of Compiler Optimisation: Analyses 6
2.3 Previous Work 15
2.4 Probabilistic Points-to Analysis 18
2.5 Dynamic Points-to Analysis 20
2.6 Summary 21

3 Quantifying Uncertainty in Points-to Relations **22**
3.1 Uncertainty in Pointer Analysis 22
3.2 Quantification Methodology 28
3.3 Run-time Statistics Collection 29
3.4 Summary 29

4 Results **30**
4.1 Evaluation Setup 30
4.2 Experimental Results 36
4.3 Profiling Results 37
4.4 Causes of Uncertainty 41
4.5 Variations with Input Sets 42
4.6 Summary 42

5 Conclusions and Future Work **43**
5.1 Conclusions 43
5.2 Future work 44

Bibliography **46**

List of Figures

2.1 An example code and its control-flow graph. 8
2.2 A program and its corresponding call-graph 9
2.3 A snippet of a program with no data dependence and its corresponding data-flow graph. 10
2.4 A snippet of a program with true data dependence and its corresponding data-flow graph. 11
2.5 A snippet of a program with anti-dependence and its corresponding data-flow graph. 11
2.6 A snippet of a program with output data dependence and its corresponding data-flow graph. 11
2.7 A point in the program code and its corresponding points-to graph. . . 13
2.8 A point in the program code and its corresponding points-to graph. . . 14

3.1 The control flow generates two possible address at d = *p (block B4), so there is uncertainty in that part of the program. 23
3.2 The inability of compiler to handle arbitrary arithmetic operations on pointers can generate imprecise points-to sets when updating the value of p. 23
3.3 The compiler must have some knowledge of the side-effects in this called procedure to produce an accurate points-to set of p. 25
3.4 An example of recursive data structure using linked structured objects in pointers operations. 26
3.5 The compiler can't distinguish difference between the elements *type1[21]* and *type2[22]* of x, an array of aggregate data structure. 26
3.6 The compiler assigns the same address to both dynamically allocated pointers. 27

4.1 Example of program in C and location sets annotations. 33

4.2 Example of program in C and points-to graphs. 34
4.3 Example of program in C and an intermediate points-to graph. 35
4.4 Example of program with different procedure call contexts. 36

List of Tables

4.1 Application characteristics. 31

4.2 Breakdown of static accesses according to the number of possible target memory locations. Results for the source code analysis for SPEC2000 programs. 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 de-references 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. 38

4.3 Breakdown of static accesses according to the number of possible target memory locations. Results for the source code analysis for Media-Bench programs. 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). 39

4.4 Breakdown of static accesses with 2 or more possible target memory locations (Table 4.3) 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 4.3), 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 were the static ambiguity disappears at run time. 40
4.5 Classification of dynamic accesses according to the difference with respect to the static behaviour and according to the cause for the difference 41

Chapter 1

Introduction

1.1 Motivation

Research on compilers spans many decades, and can be said to have been born with the first programming language [1]. In the current stage of development and research, compilers can be divided into two parts: front-end and back-end [2]. The front-end is the part that takes care of lexical analysis, parsing and semantic analysis. The back end is the part that, using the results of the front-end, generates an appropriate machine code to be executed on the hardware.

Compilers often make many optimisations to improve the performance of programs [2, 3]. These optimisations are made by the back-end using some form of intermediate code generated by the compiler front-end. Many of these optimisations require an accurate knowledge of the behaviour of control and data flow of the programs being optimised. The complete knowledge of how the control and data flows behave in a program is often hard to obtain due to some features present in some programming languages such as: the use of procedures, complex data types, recursive function calls, function pointers, among others [4]. The conjunction of these difficulties often makes the analysis work, and consequently the optimisations, a hard, and sometimes imprecise task. Although there is much research on this, it still constitutes a challenge. Perfect data-flow analysis in the general case has been proven to be an undecidable problem [5]. Thus, solutions to this problem are likely to be approximate. Moreover, often the most complex, and more precise analysis, which could give better results, is avoided by compiler designers because of their costs [2, 6]. Many compilers use simpler and faster analysis resulting in less precise, but faster, results.

In order to perform a data-flow analysis it is first necessary to perform control-flow

and call-graph analyses. Together, the control-flow graph and the call graph describe all possible relationships among individual instructions and procedures of a program. The control-flow graph can then be used to perform a data-flow analysis where we try to know how a program manipulates data at each level of its call graph. The data-flow analysis describes the behaviour, in terms of uses and definitions, of every data in every point of the program, thus, establishing the dependences among them [6]. A subset of data-flow analysis is the points-to analysis. It is performed using a subset of the variables of a program, namely the pointer variables. It is a difficult analysis as pointer variables have a very flexible behaviour and can be used as arguments in procedure calls and can sometimes point to other pointers that can also point to other pointers, and so on. A precise data-flow analysis, and consequently points-to analysis, gives a detailed dependence graph describing all possible data dependences among the instructions in the program.

After knowing the behaviour of all variables of a program, as they are manipulated by procedures and individual instructions, the compiler can try to optimise the code generated. This optimisation task must respect the control- and data-flow relationships among instructions exposed by both the control-flow and data-flow analyses without violating any dependence, in conservative compiler environments. This strategy wastes many possible more aggressive optimisation opportunities and may lead to a less optimised code. Alternatively, recent compilers use speculative execution as a means to enable further optimisations. In this kind of execution, possible data and control dependences may not need to be respected by the compiler. For instance, the Advanced Load is an optimisation method used in the Intel^{TM} Itanium^{TM} processor [7], to reduce the latency of costly load operations. In anticipation that a data item will be required in the future an advanced load is performed, even without confirmation that the item will be need. The Intel^{TM} Itanium^{TM} uses the Advanced Load Address Table (ALAT) to store information about advanced load instructions that are used while in speculative execution. In this system the advanced load order is issued by an *ld.a* instruction, which allocates an entry in the ALAT and starts the data transfer. The system checks the success of the advanced load with a *ld.c* or *chk.a* instruction that looks for the related information in the ALAT. The improvement in performance in the speculative execution of programs is achieved when demand cache misses are executed earlier and overlapped with computation.

Another important example is Thread-Level Speculation (TLS), which is a technique that improves sequential program performance by dividing conventional sequen-

tial programs into multiple small threads [8, 9] and speculatively runs them in parallel. The monitoring of this execution is done by special hardware that checks the writes of data by earlier threads against the reads by later threads. If a data flow violation occurs the later threads that read a value too soon are squashed and restarted to read the right value.

Both methods are promising uses of speculative execution and both require more accurate (although perhaps not safe) knowledge of the data flow in the program. In fact, to exploit such techniques we need a new methodology of data-flow analysis that takes in account the number of possible behaviours exhibited and their frequency of occurrence. An understanding of the variation between the static analysis result and the frequency in which the actual behaviours appear in the normal execution is necessary to help calculate the improvements of using speculative execution, and so, to justify its use. This data must be produced in the analysis phase to be used by the optimiser. So the optimiser can estimate whether the speculation may improve the performance and use this information as a guide to decide which opportunities of speculative execution it will use in the optimisation. There have been some works on how to use probabilistic analysis techniques [10, 11, 12, 13] to guide the speculative optimisation of programs. This work is related to those and aims to provide a better understanding of the behaviour of pointer variables and the reasons of why many opportunities of more aggressive optimisations are lost by conservative compilers during their points-to analysis. We propose a methodology to analyse and quantify the points-to behaviour, and specially the differences between static estimations and run-time behaviour, and understand the reasons for such differences.

1.2 Contributions of this Thesis

This thesis proposes a method to quantify points-to uncertainty in points-to sets of programs. This method is used to evaluate the uncertainty present in applications from two very important classes of applications. It does so using context- and flow-sensitive pointer analysis in order to avoid uncertainty simply due to less aggressive analyses. The thesis also identifies the main reasons for the discrepancies between the actual run-time behaviour of such points-to sets and the behaviour expected from the static pointer analysis. To the best of my knowledge this is the first work that quantifies points-to uncertainty in this level of detail and qualifies the causes of that uncertainty.

An important result is that discovering the reasons for the discrepancies between

static and run-time points-to sets can lead to improvements in the precision of points-to analysis, leading to better and most precise dependence graphs that will give a better support for future more aggressive speculative optimisations.

1.3 Thesis Structure

This thesis is structured in two parts. Part I has three chapters: Chapter 1 provides an introduction; Chapter 2 describes the background information about compiler optimisations, control- and data-flow analysis and points-to analysis and discusses related work.

Part II has three chapters: Chapter 3 describes in detail the methodology for quantifying uncertainty in points-to relations; Chapter 4 presents results of static points-to analysis and the dynamic profiled execution; and Chapter 5 presents conclusions and discusses future work.

Chapter 2

Background

In this chapter we first present briefly the common compiler analyses related to this thesis. We then present and discuss in more detail the recent body of research more directly related with this thesis.

2.1 Compiler Optimisations

Modularity is a powerful methodology to construct good programs. It leads to structured and easy to understand programs. Modularity is achieved with the use of procedures. These are very useful as they can be seen as interfaces and black boxes and this concept allows for a better program abstraction, design and maintenance [2]. But the use of procedures may inhibit code optimisation, producing less efficient codes. One problem is that procedures make it harder to collect control and data-flow information [2]. In a less precise analysis a compiler can assume that a procedure can affect (by using and/or changing) any global variable when it is called and that at any call site, any arbitrary variables can be provided as parameters of that call. These problems combined with the intense use of procedures and pointers in modern program languages, can lead to a loss of many optimisation opportunities, generating a poor intermediate code.

A way to mitigate the effects of problems like the ones exposed above is to perform some optimisations over the intermediate code generated by the compiler front-end. These optimisations occur after the compiler back-end performs some analysis over the intermediate code collecting some data to guide the future optimisation. These analyses are complex and will give information to guide modifications and/or optimisations in the intermediate code to improve the future machine code. These modifi-

cations are hardware/architectural dependent. Common optimisations include: loop fusion, in-lining, suppression of variable aliases, and others. The most sophisticated ones can even provide more aggressive optimisations like: code parallelisation, speculative execution and so on.

2.2 Pre-requisites of Compiler Optimisation: Analyses

As discussed in Section 2.1 compiler optimisations are key to generating efficient machine code. Before the compiler can perform optimizations it needs to know how the program behaves. Collecting knowledge of possible paths (control flow) and variable usage behaviour and the relationships between them is the first step to before optimization. The analysis of the program code gives a description of these behaviours.

2.2.1 Control-flow Analysis

The control-flow analysis is performed over the intermediate code and it construct control-flow graph which shows possible points to be executed next, at each point in a program. The following terms are used in control-flow graphs:

- Entry block: It is the block through which all control flow enters the graph.
- Exit block: It is the block through which all control flow leaves the graph.
- Back edge: It is an edge that points to an ancestor node in a depth-first (DFS) traversal of the graph.
- Critical edge: It is an edge that is neither the only edge leaving its source block, nor the only edge entering its destination block. These edges have to be split so a new block is created in the middle of the edge to insert computations on the edge.
- Abnormal edge: It is the edge whose destination is unknown and it tends to inhibit optimisation. They exist because of exception handling constructs.
- Impossible or fake edge: It is an edge that has been added to the graph just to make the exit block postdominate all blocks and it cannot be traversed.
- Dominator: We say that a block M dominates block N if every path from the entry that reaches block N has to pass through block M.

- Postdominator: We say that a block *M* postdominates block *N* if every path from *N* to the exit has to pass through block *M*.
- Immediate dominator: We say that a block *M* immediately dominates block *N* if *M* dominates *N*, and there is no intervening block *P* such that *M* dominates *P* and *P* dominates *N*.
- Immediate postdominator: We say that a block *M* immediately posdominates block *N* if *M* posdominates *N*, and there is no intervening block *P* such that *M* posdominates *P* and *P* posdominates *N*.
- Dominator tree: It is a data structure that describes the dominator relationships. It shows an arc from block *M* to block *N* if *M* is an immediate dominator of *N*.
- Postdominator tree: Similar behaviour of dominator tree.
- Loop header: It is the target of a loop-forming back edge and it dominates all blocks in the loop body.

The control-flow analysis is performed over the abstract syntax tree generated by previous steps of the compiler front-end. An example of small program and its control-flow graph is shown in Figure 2.1.

For a given program, a control-flow graph *G* is defined as a finite set *N* of nodes and a finite set *E* of edges. An edge *(i, j)* in *E* connects two nodes n_i and n_j in *N*. The notation *G= (N, E)* is used to describe a control-flow graph *G* with *N* nodes and *E* edges. So in a program, each basic block turns into a node of the control-flow graph, and the flows of control between them are the edges of its control-flow graph. The blocks and nodes are labelled so a block b_i corresponds to node n_i and an edge *(i, j)* connecting basic blocks b_i and b_j denotes that control goes from block b_i to block b_j. The set *N* has a start node that has no incoming edge and an end node that has no outgoing edge. In a control-flow graph *G = (N, E)*, a sequence of *k* edges, $k > 0$, (e_i, e_{i+1}, e_k), means that there is a path of length *k* through the flow graph.

Another important and related analysis is the call-graph analysis. It describes control flow relationships among procedures of a program and is performed over the abstract tree. It generates a static call graph where each node represents a procedure directed edges between nodes represents the calling order of the procedures. So in a given program *P* with p_1,..., p_n procedures, a static call graph of *P* is the graph $G = < N, S, E, r >$ with node set $N = p_1, ..., p_n$, the set *S* of a call site labels, the set $E \subseteq N x N$

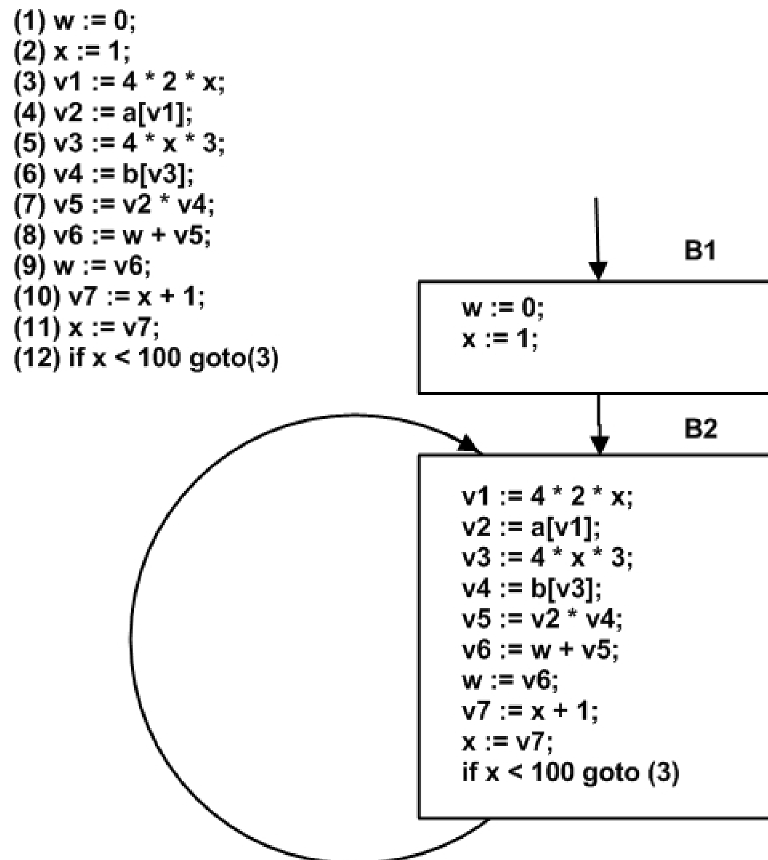

Figure 2.1: An example code and its control-flow graph.

of labelled edges, and the distinguished entry node $r \in N$ (that is the Main procedure). For each e = <p_i, s_k, p_j>,s_k denotes a call site in p_i from which p_j is called (e.g., the source code line number of the call site). Figure 2.2 shows an example of a program and its corresponding call-graph.

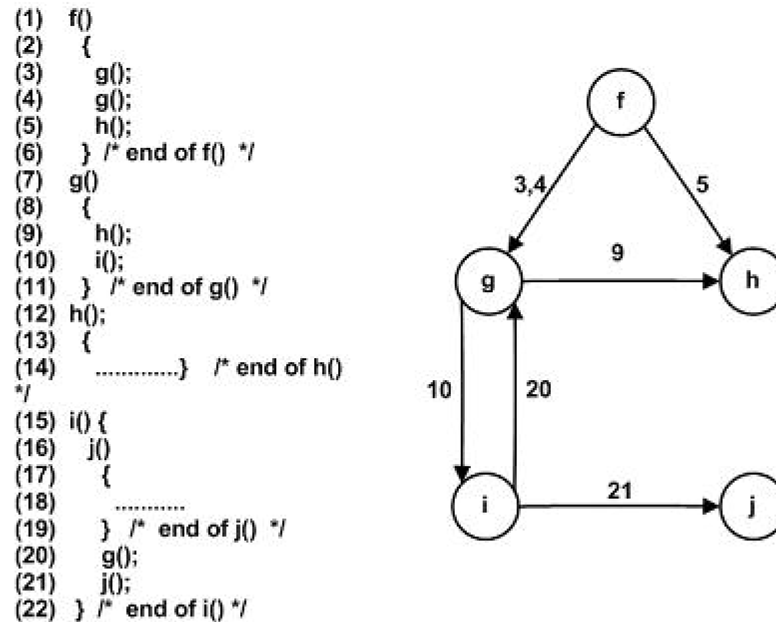

Figure 2.2: A program and its corresponding call-graph

2.2.2 Data-flow Analysis

The data-flow analysis collects information about possible flows of data between points of a program. It uses the control-flow graph and call-graph generated by control-flow analysis and call-graph analysis. Data-flow analysis is performed using data-flow equations that compute the possible variable values for each node of the control-flow graph, and is often described as a difficult and generally intractable task [2, 3]. These equations compute the possible variable values for each node of the control-flow graph, using four basic data sets:

- GEN: data items created in a specific basic block;
- KILL: data items invalidated in a specific basic block;
- IN: data items getting into a specific basic block;
- OUT: data items getting out a specific basic block;

These sets are applied in the basic blocks where nodes are single instructions and variables are edges.

Its precision depends on many factors and sometimes compilers use less sophisticated data-flow analyses to make it feasible in a production environment. The data-flow analysis can be flow-sensitive when it takes into account the side effects of different control paths in the program, and context-sensitive when it takes into account the side effects of different set of parameters used in different call sites of the same procedure. Another issue that improves the precision of data-flow analysis is the granularity of individual memory objects and how they are analysed: handling individual scalar variables and individual fields of complex data structures and even handling dynamically created objects. These three factors, namely control-flow sensitivity, context sensitivity, and representation and granularity of objects have a significant impact on the precision and complexity of data-flow analyses solutions. The result of the data-flow analysis is the data flow graph (DFG). A data-flow graph (DFG) is a graph that represents data dependences between a number of operations and can expose and represent data constraints. Figures 2.3, 2.4, 2.5 and 2.6 represent basic blocs with no data-dependences, true data-dependence, anti-dependence and output dependence respectively.

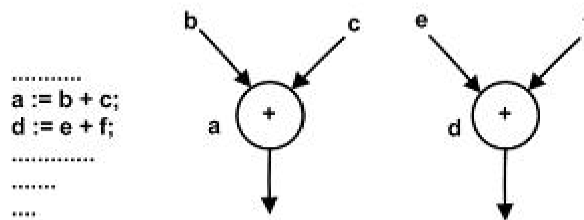

Figure 2.3: A snippet of a program with no data dependence and its corresponding data-flow graph.

A DFG is defined as $G_{(DFG)} = (V_{(DFG)}, E_{(DFG)}))$ directed graph where $V_{(DFG)}$ is the set of vertices representing operations and $E_{(DFG)} \subseteq V x V$ is the set of precedence edges where $E_{d(DFG)} \subseteq E$ is the set of data dependence edges (values) and $E_{s(DFG)} \subseteq E$ and $E_{d(DFG)} \cap E_{s(DFG)} = \emptyset$.

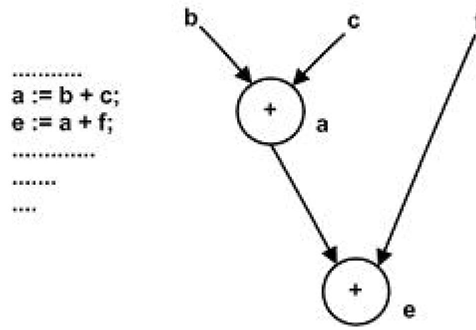

Figure 2.4: A snippet of a program with true data dependence and its corresponding data-flow graph.

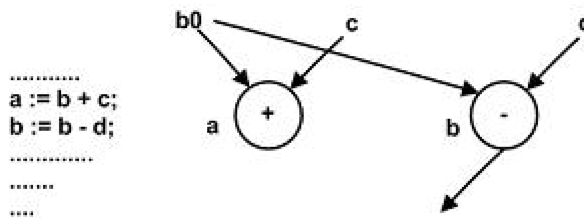

Figure 2.5: A snippet of a program with anti-dependence and its corresponding data-flow graph.

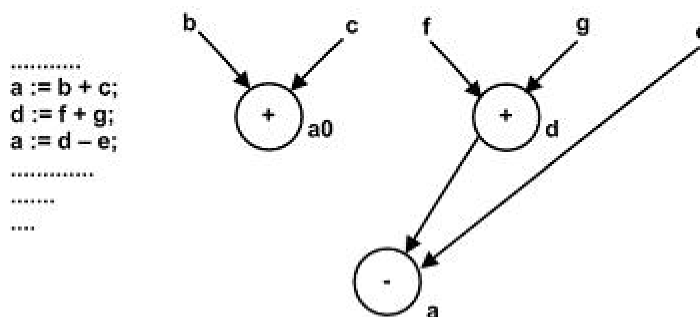

Figure 2.6: A snippet of a program with output data dependence and its corresponding data-flow graph.

2.2.3 Points-to Analysis

Points-to analysis is a subset of data-flow analysis that deals specifically with flow of data through pointer variables. The goal of pointer analysis is to compute for every program point the set of memory objects that each pointer variable 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 precision of points-to analysis depends on how the analyse algorithm works. It can be: context sensitive and / or flow sensitive and how it abstract memory granularity. A context-sensitive algorithm takes in account different sets (values) of parameters used in a procedure or function and makes new analysis for each call of a procedure with new sets of parameters because these new values can generate new sub-graphs that will affect the main graph in the return of the procedure call to the program main flow. This approach produces a better result. A flow-sensitive algorithm takes in account the possible many paths due to control-flow structures and procedures and function calls present in the program, generating sub-graphs that are merged with the may graph. The granularity is the way which complex data structures like structures and arrays are abstract as memory location. If they are treated as one simple locations set we have a less precise representation and consequently less precise points-to analysis. If we have a detailed abstraction where the individual elements of these complex structures are treated as individual location sets we have more precise points-to analysis. Another important thing is how the analysis will treat the dynamically allocated memory objects like heaps. If it handles this unique location set for all dynamic memory allocated objects we have a less precise solution and if it handles each new dynamic memory allocated object as a new location set it produces a more precise result. All these tree issues can produce smaller points-to sets because they can generate fewer points-to relationships with less ambiguity. Following the notation in [14] 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 $R \subset P\ X\ V$ 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 $p \rightarrow v$. 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.

The points-to graph is the result of computing the points-to set 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 [14] 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. We show an example of program code and its corresponding points-to graph in figure 2.7.

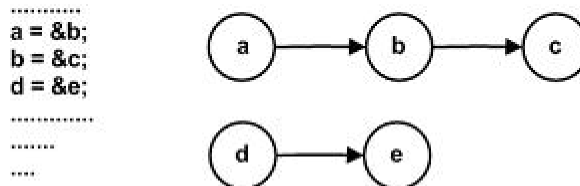

Figure 2.7: A point in the program code and its corresponding points-to graph.

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 pointer 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 pointer 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 before, weak updates are the source of possibly points-to relationships and they appear due to a few different reasons.This uncertainty or ambiguity is generated by more than one possible path that a program can expose during the analyse process. For example consider the fragment of a program in figure 2.8

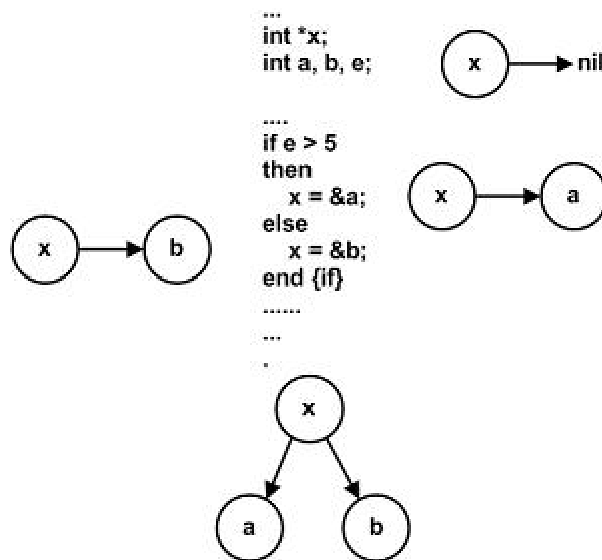

Figure 2.8: A point in the program code and its corresponding points-to graph.

The control structure gives two possible ways to the flow according to the value of the variable *e*. So, at the end of the control flow structure (if), the analysis merge of the two points-to graphs of the control flow structure (one generated for the *then* case and other generated for the *else* case), resulting in a new graph with two possible points-to values for pointer *x*. Of course, during execution only one value will be taken because only one path will be taken.

2.3 Previous Work

In this section we discuss the most relevant work in our research area. As discussed in the sections before, the program data-dependence analysis is the basis for the alias and points-to analysis. In a data-dependence analysis we establish the dependence between operations in epochs of a program. These sets of dependences can be represented by a data-flow graph. The control-flow graph is used to perform alias analysis using data-flow equations and it gives the sets of GEN, KILL, IN and OUT. These sets of variables are used for basic code optimisations. The points-to analysis tries to capture the behaviour of a special kind of variable: pointer. As a pointer has a characteristic behaviour and is very flexible in its use, a different and more specific analysis is necessary. The points-to analysis is performed over the data-flow graph and it generates points-to sets of a program.

2.3.1 Alias Analysis

Alias analysis is key to several code optimisation techniques and there have been numerous studies about it. A review of this extensive literature would be too lengthy and beyond the scope of this thesis. Here we limit ourselves to work more closely related to ours. In [15], R. A. Chowdhury et al, study using four alias analysis algorithms: Address-taken, Steesgaard, Shapiro-Horwitz and assume no alias. It applies the major scalar optimisations proposed by them and compare their obtained results (compile time, optimisations opportunities, optimisations in sequence, optimisations enabled independently) to compare the precision among these algorithms that use different approaches. The results of this study helps to the choose which algorithm to use when an additional accuracy is need. A framework for memory disambiguation that generates symbolic probabilities of the alias to array references in programs, using the concepts of dependence analysis is proposed by [11], where the symbolic probabilities are generated using a cost model to determine whether a data speculation is profitable or not before applying a set of proposed heuristics for generating the approximate aliasing probabilities. An algorithm for alias analysis of executable code of a program instead the source code is described in [16] by S. Debray and R. Muth and M. Weippert, and it uses a scheme that performs the analysis using the instruction inspection technique and abstract address sets concept.

An important tool used to represent variable aliases in the analysis process is the Static Single Assignment (SSA) form [17]. A derivative form of SSA, when there

are aliases, is proposed in [18] by F. Chow and S. et al, and results in a better SSA representation called HSSA (Hashed SSA). In [19], R. Lo et al, again an extension of the method used in SSA forms is proposed, to control alias and give support for possible speculative code optimisations.

2.3.2 Points-to Analysis

The basic alias analysis generates acceptable results to be used in the optimisation step, but it is not well suited to handle a special type of variable, namely pointers, which need a different analysis formulation. Pointers, heaps, and pointer functions, present in some languages have special behaviours so a different analysis is necessary. This is called pointer analysis (or points-to analysis) and we describe the most significant works in this area in this section. An excellent summary of past work can be found in [5].

A study of the influence of naming schemes and granularity on pointer analysis, and a new approach that combines profiling techniques and compiler analysis based on Intel's ORC for different naming schemes, is proposed in [20] by T. Chen and J. Lin, W. Hsu and P. Yew, to improve the performance of programs. Basic pointer assignment statements, dataflow equations for basic statements, and dataflow information for basic statements are described in [14]. An interprocedural, flow-sensitive and context-sensitive pointer analysis algorithm, for multithreaded programs, is proposed and implemented in the development environment SUIF 1 [21, 22].

In [23], R. P. Wilson and M. S. Lam, partial transfer functions are proposed to provide context sensitivity to a pointer analysis, by describing the behaviour of a procedure, taking in account that some alias relationship is held when the procedure is called. Also low-level representation of memory (memory divided into blocks of contiguous storage in local, global and return variables), locations sets (represent a block storage) concepts are described and an algorithm for context-sensitive pointer analysis that resumes the effects of procedures using these concepts, is proposed. A new context-sensitive interprocedural method to points-to analysis, is proposed in [24] by M. Emami and R. Ghiya and L. J. Hendren, using the concepts of abstract locations, R-locations and L-locations, basic points to analysis rules, interprocedural rules for points-to analysis and that handles with function pointers. A Storage Shape Graph (SSG), which summarises all pointer paths into and through allocated storage that could arise by execution of any path to the statement after the analysis, and its opera-

tions (add nodes, add edges, merge nodes) are defined in [25] by M. Hall et al. There, this structure is used in a proposed algorithm to analyse programs that contain lisp-like structure. It uses a framework that makes simple references to nodes of control-flow graph with fixed number of fields, some of which are pointers, and summarises the information in the SSG. In [26], M. Hind et al, the major tasks performed for three proposed algorithms (flow-sensitive interprocedural; flow-insensitive intarprocedural and a flow-insensitive interprocedural) for alias analysis are described. Theses algorithms use call graphs, data-flow equations and sets of variables (MOD, USED, KILL) to make their analysis. An interprocedural flow-sensitive points-to analysis that uses non-standard sets of types for storage, typing rules based on this set of non-standard set of types, storage shape graph and type inferences, is described in [27] by R. Ghiya and D. Lavery and D. Sehr, and performs a almost linear time pointer analysis.

An extension of [27], that results in a new flow-insensitive pointer analysis algorithm, is proposed in [28] by M. Shapiro and S. Horwitz, and it is more precise because it produces a smaller set of alias pairs. In [29], R. Ghiya and D. Lavery and D. Sehr, a framework for memory disambiguation that has many sources and clients together is proposed. It uses seven methods of points-to analysis (direct memory disambiguation, simple base and offset indirect (sbo indirect) analysis, array data dependence analysis, interprocedural points-to analysis, global, interprocedural flow-insensitive points-to analysis, type-based), abstract memory locations (LOCs) and SSA concepts. It was evaluated using standard benchmarks on the ORC Itanium^{TM} development environment. A comparative study of how the data are collected from 27 mid-sized programs in the points-to analysis is conducted in [30] by P. Anderson et al, and shows the importance of this information in future possible works. An empirical comparison of the effectiveness of the major pointer analysis algorithms on C programs is described in [31] by M. Hind and A. Pioli, and the results and empirical data are used in the executions are used as way to evaluated the different algorithms for points-to analysis.

In [32], E. M. Nystrom and H.S. Kim and W.M. Hwu, a context-sensitive pointer analysis is studied and its accuracy and scalability are analysed. This analysis is made in two phases: first a bottom-up step propagates analysis results from callees and callers and then a top-down step computes the actual pointer data. The pointer analysis uses Andersen's Algorithm (bottom-up and top-down context-sensitive solution) and the problems caused by procedural side effects were removed using concise procedure summaries that can be removed later if it is necessary.

A modular interprocedural pointer analysis is describes in [33] by K. Olukotun et

al. It can be combined with other analyses to improve precision of the analysis in programs with type cast and function pointers. It is based on access-paths and the authors claim that it can reduce the overhead for context-sensitive transfer functions and can make the distinction of non-recursive heap objects.

In [34], R. Z. Altucher and W. Landi, methods to improve the quality of def-use sets for dynamic allocated objects are proposed. They are based in Ruggieri/Murtagh naming scheme for dynamically created locations concepts using expanded naming scheme to handle with dynamically allocated locations.

2.3.3 Shape Analysis

A shape analysis to deal with disambiguation of heap-allocated data structures in C programs is proposed in [35] by R. Ghiya and L. J. Hendren. A context-sensitive interprocedural analysis method is described using the McCAT compiler to improve the tests of dependence and for parallelisation giving a more precise heap analysis giving better and helpful analysis results.

2.4 Probabilistic Points-to Analysis

The analysis methods explained above used to analyse common variables, pointers, heaps and pointer functions are conservative ones and sometimes this simple alias and pointer analyses are not sufficient to describe the run-time behaviour of program variables. In fact, many believe that even advanced static analyses are insufficient to capture the run-time behaviour. Some enhanced techniques have been proposed to give a more accurate knowledge of the possible behaviours of these variables. Some of them use probabilistic rules; others use profiling execution to try to obtain a more precise picture of this problem. This is mainly because the static analysis assumes that all control paths can be executed with equal likelihood and that all targets in an alias or points-to set can be accessed with equal likelihood. So a more aggressive approach, albeit speculative, is to add probabilities to the alias and points-to relationships. The use of probabilities and speculation methods may increase the opportunities for optimisations even with the risk of executing an incorrect operation.

In [36], M. Fernandez and R. Espasa, the shortcomings of alias analysis by simple instruction inspection and residue-based global alias analysis is described and to improve the precision of the analysis it proposes two algorithms (region-based spec-

ulative alias and profile-guided speculative alias analysis) that speculatively execute a may-alias analysis with a new variable: the safeness. The shortcomings of static, dynamic and hybrid disambiguation methods are described in [37] by A. S. Huang, G. Slavenburg and J. P. Shen, and a speculative disambiguating one, that has better performance and precision than conservative disambiguation methods, especially for non-linear patterns of memory access, is proposed and implemented in a VLIW architecture.

The mechanism to evaluate the degree of dependence of speculative threads and when to turn off this mechanism is studied in [38] by J. Lin et al. It describes a cost model to be used by the compiler in the decision and the evaluation of the degree of dependence is made by probabilistic points-to analysis where the probabilities of relationships are calculated and the degree of dependences between loop iterations is qualitatively computed. In [39], T. Chen et al, the way to deal with important issues of data dependence profiling to avoid the high latency of these tools is described. Issues like: data dependence detection using shadow variables, handling function calls in the profiling, the behaviour of the loops and dependence probability are discussed and some solutions are proposed to increase the efficiency of data profiling.

In [12] the alias relationships are classified in two groups: the ones that must or definitely point to an address during all possible executions and ones that may or possibly point to an address that may occur in some executions. This information is not enough to support more aggressive compiler optimisation because it does not describe how often these aliases may occur. So it proposes incorporating probabilistic rules and operations to the points-to analysis to attempt to improve the accuracy of static analyses. This new framework was incorporated into SUIF and MachSUIF and the results show improvement in the results of pointer analysis in several benchmark programs.

Conventional points-to analysis produces sets of pairs of *definitely*, *definitely not* and *may* points-to at any point of a program. *May* points-to are not optimised in many compilers to ensure correctness. Speculative optimisations can use these imprecise information especially when it has some data about how often (frequency) this *may* points-to occurs. In [13] a probabilistic pointer analysis algorithm is proposed. It produces statically and probabilistic data about points-to relationships of a program using linear transfer functions. The results are encoded as sparse matrices and optionally it can use simple control-flow edge profiling. It supplies pairs of points-to relationships with their probabilistic data and shows good results even when not using profiling information and analysing big SPEC2000 integer benchmark programs.

2.5 Dynamic Points-to Analysis

Using probabilities to further qualify the results of static points-to analysis closes somewhat the gap between static and run-time results, but there is still a large difference between the predicted static results and the real behaviour. The following works try to establish why difference occurs.

In [40], M. Mock et al, the behaviour of pointers in C programs is compared using pointer analysis algorithms against the real behaviour of pointer variables during program execution. It shows that scalar pointer analyses produce very different information from the run-time observation and this shows that a better knowledge can be produced by pointer profile data values, which can be used to improve performance of programs. It is the work that has most similarity with ours. The differences between them are: [41] makes a comparison between the singleton points-to sets of static analysis and dynamically collected and our work makes a more specific collection of static and dynamic points-to sets (not only singleton sets); [41] compares the static result of three different points-to analysis algorithms and we use only one (the most precise one). We took in account the sets with unk locations and the sets that analyse dynamic allocated pointers (heap) and showed that they are a source of possible inaccuracies and we tried to find out the main culprites of the breakdown between static and run time results.

The difference is that our work makes a more precise discrimination of points-to sets with more options and taking in account the sets with unk locations and the sets that include dynamically allocated variables (heap). We try to find out the main culprites of the difference between static and run time results.

Experimental data, using different versions of iterative algorithms to solve data-flow analysis were shown in [41] by K. D. Cooper and T. J. Harvey and K. Kennedy. The reducibility role is explained using Kam-Ullman time bound; it shows the difference of behaviour between versions of the iterative algorithms and provides practical advices to improve performance of interactive solver.

A new approach, that duplicates, isolates, and eliminates hot paths in data-flow graphs, is proposed in [42] by M. Mock et al. This improves the precision of data flow analysis by eliminating the unnecessary un-profitable paths and the results show good improvement in path qualification.

In [43], D. Liang and M. Pennings and M. J. Harrold, study reference information used in Java programs for knowledge of dereference instances. Run-time collected data

of dynamic reference information is used as reference information and its precision is measured and examples where this approach is not adequate are discussed. The data collected can be used in other similar analyses improving their precision.

The difficulties of dealing with the static alias analysis for modern programs is described in [44]. It shows those difficulties proving that interprocedural May Alias is not a recursive problem and that interprocedural Must Alias is not a recursively enumerable problem showing that those analyses are far from being precise and much more has to be done specially for dynamically allocated structures.

The still unsolved problems of Pointer Analysis, such as precision; accuracy; scalability and so on are described in [5]. It proposes a better knowledge of the problem to identify its needs and its issues for optimisations and program understanding tools using adequate metrics to evaluate it. It suggests that additional information must be incorporated to future analysis to address the discussed issues.

2.6 Summary

In this chapter the background foundations of our work were explained. We explained why we need to optimise program codes. Descriptions of the main analyses performed by the compiler over the intermediate code generated in the first steps of program code compilation were described and how they are related with each other. The most relevant research works in alias and points-to analysis were briefly described to supply an overview of the research in this area.

Chapter 3

Quantifying Uncertainty in Points-to Relations

In this chapter we will describe our work. The possible reasons for uncertainty when analysing the use of pointers in a program are described and a methodology to quantify this uncertainty is presented.

3.1 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 behaviour, the static analysis can only assume that both updates are possible and at the merge point both targets are possible like in the example in figure 3.1.

Pointer arithmetic: A problem occurs when the value of a pointer is updated through some arithmetic operation, as show in figure 3.2. 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 as we mentioned before. Many time the points-to analysis use a simple representation of array as unique location set assigned to the head of the array and any other position of the array is threaded as an offset position, and this approach can generate a misinterpretation of the compiler when analysing arithmetic involving pointers and arrays.

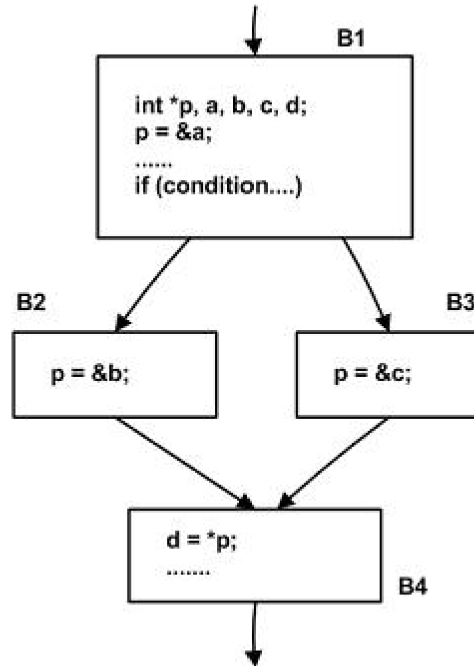

Figure 3.1: The control flow generates two possible address at $d = *p$ (block B4), so there is uncertainty in that part of the program.

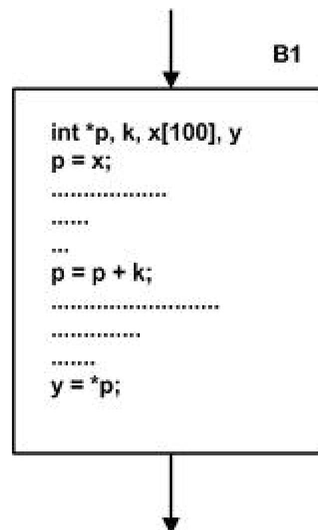

Figure 3.2: The inability of compiler to handle arbitrary arithmetic operations on pointers can generate imprecise points-to sets when updating the value of p.

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. An example is shown in figure 3.3.

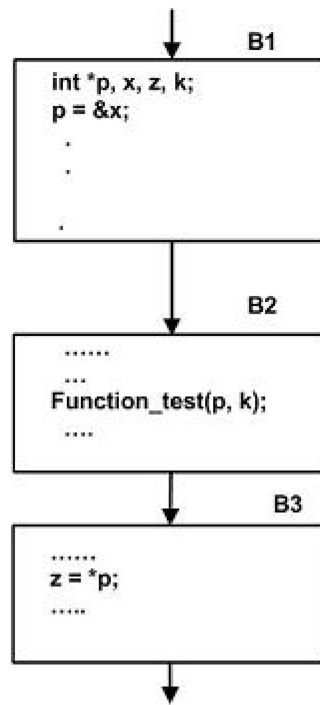

Figure 3.3: The compiler must have some knowledge of the side-effects in this called procedure to produce an accurate points-to set of p.

Recursive data structures: A problem occurs when pointers are used to link objects from recursive data structures, like in figure 3.4. 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., [45, 35]) has been specially designed to handle such cases. This papers 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., a structure). As discussed before many compiler use a simple approach and assign a single location set for the whole structure. Others, like SPAN, assign single location set for the whole structure and use offset to distinguish the different fields. Both approaches cause a lost of precision and make the pointer analysis less precise and it can not disambiguate accesses to different parts of the aggregate in the example of figure 4.5 where we have an array of aggregate that make the analysis even more difficult.

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

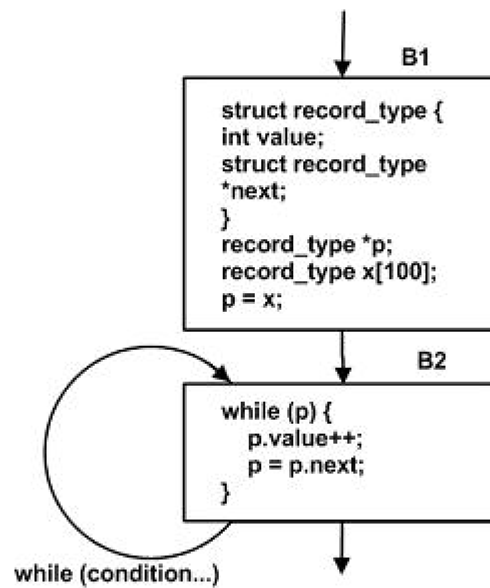

Figure 3.4: An example of recursive data structure using linked structured objects in pointers operations.

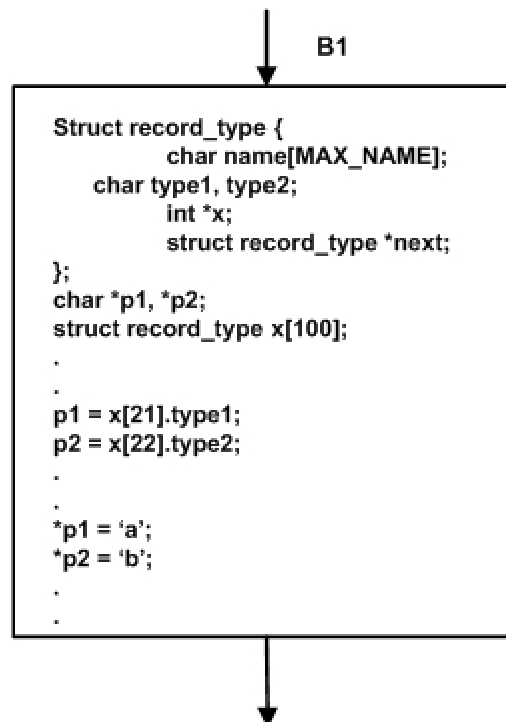

Figure 3.5: The compiler can't distinguish difference between the elements *type1[21]* and *type2[22]* of *x*, an array of aggregate data structure.

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. A code example is shown in figure 3.6.

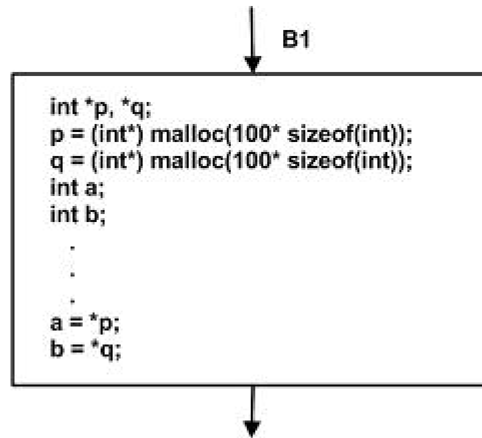

Figure 3.6: The compiler assigns the same address to both dynamically allocated pointers.

3.2 Quantification Methodology

3.2.1 Source Code Analysis

To collect the static points-to statistics we modify a context- and flow-sensitive pointer analysis algorithm [14] 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., ...= $*^n p$;): This is counted as n uses via pointers. The number of possible target locsets is counted for each de-reference. For instance, in ...= **p; if p may only point to a single target locset, but *p may point to two target locsets, and 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., $*^n p$ =...;): 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.

3.3 Run-time Statistics Collection

To collect the dynamic points-to statistics we further modify the context- and flow-sensitive pointer analysis of [14] 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.

3.4 Summary

In this chapter we explained the possible causes of uncertainty during the static points-to analysis. We propose a methodology to collect and quantify a statically generated points-to set data as a extension of a SUIF framework package. We also propose a methodology to modify the original programs code with new commands to generate profiling data to collect the run-time data.

Chapter 4

Results

4.1 Evaluation Setup

In this chapter we analyse the results of our quantitative study of static and profiled points-to sets. The results of static analysis are plotted and compared with the results of profiled execution of the programs. Each case is broken down and analysed, and the culprits of discrepancies are identified.

4.1.1 Applications

To quantify the uncertainty that is intrinsic to static context- and flow-sensitive pointer analysis, we used a subset of the SPEC2000 integer benchmarks [46], www.spec.org/cpu2000, and of the MediaBench benchmarks [47], C. Lee and M. Potkonjak and W. H. Mangione-Smith that are written in C. 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.

The reason why not all programs of SEPC 2000 and Media Bench benchmark were not used was that they did not complete the static analysis. As we modify the SPAN package to include a new set of variables that save how many times these variables were called with the calling context, this new feature sometimes exceeded the maximum size of a variable name and caused the crash of the running analysis. There was not time to solve this problem, so we did not use some programs.

For the run-time experiments the input sets used are the standard ones provided with each suite (ref for SPEC2000). Table 4.1 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.

Application	Suite	*Lines of Code (KLOC)*	*Total (Pointer) Location Sets*	*Pointer Uses*	*Pointer Modifications*
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 4.1: Application characteristics.

4.1.2 Static Analysis

The statistics collection methodology described in Section 3 was implemented on the SPAN tool [48, 49], which is an add-on to the SUIF compiler [22, 25] that implements the pointer analysis algorithm of [14].It is a context- and flow-sensitive points-to SUIF 1 package that performs points-to analysis. It works over SUIF1 program intermediate code annotation as input and generates a SUIF1 output file with the annotated pointer information of the analysed program. It uses location sets as abstract model to representing physical memory locations so the annotated information about pointers is presented by locations sets. Different locations sets mean different abstract locations

that mean different physical memory locations. It associates the location sets to all variables in the program and it includes procedure parameters and heap-allocated variables. The location set is represented by a unique integer identifier in the output file so different location sets are assigned to different identifiers. SPAN uses the location sets to make two kinds of annotations during program analysis: one associated with variable symbols and a second associated with instructions (de-references one: lod, str and memcpy) and this second gives information about pointers. Figure 4.1 gives an example program and two examples of each annotation.

When one de-reference annotation associated with instructions is performed, the location set information is updated and a new points-to graph is generated as shown in the figure 4.2.

Many points-to analysis algorithms use the location set notation, but SPAN produces a more precise points-to information due to its flow- and context sensitive analysis. When a procedure call occurs SPAN uses the parameters passed in this procedure call to generate a new intermediate points-to graph and at the return of the point, merge the intermediate graph with the main points-to graph as shown in the figure 4.3.

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. For example in the figure 4.4 if we have a procedure g with the parameters x, y and w in epoch of a program and, in another epoch the same procedure is called but with different parameter (a, b and c for example). As SPAN is context-sensitive it will execute a new analysis to take in account possible new points-to values due to the new context. 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 optimising 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 oc-

```
void swap(int **p, int **q)
{
 int *tmp;
 int x, z, k;
 if (p > q) then
     {
      tmp = *p;
      *p = *q;
      *q = tmp;
 endif
}


main()
{
  int a, b, c, e;
  int *pa, *pb;
  int **pc;

  pa = &a;
  pb = &b;

  *pa = 0;
  *pb = 0;

  swap(&pa, &pb);

  *pa = 1;
  *pb = 1;

  *pa = *pb;
  *pa = ((b + c)/a) + *pb;

  a = *pa + *pb;

  swap(&pb, &pc);

  **pc = *pb;

}
```


Figure 4.1: Example of program in C and location sets annotations.

```
void swap(int **p, int **q)
{
 int *tmp;
 int x, z, k;
 if (p > q) then
    {
     tmp = *p;
     *p = *q;
     *q = tmp;
 endif
}


main()
{
  int a, b, c, e;
  int *pa, *pb;
  int **pc;

  pa = &a;
  pb = &b;

  *pa = 0;
  *pb = 0;

  swap(&pa, &pb);

  *pa = 1;
  *pb = 1;

  *pa = *pb;
  *pa = ((b + c)/a) + *pb;

  a = *pa + *pb;

  swap(&pb, &pc);

  **pc = *pb;

}
```


Figure 4.2: Example of program in C and points-to graphs.

```
void swap(int **p, int **q)
{
 int *tmp;
 int x, z, k;
 if (p > q) then
     {
      tmp = *p;
      *p = *q;
      *q = tmp;
 endif
}


main()
{
  int a, b, c, e;
  int *pa, *pb;
  int **pc;

  pa = &a;
  pb = &b;

  *pa = 0;
  *pb = 0;

  swap(&pa, &pb);

  *pa = 1;
  *pb = 1;

  *pa = *pb;
  *pa = ((b + c)/a) + *pb;

  a = *pa + *pb;

  swap(&pb, &pc);

  **pc = *pb;

}
```


Figure 4.3: Example of program in C and an intermediate points-to graph.

```
Program ....
....
........
..............
Line Label 1 g(x, y, w)
................
..........
......
Line Label 2 g(a, b, c)
..............
........
.....
```

Figure 4.4: Example of program with different procedure call contexts.

currence of the unk target 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 [50, 34].

4.1.3 Profiling Environment

To monitor the actual run-time behaviour 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 [22] 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.

4.2 Experimental Results

4.2.1 Static Pointer Analysis Statistics

We start our study by measuring the amount of uncertainty resulting from the static pointer analysis. Table 4.2 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 4.2. 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 4.2).

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.

4.3 Profiling Results

4.3.1 Run-time Uncertainty

The first step in quantifying the run-time behaviour 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 4.2 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 4.2. Table 4.3

Application	*Uses (u) and Modifications (m) with N possible targets (including* unk *target, including* heap *target, number of source code lines)*			
	$N = 1$	$N = 2$	$N = 3$	$N > 3$
gzip	u: 277	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 43	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
vpr	u: 2488	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 428	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
mcf	u: 67	0 (0,0,0)	0 (0,0,0)	6 (0,0,3)
	m: 13	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
crafty	u: 4970	542 (534,67,59)	2 (2,2,1)	119 (0,26,24)
	m: 479	47 (45,11,9)	146 (146,66,13)	0 (0, 0, 0)
parser	u: 25178	241 (241,241,35)	36 (0,0,11)	7841 (181,230,259)
	m: 20	32 (32,32,6)	0 (0,0,0)	31 (9,4,9)
bzip2	u: 119	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 0	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
twolf	u: 3687	6 (6,0,6)	0 (0,0,0)	0 (0,0,0)
	m: 77	2 (2,0,2)	0 (0,0,0)	0

Table 4.2: Breakdown of static accesses according to the number of possible target memory locations. Results for the source code analysis for SPEC2000 programs. 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 de-references 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.

Application	*Uses (u) and Modifications (m) with N possible targets (including `unk` target, including `heap` target, number of source code lines)*			
	$N = 1$	$N = 2$	$N = 3$	$N > 3$
epic	u: 156	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 13	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
unepic	u: 59	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 6	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
mpeg2enc	u: 395	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 276	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
mpeg2dec	u: 499	8 (8,8,2)	0 (0,0,0)	6 (6,6,1)
	m: 75	0 (0,0,0)	0 (0,0,0)	10 (10,10,2)
g721-enc	u: 22	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 0	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
g721-dec	u: 6	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 2	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
gsmencode	u: 154	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
	m: 0	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)
gsmdecode	u: 346	0 (0,0,0)	0 (0,0,0)	9 (0,0,9)
	m: 31	0 (0,0,0)	0 (0,0,0)	0 (0,0,0)

Table 4.3: Breakdown of static accesses according to the number of possible target memory locations. Results for the source code analysis for MediaBench programs. 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).

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 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 4.4: Breakdown of static accesses with 2 or more possible target memory locations (Table 4.3) 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 4.3), 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 were 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 is more detail the reasons for the differences between static and dynamic results.

4.4 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 behaviours. Table 4.4 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.

Behavior difference			*Number*
Static	*Actual*	*Cause*	*of cases*
2 or more targets	Not executed	-	282
2 targets (inclusive *unk*)	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 4.5: Classification of dynamic accesses according to the difference with respect to the static behaviour and according to the cause for the difference

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.

4.5 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 behaviour 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 behaviour 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 behaviour, but by some other run-time behaviour. For instance, probabilistic pointer analysis [12] uses the frequency of path execution to estimate the probability of points-to relations. Nevertheless, little variability in the run-time points-to behaviour 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 behaviour of the points-to relations between executions with the ref and train input sets. A similar result was obtained in [40].

4.6 Summary

After analysing each case where we found discrepancies between the static results and run time ones, we conclude that the static analysis is quite good and only small improvement in the analysis methodology will make the static analysis even better and more precise.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The main goal of this thesis was to discover the causes of the failure of static points-to analysis and to quantify and characterize the actual behaviour of pointers in the programs. It also aimed to show how more accurate points-to information can be used to refine the knowledge of what data sets are used by what parts of the program.

To achieve the first target this thesis systematically quantified the amount of uncertainty due to may-alias points-to relations for two well-known classes of benchmarks. Unlike previous works [13, 15] that considered pointer analysis algorithms that trade-off reduced precision for increased scalability, in this thesis we were 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 behaviour.

5.2 Future work

We intend to continue this work implementing the possible optimisations that we referred to in the previous sections. We should use the SUIF framework, ie MACHSUIF creating new optimisation passes as we need.

5.2.1 Points-to optimisations

As showed in the discovery of the culprits of the breakdown between the static analysis result and the real behaviour of the pointers in the program, we have some improvement to be done in the static analysis, before work on points-to optimization passes themselves.

The information supplied by the data-flow and points-to analyses can be used to perform code optimisations. Conservative optimisations do not violate data-flow dependences, such as site independent constant-propagation, inter-procedural register allocation, and so on, making a more efficient machine code. But these conservative approaches may miss many possible optimisations that a more accurate data-flow analysis, especially points-to one, can expose. So a more aggressive approach to optimisations can be used. Then with speculative execution it is possible to execute parts of the code that are data-dependent, even if the correct results are not available. It allows for further optimisations that increase overall the performance of the program if the right speculative values are used, but if a wrong value is taken a roll-back mechanism must recover the result, wasting much time.

We must increase the precision of points-to analysis of aggregated objects making possible to distinguish the different fields that compose them. A better way to deal with pointer arithmetic must be develop to avoid miss interpretation of these operations. And finally, the array manipulation, ie. pointer arithmetic and index into array-like object must be improved. After tests that confirm the elimination of these causes of failure we can develop a points-to optimisation pass to be incorporated into MACHINE SUIF framework. It must use the data collected in the points-to step plus the generated PDOG. The points-to data will show the pointer variables with multiple possible addresses of uncertainty and their degree of uncertainty. The PDOG will expose sets of epochs that use those variables and repeat their instances along the pro-

gram analysis and execution. The optimisation pass will be developed to execute into multi-core machines. It will execute those epochs with uncertainty points-to variables in a speculative way. Each epoch will execute with one of the possible value. When the possible value is achieved the wrong ones will be dropped without prejudice of the normal execution due to the multiple-core execution.

Bibliography

[1] J. E. Sammet. Programming languages: History and future. *Communications of ACM, Vol. 15, No. 7*, pages 290–305, July 1972.

[2] A.V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1995.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2001.

[4] B. W. Kernighan and D. M. Ritcher. *The C Programming Language*, volume 1. Prentice Hall, second edition, 1988.

[5] 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.

[6] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.

[7] Dulong et al. An overview of the intel ia-64 compiler. *Intel Technology Journal*, November 1999.

[8] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Vol. 48, No 9*, September 1999.

[9] L. Hammond, M. Willey, and K. Olukotun. Data speculation for a chip multiprocessor. *In Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[10] P. Chen, M. Hung, Y. Hwang, R. D. Ju, and J. K Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *In Proc. of Symposium on Principles and Practice of Parallel Programming*, pages 24-36, June, 2003.

[11] R. D. Ju, J. Collard, and K. Oukbir. Probabilistic memory disambiguation and its application to data speculation. *Computer Architecture News, Vol. 27, No 1*, March 1999.

[12] Y. Hwang, P. Chen, J. K. Lee, and R. D. Ju. Probabilistic points-to analysis. *Lecture Notes in Computer Science, Vol. 2624, Languages and Compilers for Parallel Computing (LCPC 2001 Issue)*, pages 290–305, 2003.

[13] J. D. Silva and J. G. Steffan. A probabilistic pointer analysis for speculative optimizations. *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[14] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs, in proceedings of acm sigplan. *In Proceedings of Conference on Programming language design and implementation*, pages 77–90, June 1999.

[15] R. A. Chowdhury, P. Djeu, B. Cachoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. Technical Report TR-03-59 1, University of Texas at Austin, 2003.

[16] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. *In Proceedings of the Symposium on Principles of Programming Languages*, pages 19–21, January 1998.

[17] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Languages and Systems, Vol. 21, No 3*, pages 627–676, May 1999.

[18] F. Chow, S. Chan, R. Lo S.-M. Liu, and M. Streich. Effective representation of alias and indirect memory operations in ssa form. *Proc. of International Conference on Compiler Construction*, pages 253–257, April 1996.

[19] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. *In Proc. of Conf. on Programming Languages Design and Implementation*, pages 26–37, June 1998.

[20] T. Chen, W. Hsu J. Lin, and P. Yew. An empirical study on the granularity of pointer analysis in c programs. *In Workshop on Languages and Compilers for Parallel Computing, pages 151-160*, July 2002.

[21] The suif parallelizing compiler guide. Stanford Compiler Group, 1994.

[22] D. L. Moore. The suif programmer guide. The Portland Group, Inc, 1999.

[23] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. *In Proceedings of the conference on Programming language design and implementation*, pages 1–12, June 1995.

[24] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *In Proceedings of Conference on Program Language Design and Implementation*, pages 242–256, June 1994.

[25] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, and S.W. Liao and. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer, Vol. 29, No. 12*, pages 84–89, December 1996.

[26] M. Hind, M. Burke, P. Carini, and J. D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems, Vol 21, No. 4*, pages 848–894, July 1999.

[27] B. Steensgaard. Points-to analysis in almost linear time. *In Proceedings of Symposium on Principles of Programming Languages*, pages 32–41, June 1996.

[28] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. *In Proceedings of the Symposium on the Principles of Programming Languages*, pages 1–14, January, 1997.

[29] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. *In Proceedings of the Conference on Programming Language Design and Implementation*, pages 47–58, June 2001.

[30] P. Anderson, D Binkley, G. Rosay, and T. Teitelbaum. Flow insensitive points-to sets. *In Proceedings of 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 81–91, November 2001.

[31] M. Hind and A. Pioli. Which pointer analysis should i use? *In Proceeding of the ACM SIG-SOFT International Symposium on Software and Analysis (ISSTA '00)*, pages 84–89, Portland, Oregon, August, 2000.

[32] E. M. Nystrom, H.S. Kim, and W.M. Hwu. Global optimization by suppression of partial redundancies. *Intl. Static Analysis Symp*, pages 165–180, August 2004.

[33] 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.

[34] 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.

[35] 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.

[36] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. Technical Report UPC-DAC-2002-27 1, Computer Architecture Departament, Universitat Politcnica de Catalunya, Barcelona, 2002.

[37] G. Slavenburg A. S. Huang and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. *In Proceedings of the Int. Sump. on Computer Architecture*, pages 200–210, April 1994.

[38] T. Chen J. Lin, R. D.-C. Ju W.-C. Hsu, T.-F. Ngai, P.-C. Yew, and S. Chan. A compiler framework for speculative analysis and optimizations. *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 289–299, June 2003.

[39] T. Chen, X. Dai, J. Lin, W. Hsu, and P. Yew. Data dependence profiling for speculative optimizations. *In Proceeding of the International Conference on Compiler Construction*, 2004.

[40] 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.

[41] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative dataflow analysis, revisited. *Programming Language Design and Implementation - PLDI 2003*, San Diego, CA, November 2003.

[42] 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.

[43] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the precision of static reference analysis using profiling. Tech Report GIT-CC-02-05 1, College of Computing - Georgia Institute of Techonology, February 2002.

[44] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems, Vol. 1, No. 4*, December 1992.

[45] M. Sagiv, T. Reps, and R. Wilhelms. Parametric shape analysis via 3-valued logic. *Symp. on Principles of Programming Languages*, pages 105–118, January 1999.

[46] Standard performance evaluation corporation. www.spec.org/cpu2000.

[47] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Intl. Symp. On Microarchitecture*, pages 330–335, December 1997.

[48] R. Rugina and M. Rinard. Span: A pointer analysis pass. Technical report 1, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[49] R. Rugina and M. Rinard. Span: A shape and pointer analysis package. Technical report, m.i.t. lcstm-581, M.I.T., 1998.

[50] 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.