

Impact of Function Pointers on the Call Graph

G. Antoniol, F. Calzolari and P. Tonella

ITC-IRST

Istituto per la Ricerca Scientifica e Tecnologica

Povo (Trento), Italy I-38050

antoniol|calzolar|tonella@irst.itc.it

Abstract

Maintenance activities are made more difficult when pointers are heavily used in source code: the programmer needs to build a mental model of memory locations and of the way they are accessed by means of pointers, in order to comprehend the functionalities of the system.

Although several points-to analysis algorithms have been proposed in literature to provide information about memory locations referenced by pointers, there are no quantitative evaluations of the impact of pointers on the overall program understanding activities.

Program comprehension activities are usually supported by tools, providing suitable views of the source program. One of the most widely used code views is the Call Graph, a graph representing calls between functions in the given program. Unfortunately, when pointers, and especially function pointers, are heavily used in the code, the extracted call graph is highly inaccurate and thus of little usage, if a points-to analysis is not preliminarily performed.

In this paper we will address the problem of evaluating the impact of pointers analysis on the Call Graph. The results, obtained on a set of real world programs, provide a quantitative evaluation and show the key role of pointer analysis in Call Graph construction.

1. Introduction

Many widely used programming languages like C provide powerful constructs (e.g., structures, pointers and function pointers) to dynamically access memory locations. It is commonly accepted that these features improve language expressivity and allow different implementation choices giving greater language flexibility to developers.

The counterpart of these advantages is that program understanding and maintenance activities have to cope with these language features and thus are made much more difficult.

Since dynamic or static locations may be accessed through complex chains of dereferences, field selections and array subscripts, it becomes a difficult task to map them to the final referenced memory. Furthermore, function pointers may be employed to dynamically select the function to be called.

When such kinds of language features are used in the program, even the call graph, one of the basic program understanding code views could be highly inaccurate or even incorrect [10].

Several software engineering tools extract the Call Graph (CG), providing this useful program view in order to help maintenance involved people to increase their understanding of the code [10]. The CG should describe exactly which calls could be made from one program function to any other function in the program in any possible program execution. Unfortunately, since computing call relation has been proven to be undecidable, each tool computes some sort of approximation of the exact CG.

One of the main sources of approximation in the call graph determination is the possibility to call a function by means of a function pointer. In such a case the generic assumption that any function could be invoked by means of a function pointer can be refined only by performing a points-to analysis on the program. Such analysis will statically determine the set of functions pointed-to by every function pointer. Thus new edges are added to the call graph, eventually connecting some portions of the call graph that may result disconnected if the edges associated to calls through function pointers were ignored.

In literature several points-to algorithms [5][8][9] have been proposed in the framework of optimizing compilers and program parallelization and vectorization, to provide information about memory locations referenced by pointers. Recently the achieved knowledge about these algorithms has been applied in the field of software maintenance, investigating problems related to program slicing [7][14]. Different flow and context sensitive algorithms have been developed, each of them balancing the trade-

off between results accuracy and computational complexity [6][11][14]. The importance of controlling the time complexity has been investigated in [3], where a solution based on a fine grained context sensitivity specification has been proposed, while the effect of pointer variables and aliasing in symbolic execution is discussed in [4].

However there are no quantitative evaluations of the impact of pointers on the overall program understanding activities. Since the CG is one of the most useful code views, we decided to evaluate the impact of points-to analysis on the CG, by selecting a test suite of real world programs and assessing how CG building could be influenced by pointers.

We analyzed a test suite of 12 public domain programs, spanning from about 2500 lines of code (LOCs) up to about 52000 LOCs, and of a large industrial system consisting of 402 executables for about 4.7 million LOC (MLOC). Obtained results are similar for public domain and industrial code, and provide experimental support to the common feeling about the importance of points-to analysis.

This article is organized as follows: Section 2 will introduce the flow insensitive points-to algorithm presented by [1], supporting explanation with a small example that highlights the role of function pointers. Section 3 will provide experimental evaluation of the impact of function pointers on call graph construction for the selected test suite of about 5 MLOC. Finally, conclusions will be drawn in the last Section.

2. Call graph construction in presence of function pointers

When pointers are used inside a program under analysis it is important to determine the set of stack or heap allocated objects, data members and variables (i.e. *locations*) that each single pointer could access. Points-to analysis (PTA) is aimed at determining the set of locations a location may point to. Several algorithms have been proposed to perform PTA [1][5][8][9][12][13][14], which basically differ for the accuracy in reached results, obtained at the price of computational complexity. Since the analysis is statically performed, the results are computed with a conservative approach: for a given pointer, the set of identified locations approximates the real set of locations that the pointer could point to. In general, different algorithms identify a superset of the real set of locations that will be actually pointed by the given pointer. In this paper we will follow [1], the most accurate of the flow-insensitive algorithms.

PTA plays a central role in program understanding, and it is the starting point for further important analyses, as for example data dependence computation and call graph extraction.

2.1. The τ -types

In order to perform the PTA, we have to define the non standard set of types used in [1] and [13], and denoted as τ -types. τ -types are not related to program types: their function is to model how memory locations are accessed through pointers.

Since types may be recursive, type variables, usually denoted as τ_i , $i = 1, \dots, n$, are introduced to describe mutually referencing objects. Each τ -type is related to one single location, and thus a preliminary step assigns a type variable to every program's location as follows:

$$\forall x_i \in \text{Variables} \Rightarrow x_i : \tau_i$$

Every type variable has the attribute `ref` representing the set of type variables associated to the referenced locations. If a location contains actual data (i.e., it does not reference any other location), its type is defined as $\tau_i = \text{ref}(\emptyset)$. Otherwise, when it contains location addresses, it is indicated as: $\tau_i = \text{ref}(\Pi_i)$, being Π_i the set of τ_j referenced by τ_i , i.e. all the types of the locations possibly pointed by the location of type τ_i . For a given location x of type τ_i , the points-to set is defined as follows:

$$x: \tau_i = \text{ref}(\Pi_i) \\ \text{points-to}(x) = \{ y \mid y: \tau_j, \tau_j \in \Pi_i \}$$

The points-to relation between locations is conveniently represented by the *Storage Shape Graph* (SSG) [13], where each node is a τ -type (and is labelled with the location of that type), and a directed edge exists between τ_i and τ_j iff $\tau_i = \text{ref}(\{\dots, \tau_j, \dots\})$ [1].

Initially all locations are associated with different type variables, τ_i , each initialized as: $\forall i : \tau_i = \text{ref}(\emptyset)$. Then the program is analyzed¹, and whenever a statement involving pointers is encountered, referenced types are updated. The points-to algorithm infers a typing environment under which the program is *well-typed*, i.e., the SSG indicated by the τ -types is a conservative description of all possible dynamic storage configurations (well-typedness is defined in terms of typing constraints [13]).

2.2. The points-to algorithm

In this Section an algorithm is described [1] to compute the τ -types for the locations of a program, so that the resulting typing environment satisfies the well-typedness property. The notion of access path will be first introduced.

An *access path* is a unary expression described by the grammar of Figure 1. It contains an arbitrary sequence of dereferences and accesses to structure fields, that may be possibly combined through the \rightarrow operator. Since accesses to array components can be obtained even through normal

¹Since analysis is both flow and context insensitive, program's instructions could be analyzed in any arbitrary order.

$A ::=$	$(\text{type-id}) A$	$ * A$	$ A[\text{EXPR}]$
	(A)	$ \text{malloc}(\text{EXPR})$	$ A->\text{id}$
	$A.\text{id}$	$ \text{id}$	

Figure 1. Grammar of the access paths.

dereferences, they are treated as a particular case of dereference. Furthermore each field of a structure location has its own type variable. When an access path does not contain dereference operators ($*$, $->$ and $[\text{EXPR}]$), it represents a *fixed location* [11]. Otherwise it represents a *variable location*, that may be associated with different fixed locations at run-time, depending on the possible values of the involved pointers.

If a `malloc` library function call is encountered at line n , the dynamic allocation is represented as a fixed location object `heap_n`. While array components are fixed location objects denoted as `id[]`, a variable location leading to an array component has the form `id[EXPR]`.

$A ::=$	$(\text{type-id}) A_1 \{A.\text{type} = A_1.\text{type}\}$
	$* A_1 \quad \{A.\text{type} = \text{deref}(A_1.\text{type})\}$
	$A_1[\text{EXPR}] \quad \{A.\text{type} = \text{deref}(A_1.\text{type})\}$
	$A_1.\text{id} \quad \{A.\text{type} = A_1.\text{type}. \text{id}. \text{type}\}$
	$A_1->\text{id} \quad \{A.\text{type} = \text{deref}(A_1.\text{type}). \text{id}. \text{type}\}$
	$\text{id} \quad \{A.\text{type} = \text{id}. \text{type}\}$
	$(A_1) \quad \{A.\text{type} = A_1.\text{type}\}$
	$\text{malloc}(\text{EXPR}) \{A.\text{type} = \text{ref}(\text{heap}_n.\text{type})\}$

Figure 2. Syntax directed definitions to compute the type attribute of an access path. The type of a field of a type is denoted as: `type.id.type`.

The set of τ -types of an access path is represented as an attribute ($A.\text{type}$), which can be determined using the syntax directed definitions of Figure 2.

The `type` attribute of an identifier is a set containing only its τ -type, i.e., if $\text{id} = x$ and $x: \tau_i$, then $\text{id}. \text{type} = \{\tau_i\}$. Explicit and implicit pointer dereferencing and array element accesses perform a **deref** operation on the `type` attribute. The **deref** operation returns all the referenced types of a set of types, i.e., $\tau_i = \text{ref}(\{\tau_j, \dots\})$, $\text{deref}(\{\dots, \tau_i\})$ returns $\{\dots, \tau_j, \dots\}$.

(1)	$a_1 = a_2 \implies$ for each $\tau_1 \in a_1.\text{type}$, $\tau_2 \in a_2.\text{type}$
	$\tau_1.\text{ref} \leftarrow \tau_1.\text{ref} \cup \tau_2.\text{ref}$
(2)	$a_1 = \&a_2 \implies$ for each $\tau_1 \in a_1.\text{type}$, $\tau_2 \in a_2.\text{type}$
	$\tau_1.\text{ref} \leftarrow \tau_1.\text{ref} \cup \{\tau_2\}$

Figure 3. Basic actions performed for each pointer affecting statement.

The points-to algorithm examines in turn all the instruc-

tions of the program under analysis and when assignment instructions affecting pointers are encountered a τ -type update action is taken. Two basic kinds of instructions have to be considered during the analysis: assignments which take the address of a location, using the `&` operator, and assignments which do not take it. The corresponding update actions are defined in Figure 3.

An assignment between two access paths (instruction (1) in Figure 3) implies that the left hand side referenced types set is augmented with the types referenced by the right hand side. Instructions of the kind (2) add the right hand side type variable to the set of referenced types of the left hand side.

Interprocedurality is handled by reduction to the actions described in Figure 3. Since the analysis is context insensitive, only the points-to relations involving the actual parameters have to be considered. Each actual parameter² of a function invocation is the right hand side of the assignment, and the corresponding formal parameter is the left hand side (again by applying the rules in Figure 3).

The points-to analysis performs a fixpoint over the pointer affecting statements of the programs. These statements are repeatedly examined, until the application of the rules in Figure 3 does not change the `ref` sets any more. As a consequence, the worst case complexity of this algorithm is $\Theta(n^3)$, where n is a suitable measure of program size [1].

2.3. Function pointers

The result of the points-to analysis can be used to determine the set of functions that can be invoked by means of function pointers. In general, a call through a function pointer has the function name replaced by an access path. The set of fixed locations reachable through the access path gives the names of the invocable functions.

Figure 4 presents a small C program. Two functions compute the factorial of an integer: `fatt_rec` implements the recursive definition, while `fatt_ite` performs the iterative computation. Figure 5 presents the typing environment for this program, computed by applying the PTA presented in Section 2.2. The computed points-to relations are graphically presented by the SSG in Figure 6.

We can determine which functions could be invoked at the last statement of `main` by exploiting the SSG in Figure 6. By applying the syntax directed definitions of Figure 2 to the access path $(*a[10]->f)$ we obtain the result that functions `fatt_rec` and `fatt_ite` are possibly invoked through the field `f` of structure `s`. In fact, the array subscript operation on `a` gives location `a[]`. The indirect field selection is a dereference, leading to the node labelled `s`, followed by the field access, leading to `s.f`. The final dereference gives the locations pointed by the node labelled

²In order to avoid name conflicts, unique identifiers must be given to local and global variables, and to function formal parameters.

```

int fatt_rec(int n)
{
    if (n == 0)
        return 1;
    else return n * fatt_rec(n-1);
}
int fatt_ite(int n)
{
    int i, res;
    res = 1;
    for (i = 1; i <= n; i++) res = res * i;
    return res;
}

main(int argc, char *argv[])
{
    char c;
    int (*p)(int n); /* function pointer */
    struct A { int (*f)(int n);
                double x,y; } *a[100], s;

    p = &fatt_rec;
    a[10] = &s;
    a[10]->f = p;
    printf("Recursive <r>? Iterative <i>?");
    c = getchar();
    if (c == 'i')
        a[10]->f = &fatt_ite;
    printf("%d\n", (*a[10]->f)(3));
}

```

Figure 4. Last statement of main calls either fatt_rec or fatt_ite according to the points-to set of the access path ($*a[10]->f$).

a:	$\tau_1 = \text{ref}(\{\tau_2\})$
a[]:	$\tau_2 = \text{ref}(\{\tau_5\})$
p:	$\tau_3 = \text{ref}(\{\tau_4\})$
fatt_rec:	$\tau_4 = \text{ref}(\emptyset)$
s:	$\tau_5 = \text{ref}(\emptyset)$
s.f:	$\tau_6 = \text{ref}(\{\tau_4, \tau_7\})$
fatt_ite:	$\tau_7 = \text{ref}(\emptyset)$

Figure 5. Typing environment for the program in Figure 4.

s.f, i.e., functions fatt_rec and fatt_ite. Thus the call graph for the program in Figure 4, shown in Figure 7, must contain the two edges from the main to fatt_rec and fatt_ite. Notice that without pointer analysis functions fatt_rec and fatt_ite result never invoked, and the CG is completely unuseful.

3. Evaluation of the impact of function pointers on call graph

3.1. The CANTO tool

The points-to algorithm described in the previous section has been implemented and integrated in FLANT, the FLow ANalysis Tool which is part of CANTO (Code and Archi-

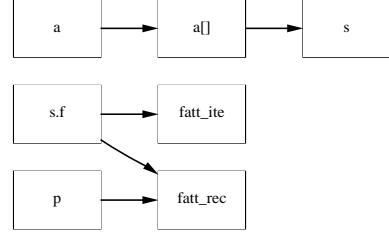


Figure 6. Storage Shape Graph for the program in Figure 4.

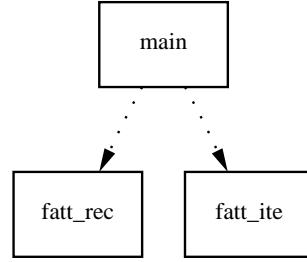


Figure 7. Call Graph for the program in Figure 4 (dotted arrows represent function calls by means of function pointers).

tecture aNalysis TOol), the program understanding and reverse engineering environment [2] developed at IRST.

PTA results are typically displayed to the user in the form of the SSG. They are also useful for other successive analyses: the points-to information can be integrated for data dependences computation or call graph construction. This is done by a module which transforms a variable location into the set of fixed locations reachable through it. At this point the call graph can be built.

3.2. The Test Suite

The impact of function pointers on the call graph was evaluated on 12 public domain programs, ranging from 2.5 to 51.4 kLOC, and on a large industrial system consisting of 402 executables with a total size of 4.7 MLOC. In both test suites the structure of the call graph is deeply affected by the invocations performed through function pointers.

Table 1 reports some features of the public domain programs used to test the impact of function pointers in call graph extraction. Their size ranged from 2525 LOC to 51460 LOC (1896 to 38130 uncommented LOC, ULOC), while the number of procedures was between 50 and 724. The total number of nodes in the AST of each program is also given. The test suite contains real public domain applications, many of which are common system utilities. They are good representative of the expressive capabilities of the

Program	LOC	ULOC	Nodes	Procs
shorten (1.23)	2525	1896	3012	50
gdbm (1.7.3)	7025	3665	2369	54
gzip (1.2.4)	8163	5215	5358	98
minicom (1.71)	12239	9379	11395	180
grep (2.0)	12935	8154	11204	130
sed (2.05)	15656	11098	9818	164
find (4.1)	16474	10684	12081	201
less (290)	18449	10970	11008	302
flex (2.5.2)	19231	13557	14909	189
gawk (2.15.6)	21654	15154	17484	205
tar (1.11.8)	42667	27855	15566	277
tcsh (6.04)	51460	38130	36363	724

Table 1. Some size measures of the public domain programs in the test suite: lines of code (LOC), un-commented LOC (ULOC), AST nodes, procedures.

C language, like, e.g., pointers and pointers to functions, structures, recursion and dynamic allocation.

Table 2 shows the total number of function calls and calls through function pointers in the original programs (both as absolute numbers and as percentages of the total calls). It should be noticed that while some programs use function pointers very seldom (e.g. `shorten`, `flex` and `gdbm`), others make use of function pointers for up to 9.1% of the total number of calls, as for example the `find` or `gawk` programs.

Program	Tot. calls	Func. ptr calls
shorten	129	0 (0.0 %)
gdbm	156	1 (0.6 %)
gzip	327	4 (1.2 %)
minicom	1271	6 (0.4 %)
grep	317	11 (3.4 %)
sed	516	3 (0.5 %)
find	229	21 (9.1 %)
less	906	4 (0.4 %)
flex	1052	0 (0.0 %)
gawk	808	24 (2.9 %)
tar	883	8 (0.9 %)
tcsh	2777	9 (0.3 %)

Table 2. Total number of function calls and calls through function pointers (test suite of public domain programs).

Program	Tot. calls	Invocable func.	Avg.
shorten	129	0 (0.0%)	-
gdbm	156	1 (0.6%)	1.0
gzip	339	16 (4.7%)	4.0
minicom	1273	8 (0.6%)	1.3
grep	322	16 (4.9%)	1.4
sed	520	7 (1.3%)	2.3
find	758	550 (72.5%)	26.1
less	940	38 (4.0%)	9.5
flex	1052	0 (0.0%)	-
gawk	921	137 (14.8%)	5.7
tar	892	17 (1.9%)	2.1
tcsh	2975	207 (6.9%)	23.0

Table 3. Total number of function calls and function pointer invocations after points-to analysis integration for function pointers resolution. Last column gives the average number of functions invocable at each call made via function pointers (test suite of public domain programs).

Table 3 reports the total number of function calls after function pointers resolution, and the invocable functions for function pointers calls. For example, in the case of the `tcsh` program, even if the calls through function pointers are only 0.3% of the total calls, the invocable functions through those calls are 6.9% of the total number of calls. The most remarkable datum regards the `find` program, for which 72.5% of the calls is achieved through function pointers. On average, 9.3% of the total calls is made through function pointers for the program test suite. Last column of the table gives the average number of functions invocable from each call made by means of function pointers³. It ranges from 1 (`gdbm`) to 26.1 (`find`) and is on average 7.6.

The number of functions invocable through function pointers by itself is not enough to understand the real impact of their use in programs. In fact, even if the number of functions invocable through function pointers is not very large (9.3% on average), nevertheless the number of functions that are disconnected in the call graph when such calls are ignored may be very large. If function pointers are not resolved, some links in the call graph may be missed, due to the fact that such links are actually originated by calls through function pointers. In some cases, when the calls to the functions in the subgraphs are issued only through function pointers, if pointer analysis is not performed to extract the call graph, entire subgraphs of the call graph may wrongly result as disconnected. In these cases, functions called only through function pointers will result unreach-

³Notice that `shorten` and `flex` do not use function pointers.

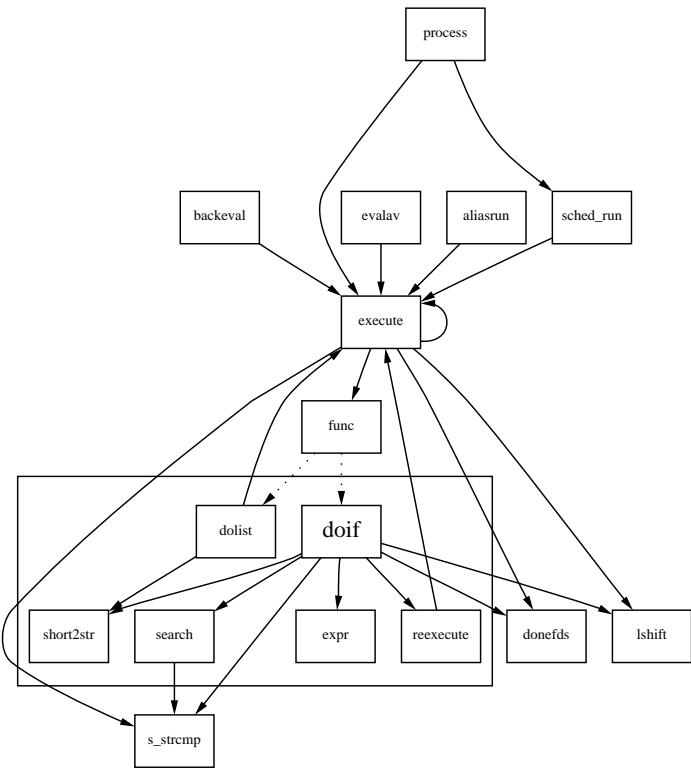


Figure 8. Portion of the call graph of `tcsh` centered on function `doif`. Dotted edges represent calls through function pointers and the transparent box contains the otherwise disconnected subgraph.

able from the `main`, figuring incorrectly as dead code.

For example, Figure 8 shows a portion of the call graph of `tcsh` centered on function `doif`, with calls through function pointers resolved (dotted edges represent function calls by means of function pointers). Notice that six of the functions in this subgraph would be unreachable without resolving function pointers: the corresponding subgraph is bounded by the transparent box in Figure 8.

Table 4 and Figure 9 give the number of functions reachable from the `main` function before and after having resolved function pointer calls. The results show that resolving function pointers gives a substantial improvement in the number of functions reachable from the `main`: data are most impressive for the `gzip`, `grep` and `find` programs, for which the number of functions reachable after function pointer resolution ranges from about twice to eight times higher than before. In these cases, the call graph extracted without function pointer resolution results highly inaccurate and of little usage both for direct program understand-

Program	No func. ptr.	Resolved f. ptr.
shorten	40 (80.0%)	40 (80.0%)
gdbm	36 (66.6%)	36 (66.6%)
gzip	37 (37.7%)	96 (97.9%)
minicom	155 (86.1%)	170 (94.4%)
grep	18 (13.8%)	115 (88.4%)
sed	124 (75.6%)	130 (79.2%)
find	21 (10.4%)	175 (87.0%)
less	266 (88.0%)	282 (93.3%)
flex	175 (92.5%)	175 (92.5%)
gawk	141 (68.7%)	163 (79.5%)
tar	186 (67.1%)	203 (73.2%)
tcsh	386 (53.3%)	690 (95.3%)

Table 4. Functions reachable from `main` before and after function pointers resolution.

ing and for further call graph based analyses. On average, on the test suite, before function pointer resolution 61.6% of the program functions was reachable, while 85.6% resulted reachable after the analysis, corresponding to an average improvement in the reachable functions of 24%.

Even when function pointer calls are resolved, a certain number of functions still remains unreachable. This can be explained partly by the fact that this is actually dead code, belonging to general purpose modules, of which only a subset of the exported functions are actually used, and partly because of the use of conditional compilation (e.g. operating system dependent code). For example, for the `find` program the functions reachable from `main` after the analysis are 87% of the total program functions: the non reachable functions are mostly contained in the `regexp.c` file, which is a general module linked to the program, exporting functions to parse regular expressions.

An industrial system was analyzed with the purpose of extracting accurate call graphs to support program comprehension and of determining possible dead code. The analyzed application is a large (4.7 MLOC) software system in the banking domain written in C language and consisting of 402 distinct executables⁴. Data on this system are shown at the top of Table 5.

In the middle of Table 5 the function pointer calls in this system are considered. Only 0.1% of the calls employs function pointers, and the resulting invocable functions are only the 1.1% of all invocations. On average each function pointer call is resolved by 1.6 invocable functions. Nevertheless, there is a very impressive transitive effect of function pointer use in the call graph structure and in its discon-

⁴As usually in literature we consider a `main` function and all the linked modules as an executable.

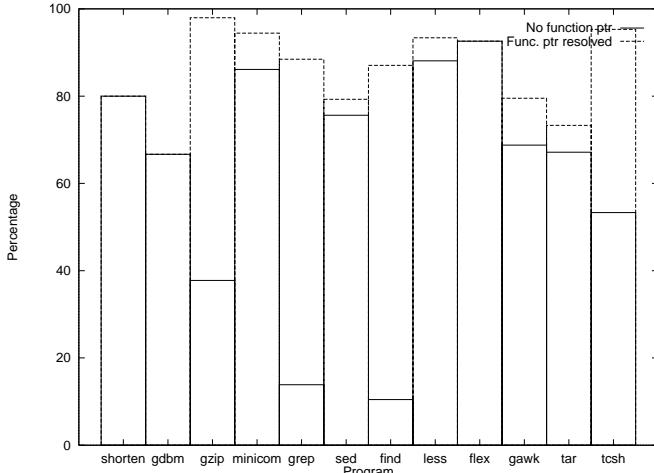


Figure 9. Improvement in the number of functions reachable from the `main` in the call graph when calls through function pointers are resolved.

nected portions.

The bottom of Table 5 shows the average number of functions reachable from the `main` before and after function pointer resolution. The average is computed over all the 402 executables. If calls through function pointers are ignored, on average 65% of the functions in a program is reachable from the `main`. This percentage increases to 83% of the functions after adding call edges determined through points-to analysis. Therefore a relevant portion of the call graph is disconnected from the `main` when function pointers are not considered, and the extracted call graphs are quite imprecise. Function pointer invocations are the only way to reach some of the functions in the analyzed programs. It should be noticed that the percentage of functions that on average are not reachable from the `main` (17%) is not the percentage of dead functions. In fact only functions unreachable from *all* the `main` functions have to be considered dead.

Figure 10 shows a portion of the call graph of an executable from the industrial application. Function names are replaced by dummy names for confidentiality reasons. The graph is focused on the `FUNC1` function, containing a call through function pointers. Invocations resulting from function pointer resolution are depicted with dotted lines. 8 functions are in the set of possibly pointed to functions for the function pointer used by `FUNC1`. It is evident from Figure 10 that by ignoring the dotted calls, not only the 8 invoked functions become disconnected, but also the 6 functions in the lower layer.

The bottom of Table 5 also shows the number of dead functions (functions unreachable from the `main` for all executables). Dead functions are divided into three categories:

Main modules	402
Library modules	815
Total modules	1217
Total functions	4695
Average funcs per exe	168
LOC	4.7 M
Calls before func. ptr resolution	
Calls	300993
Func. ptr calls	1995 (0.1%)
Calls after func. ptr resolution	
Calls	302237
Func. ptr calls	3239 (1.1%)
Average calls per func. prt.	1.6
Reachable functions per exe	
No func. ptr.	109 (65%)
Resolved func. ptr.	139 (83%)
Dead functions	
Static funcs (called by statics)	57
Static funcs (called by globals)	2
Global functions	18
Total dead functions	77

Table 5. The upper part of this table gives some size measures of the industrial system. In the middle, the numbers of function calls before and after pointer analysis is given. At the bottom functions reachable from the `main` and dead functions are given.

static functions that are called only by other static dead functions, static functions that are called also by global dead functions and global dead functions. The reason for considering such three groups is that the code provided by the user could be incomplete. If a function whose code was not provided invokes one of the dead functions, the performed analysis cannot detect the call, because the invoking function is considered a library function and the analysis cannot enter its body. While this is possible for a global function, declared `extern` in the library module, this case cannot occur for the static functions, that are accessible only from inside their module. Therefore the 57 static dead functions invoked only by statics are surely dead. On the contrary the 2 static functions invoked by some dead global function and the 18 global dead functions may be called by a library module, whose code was not provided, declaring them as `extern` functions. By looking at the functions resulting library functions from the analysis, some potentially anomalous situations emerged. In fact 7 library functions were not part of any standard UNIX library or of any special user library. 5 of them were recognized as part of a user module

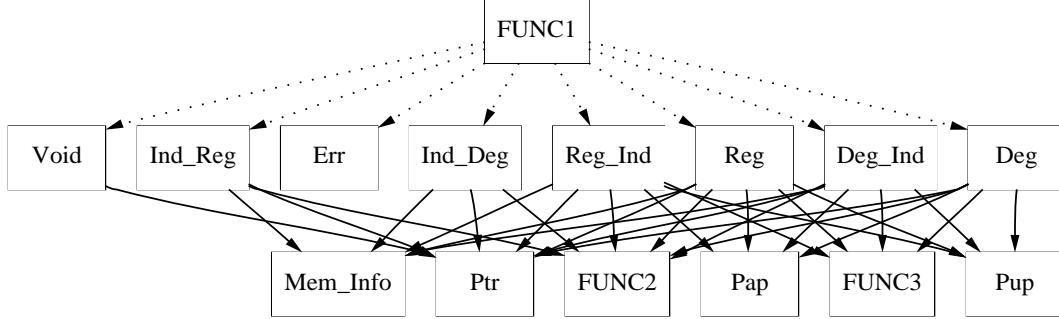


Figure 10. Portion of the call graph of an executable from the industrial application. Calls through function pointers are depicted with dotted lines.

which encapsulates a specific functionality and therefore is assured not to declare as `extern`, nor to invoke any global function from other modules. The other 2 functions were actually part of 2 modules whose code was not provided for the analysis. Therefore the users of the presented results were warned that while the 57 static functions invoked only by statics are surely dead, the other 20 functions could be invoked from inside the 2 missing modules.

4. Conclusions

In this paper we addressed the problem of evaluating the impact of pointers on the call graph, one of the most widely used views of the code that can help software engineers and maintenance involved people increase their understanding of a program.

Dynamic reference to memory locations augments programming languages expressivity and flexibility at the price of making program understanding activities more difficult when pointers related features are heavily used in source code.

Although the call graph can be extracted by several software engineering tools, there are no quantitative evaluations of the impact of pointers on the call graph. The obtained results on a set of 12 public domain programs show the key role of pointer analysis: on average about 24% of functions are reachable from the `main` only by traversing function pointers invocations and are therefore missed if the points-to analysis is not performed. A similar result was obtained for a large industrial application, in which function pointers account for about 18% of functions otherwise unreachable from the `main`. Moreover, for some of the analyzed programs the number of functions reachable after function pointer resolution ranges from about twice to eight times the number of functions reachable before points-to analysis.

References

- [1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Phd Thesis, DIKU, University of Copenhagen, 1994.
- [2] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, and S. Zanfei. Program understanding and maintenance with the CANTO environment. In *Proceedings of the International Conference on Software Maintenance*, pages 72–81, Bari, Italy, Oct 1997.
- [3] D. Atkinson and W. Griswold. The design of whole-program analysis tools. *Proc. of the Int. Conf. on Software Engineering*, pages 16–27, 1996.
- [4] A. Cimitile, A. D. Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 124–133, Opio(Nice), 1995.
- [5] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proc. of the ACM SIGPLAN'94 Conf. on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [6] A. M. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *International Conference on Computer Languages*, 1994.
- [7] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: A system for development of program analysis based tools. In *Proc. of the 33rd Annual ACM Southeast Conference*, pages 110–119, March 1995.
- [8] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, 1989.
- [9] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *Proc. of the ACM SIGPLAN'92 Conf. on Programming Language Design and Implementation*, pages 235–248, 1992.
- [10] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *International Conference on Software Engineering*, pages 90–99, Berlin, Germany, March 1996.

- [11] H. Pande, W. Landi, and B. Ryder. Interprocedural def-use associations for c systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5), May 1994.
- [12] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. *Proc. of the Int. Conf. on Compiler Construction*, April 1996.
- [13] B. Steensgaard. Points-to analysis in almost linear time. *Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.
- [14] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Points-to analysis for program understanding. *Proc. of the International Workshop on Program Comprehension*, 1997.