Examining The Prevalence and The Historical Trends of Indirect Function Calls in Open Source Systems: A Case Study, gcc 2001-2011

Saleh M. Alnaeli
Computer Science Department
University of Wisconsin-Colleges
Menasha, Wisconsin 54952
saleh.alnaeli@uwc.edu

Melissa Sarnowski
Computer Science Department
University of Wisconsin-Fox Valley
Menasha, Wisconsin 54952
SARNM6825@students.uwc.edu

Abstract

An empirical study that examines the prevalence and distribution of indirect function calls using function pointers and virtual methods in general-purpose software systems is presented. The study is conducted on a well-known, large-scale software system, gcc.4.5.3, comprising over four million lines of code. The system is analyzed and the number of function pointer and virtual method calls is determined. Additionally, function pointers are categorized based on their type and the complexity they pose when conducting inter-procedural static analysis. The results show that more often than not, function pointers are used in situations that make analysis very difficult (i.e., NP-hard). Thus, conducting accurate program analysis (e.g., program slicing, call graph generation) becomes very costly, or impractical, to conduct. Analysis of the historical data over a ten-year period of gcc shows that there is an increase in the usage of both calls using function pointers and virtual methods over its lifetime, thus posing further problems for inter-procedural analysis.

1 Introduction

One of the patterns used by software developers when writing in C/C++ that is known to pose huge challenges for software engineers to statically analyze programs is indirect function calls via function pointers [1, 2] or virtual methods [3]. A single function pointer can alias any function with the same signature, and a virtual method can alias methods with the same signature in derived classes. So, precisely knowing which specific function is invoked can only be done at run time. When software engineers conduct static analysis, an approximated, but still valid, approach is used to resolve all function pointers/virtual methods in the system and then conservatively assume that all potential functions/methods are invoked in the studied system. Consequently, more complexity and inaccuracy is imposed to static analysis undertaken. In many studies, the problem is shown to be NP-hard based on the ways function pointers are declared and manipulated in a system [1, 2]. For accurate inter-procedural analysis, function-alias analysis is a very important step that should be properly handled [4].

For example, in the context of automatic parallelization, a for-loop that contains a function call with a side effect is considered un-parallelizable i.e., cannot be parallelized using OpenMP. In order to safely analyze for parallelism, the set of possible targets of a function-pointer call must be determined. If a side effect exists for one possible target, parallelization may not be safe, resulting in a conservative approach to inhibit parallelization [3, 5]. Therefore, a conservative, yet safe approach is to assume that all calls using function pointers and virtual methods carry side effects. However, such an approach might not be practical in other software-engineering problems where the accuracy is a crucial concern (e.g., slicing, transformation.) and may pose a negative impact.

Additionally, programs written in object-oriented languages, such as C++ or Java, may have challenges in inter-procedural analysis as well. That is, static analysis is even more difficult because of object inheritance and function overloading through methods [2]. Virtual methods in an object oriented programming language can be overridden by derived classes [6]. For adequate analysis, virtual method calls need to be resolved, and literature is rich with algorithms for this purpose [3, 7-9]. In the case of safe analysis for parallelization, the set of possible targets of a virtual method call must be determined, and any calls made transitively must be included in the analysis for reliable results.

With the increased use of open source model in programming, we become motivated to conduct a deeper study that empirically examines the prevalence and distribution of indirect calls conducted via function pointers and virtual methods in general-purpose software systems, which can help software engineers properly estimate the complexity and challenges expected when static analysis is proposed. This can also play an important role in monitoring how systems get more complex as they evolve over time.

To the best of our knowledge, no historical study has been conducted on the evolution of function pointers and virtual methods in open-source systems. We believe that an extensive comprehension of the nature of function pointers and virtual usage is needed

for a better understanding of the problem and obstacles that must be considered when static analysis is conducted.

In this study, we empirically examine one of the well-known large-scale open source software systems in both academia and industry, namely gcc. The history of the gcc system is examined based on multiple metrics. The number of function pointers, virtual methods, and indirect calls is determined for each release throughout a 10-year period (2001-2011). Then, the classification of function pointer types is determined (Global, local, Formal parameter, Class member, and array of function pointers). This data is presented and analyzed to uncover trends and general observations in gcc as a case study. The analysis shows if there is a trend toward increasing or decreasing growth on the number of function pointer types and virtual methods in gcc.

The paper is organized as follows. Background and related work are presented in section 2. Section 3 presents function pointer types and is followed by section 4, which talks about virtual methods and their usage. The approach used in this study for detecting function pointers and virtual methods is introduced in section 5. A case study we designed and conducted, along with results and findings, is presented in section 6. Examination of the change in the presence of calls conducted via both function pointers and virtual methods over a 10-year period is presented in section 7. A discussion about our findings and limitations of this study and the work is finally concluded in section 8.

2 Related Work

There are many algorithms used for static analysis in the presence of function pointers and virtual methods. Our concern in this study is the usage of function pointers and virtual methods, in particular for open source software systems, and how they evolve over time for better understanding and uncovering any trends or evolutionary patterns. That is, we believe those patterns and trend can be valuable information in determining and predicting solutions and effort required to better statically analyze those systems written in C/C++ languages. As a case study, gcc is used in this investigation.

The bulk of previous research on this topic has focused on detecting and resolving function pointers and virtual methods in inter-procedural analysis. Most of these works concentrate on problems such as the construction of call graphs, particularly in the context of static program analysis [3, 7-9]. However, no study has been conducted on the evolution of open source systems over time in the inter-procedural analysis context in the presence of function pointers and virtual methods.

Ben-Chung et al. [2] conducted an empirical study of function pointers in the complete SPECint92 and SPECint95 benchmarks. They evaluate the resolution of function pointers and the potential program transformations enabled by a complete call graph. They have shown samples of function-pointer usage in the benchmark they studied, as an attempt to explore the issues that might be critical in the design of a complete inter-procedural pointer-analysis algorithm. They have observed that the call graph construction problem has become an inter-procedural pointer-analysis problem as all pointers need to be analyzed and resolved for correct results.

Ryder et al [1] examined multiple systems from different domains by statically gathering empirical information on C function-pointer usage as an attempt to better predict appropriate inter-procedural analyses required for C programs. They have classified and categorized the programs based on ease of call multi-graph construction with the presence of function pointers. They observed that calls to globally-declared function-pointer variables far outnumber the calls to any other kind of function pointer. This agrees with our observations for most of the systems we studied. However, the study was done on relatively small to medium scale C systems and virtual methods were not considered.

Here, we empirically examine a large scale open source system to determine what roadblocks exist for development of automated tools for better static and inter-procedural analysis. Additionally, we show how *gcc* system evolves overtime in terms of expected difficulty of analysis posed by function pointer/virtual method usage.

3 Function Pointers

Many studies have agreed that the ways function pointers are declared and used in software systems can play a big role in the degree of complexity of inter-procedural and static analysis [1]. For instance, the broader the scope of a function pointer (e.g., global in the worst case) within the system, the higher the complexity is (NP-hard in many cases) in conducting static analysis. In this section, the different types of function pointers are discussed. The different types of function pointers that cause in NP-hard analysis are presented in the following sections [10, 11] with the focus being on these types of function pointers.

3.1 Global Function Pointers

Global function pointers include all function pointers declared at the file scope of a program, i.e., outside any function body. This is shown in Figure 1. This includes external function pointers, e.g., the function line 1 in Figure 1. Global function pointers are one of the types that, if used, cause in NP-hard analysis [1]. In this study, the number and percentage of the function calls via global function pointers is counted. Additionally, a historical study is conducted in the studied system in order to discover how it evolves over time in terms of this type usage and distribution.

3.2 Array of Function Pointers

Arrays of function pointers are usually used in general purpose software systems, especially the systems developed in the language C. It is known to be amongst function pointer types that cause in NP-hard analysis as well [1]. An example is shown in Figure 1 line 14. The number and percentage of the function calls via arrays of function pointers is determined and counted. The study shows the evolution of function pointer array usage in gcc, which we targeted in this study as a case study.

3.3 Function Pointer Structure Fields

The number of all function calls conducted indirectly using function pointers that are structure fields is determined and counted. Additionally, distribution of indirect calls is

release	Language	KLOC	Files	Function Pointer	Virtual Method
gcc4.5.3	C/C++	4,029	40,638	10,653	32,975

Table 1. gcc System used in the study.

```
1: extern "C" int (*fpEXT1)(int&,int);
2: int (*fpEXT2)(int&,int);
-------------------------------------
3: typedef int (*FUNC) (int &, int);
4: FUNC fp;
-------------------------------------
5: class ClassFPtr {
6: public:
7:    typedef int (A::*_fVar)();
8:    fVar fvar;
9:    _fVar fvar2;
10:   void setFvar(_fVar afvar) {
11:                    fvar = afvar; }
12:};
13: ClassFPtr ObjFPtr;
-------------------------------------
14: int (*fp1[2])(int&,int);
-------------------------------------
15: struct srct{
16:    void (*fptrS)();
17:    int (*fptrArray[12])();
18:};

Types of function pointers detected:
1:  fpEXT1 external function pointer
2:  fpEXT2 Global function pointer
3:  FUNC typedef-ed function pointer
4:  fp function pointer of type FUNC in 3
7:  _fVar class member typedefed
8:  fVar  Of type _fVar
9:  fVar2 Of type _fVar
10: afVar - formal parameter of type _fVar
13: objFPtr - instance of class with Fptr
14: fp1  Array of function pointer
15: fptrS structure member
16: fptrArray array of function
17: pointer in structure
```

Figure 1. Examples of function pointers that cause analysis to be NP-Hard, and are detected by the tool *VirFptrStat*

studied which includes the calls done through function pointers that are structure fields. Examples of this are shown in Figure 1 in line 16 as a field in the local structure srct. It is

also known to be amongst function pointer types cause in NP-hard analysis[1] and it is commonly used with C/C++ code.

3.4 Function Pointer Class Members

A function pointer can be a member of a class. Classes that contain at least one function pointer as a member are all detected and counted. Additionally, all indirect function calls conducted with function pointer class members are counted and compared to other types. They are somewhat similar to the structure members in that they can be used to create objects that can be used to invoke a function indirectly. Function pointer class members are also known to cause in NP-hard analysis and thus they are consider in this study as well [1, 12]. Lines 7, 6 in Figure 1 are examples of this type.

3.5 Function Pointers Formal parameters

Function pointers can be used as formal parameters for both functions and methods and are commonly used in C/C++ software systems. They are used to pass functions as arguments to functions and methods. This type is also considered as one of the function pointers that cause in NP-hard analysis, so we considered this type in our study. The number of calls that are conducted by formal parameters is counted and its usage percentage is determined as well.

4 Virtual Methods

Calls to virtual methods increase the complexity of conducting static analysis in a similar manner as calls to function pointers [3, 7, 13, 14]. The use of virtual methods is basically a constrained use of function pointers. While static analysis in the presence of virtual methods is somewhat simpler than calls via function pointers, it still poses a difficult problem and produces a large increase in complexity (NP-hard) [3, 7, 13]. In some cases, the analysis may not even be possible since the target of a call is unknown at compile time.

Figure 2 presents virtual method examples as declared in a class. In this study, we count and determine the number of virtual method declarations and all function calls that are carried out using virtual methods. The growth of virtual method usage over time is also presented in Section 7.

5 Detection of Function Pointers and Virtual Methods

We now describe the methodology used to detect the usages of function pointers and virtual methods and to collect the data for our case study.

In this study, function pointers are divided into two categories. The first category type is function pointers that are known to cause in NP-hard analysis [1] (e.g. Global function pointers). Such a function pointer could be one of the following types: global function pointers, array of function pointer, function pointer class member, function pointers as formal parameters, or structure member. Virtual methods are all considered in this study as they are known to cause in NP-hard analysis [7, 13, 14].

Virtual methods are identified using their declarative directive virtual. Our tool does not miss any virtual methods, resulting in both complete precision and recall.

```
1: class Base {
   protected:
     string ob_Name;
     Base(string strName): ob_Name(strName){}
   public:
     string GetName() { return ob_Name; }
      virtual const string action(){ return "Default"; }
   };

2: class Obj1: public Base {
   public:
     Obj1(string strName): Base(strName){}
     virtual const string action()
              {return "predefined name";}
   };
3: class Obj2: public Base {
   public:
     Obj2(string strName):  Base(strName) {}
     virtual const string action() {
       string userEntery;
        cout<< " enter a name > ";
        cin>>   userEntery;
        ob_Name =userEntery;
        return "user entered"; }
  };
void Report(Base &rAnimal) {
   cout << rAnimal.GetName() <<
        " is "<< rAnimal.action() << endl;
}
4: int main()
{ Obj1 iObj1("nameObj1");
  Obj2   iObj2("");
 Base* rAnimal;  int sel;
 cout<<" enter 1 or 2 ";  cin>>sel;
 if (sel==1)
    rAnimal = &iObj1;
 else
    rAnimal = &iObj2;
 for(int i =0; i<10; i++)
     Report(*rAnimal);
} // end of method main
```

Figure 2. Virtual Method examples as declared in classes and its inherited classes detected by VirFptrStat

We developed VirFptrStat, a tool to statically gather empirical information about function pointer and virtual method usage. When referring to VirFptrStat, we consider the usage

in regards to function calls that are conducted via function pointers and virtual methods. That is, VirFptrStat statically extracts function pointers and virtual methods by both declaration and most importantly calls. All source code files are analyzed to determine if virtual methods or function pointers (declarations or calls) exist. If these do exist, then we count the function pointers that fall into the category of types that cause in NP-hard analysis. First, we collect all files with C/C++ source-code extensions (i.e., c, cc, cpp, cxx, h, and hpp). Then, we use the srcML (www.srcML.org) toolkit [15] to parse and analyze each file.

VirFptrStat, analyzes the srcML to search the parse tree information using lxml from the lxml toolkit supported by .NET framework. In Python, we use the lxml.etree, a XML toolkit which is a Pythonic binding for the C libraries, libxml2 and libxslt. It combines the speed and XML feature completeness of these libraries with the simplicity of a native Python API, mostly compatible with ElementTree API [Behnel 2014]. That is, it has a similar implementation to SAX, where huge files can be easily read in chunks for better implementation with a limited number of resources. The tool is used to identify every function pointer and virtual method in a system. If a function pointer is determined to be among the types that cause in NP-hard analysis, it is recorded, otherwise it is distinguished from the hard type's number and recorded as another category.

Once in the srcML format, VirFptrStat iteratively finds each virtual method and function pointer and then analyzes the function pointers to find the different types. A count of each function pointer or virtual method per class is also recorded. It also records the number of hard function pointers found. The final output is a report of the number of hard function pointers and virtual methods and their distribution in gcc. Our tool, VirFptrStat, detects all types of function pointers whenever they are present in the code. Figure 1 contains examples of the detection of these types of function pointers. Pointers to member functions declared inside C++ classes are detected as well. Locally declared function pointers (as long as they are not class members, in structures, formal parameters, or an array of function pointers) that are defined in blocks or within function bodies are considered as simple or typically resolved pointers.

6 Findings and results

We now study the presence, usage, and distribution of function pointers types and virtual methods of gcc.4.5.3, which is a large-scale, open-source software project. Usage means the calls conducted via function pointers and virtual methods. Table 1 presents some information about gcc, including the version, number of files, and LOCs. Note that since gcc is developed in C/C++/Java languages, it does use function pointers in addition to virtual methods. Some information about the distribution of all function pointer types that are statically declared in gcc4.5.3, as well as the number of virtual methods, is presented in Figure 3 and Table 1 respectively.

Our study focuses on three aspects regarding function pointers and virtual methods. The **first** aspect we focus on is the usage of function pointers and virtual methods along with their distribution in gcc. By usage we mean that all function calls that were done via function pointers. This distribution can be used to evaluate the expected complexity of

the studied system, gcc, when static analysis is conducted. That is, if gcc uses the function pointer types that cause in NP-hard analysis more frequently than other types that do not cause in NP-hard analysis, then the indication is that the system is potentially difficult to statically analyze using a compiler or other automated tools. **Second**, we examine which function-pointer type is the most prevalent amongst types that cause in NP-hard analysis in gcc4.5.3. **Finally**, we examine how the presence of function pointers and virtual methods change over the lifetime of a software system. We propose the following research questions as a more formal definition of the study.

R1: *What is a typical number and percentage of function calls that are conducted via virtual methods and function pointer types that cause in NP-hard analysis in open source system such as gcc.4.5.3?*
R2: *Which function pointer types amongst those that cause in NP-hard analysis are the most prevalent?*
R3: *Over the history of gcc system, is the presence of virtual methods and function pointers that cause in NP-hard analysis increasing or decreasing?*

Question R3 concerns the growth of virtual methods and function pointers that cause in NP-hard analysis as we believe that they pose difficulties in any inter-procedural or static analysis performed on a large-scale software system. We now examine our findings within the context of these research questions.

6.1 Number and Percentage of Calls

The results collected for gcc4.5.3 are presented in Table 2. We give the number of function pointer calls along with the number of virtual method calls. Figure 4 shows the percentage of function pointer calls that cause in NP-hard analysis computed over the total number of detected function calls.

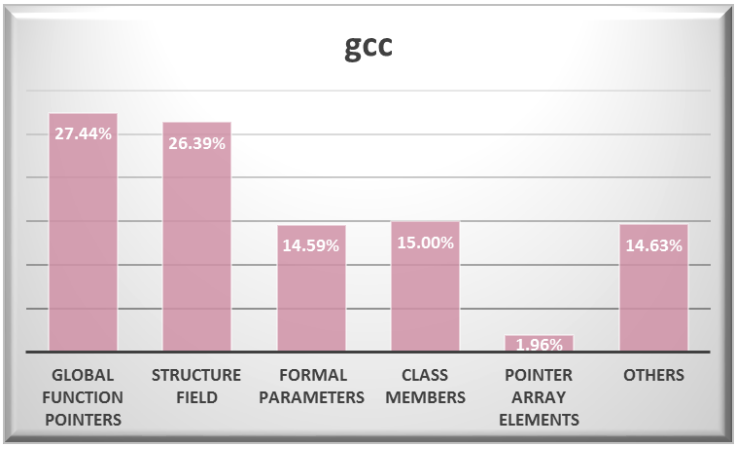

Figure 3 .Percentage distribution of function pointer types that are statically declared in gcc.4.5.3

As can be seen, the overall average of detected function pointers that cause in NP-hard analysis is 93% in gcc 4.5.3(by Dec 2011). That is, on average, most of the function pointer calls in gcc can potentially cause big challenges for tools and compilers that use static analysis and inter-procedural approaches. This addresses R1.

Virtual function calls were also examined in this study. We have counted the number of virtual function calls in gcc.4.5.3, including the count inherited virtual functions. Table 2 presents the number of virtual function calls in gcc. gcc has 328 function calls conducted by virtual methods, whereas 4,131 function calls are conducted via function pointers. This also answers R1.

Year	Ways Functions Are Indirectly Called							
	Virtual Methods	Function Pointers	NP-Hard Types	Global Function Pointers	Structure Field	Function Parameters	Class Member	Static Arrays
2001	194	951	900	531	243	29	89	8
2002	195	718	665	510	17	40	93	5
2003	210	2,132	2,014	1,767	39	100	103	5
2004	212	2,593	2,469	2,061	118	149	136	5
2005	225	2,913	2,801	2,403	117	144	131	6
2006	206	2,896	2,774	2,357	144	125	142	6
2007	214	3,076	2,940	2,474	154	155	151	6
2008	220	3,463	3,319	2,720	175	253	165	6
2009	236	3,429	3,273	2,601	184	298	184	6
2010	295	3,850	3,570	2,831	215	329	189	6
2011	328	4,131	3,852	3,064	225	366	191	6

Table 2. gcc collected results, 2001 to 2011, (indirect function calls via virtual methods and function pointers that cause in NP-Hard problems)

6.2 Function Pointer Types Distribution

We now address R2 and present the details of our findings on the distribution of calls conducted via function pointer types that cause in NP-hard analysis. Figure 5 presents the average percentage of each function pointer type's usage (calls) that occurs in the gcc 4.5.3 system. It has multiple function pointer types. As can be seen, global function pointers are by far the most prevalent in gcc.4.5.3 at 80%. Then, function pointers follow it as formal parameters at 9%, followed by data function pointers as function pointers that are structure fields 6%, which are followed by function pointer class members at 5%, thus addressing R2. Additionally, we lumped the function pointer types that cause in NP-hard analysis in order to show the presence of function pointer types that cause in NP-hard analysis versus other function pointer types, and results are presented in Figure 4. We see that gcc.4.5.3 has a larger average percentage of function pointer types that cause in NP-hard analysis, at 93%, than other types, which are at 7%. Figure 5 shows the

percentage of function pointer types that cause in NP-hard analysis for gcc.4.5.3 (addressing R2 as well). The figure indicates that the global function pointer type is prevalent.

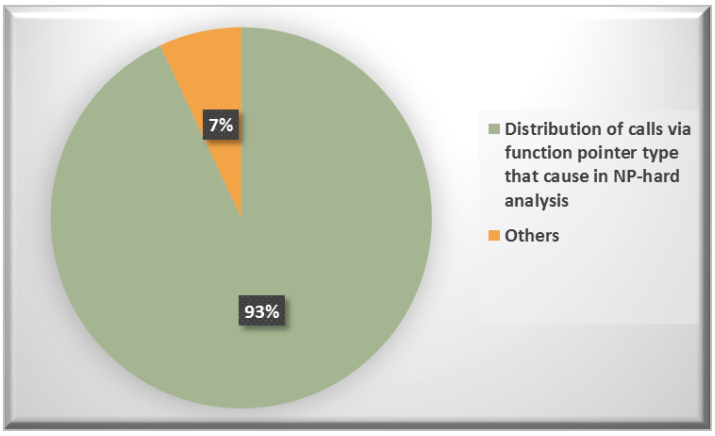

Figure 4 . Average percentage of calls via function pointer type that cause in NP-hard analysis vs. others

Clearly, function pointer types that cause in NP-hard analysis exist in all categories. In particular, it is apparent that function pointers as global variables present the most serious challenges. That is, it appears that resolving this problem and finding a form of refactoring to this type will have a very big impact on the static analysis of common software applications. On average, the next most prevalent function pointers are those that occur as structure fields and formal parameters, followed by class members, with minor occurrences of arrays of function pointers.

The main observation here is that gcc extensively uses function pointers that are known to cause in NP-hard analysis in indirect calls compared to the other types as shown in Figure 3. That is, if we have a means to resolve this problem, gcc will obviously be greatly affected, making static analysis easier.

6.3 Virtual Methods and Function Pointers declarations

To better address R2 aside from the distribution of calls via virtual methods and function pointer distribution, gcc was examined with respect to virtual method and function pointer declarations. Table 1 and Figure 3 present the number of virtual methods and function pointers statically declared in the gcc.4.5.3 system.

This includes all definitions of function pointers as parameters, in structures, in arrays, and as global declarations [1]. In gcc.4.5.3, 10,653 function pointer static declarations were detected; 10,332 of these were types that cause in NP-hard analysis. On the other hand, gcc has 32,975 virtual methods (including pure virtual methods).

7 Historical Trends

To address R3 we examined a 10-year period of gcc from 2001 to 2011. Our goal is to uncover how each gcc system evolves in the context of static and inter-procedural analysis potential difficulties caused by the usage of virtual methods and function

pointers. Here, we measure this by examining the change of virtual method and function pointer calls in gcc. Additionally, we put more focus on the change in the presence of calls via function pointer types that cause in NP-hard analysis.

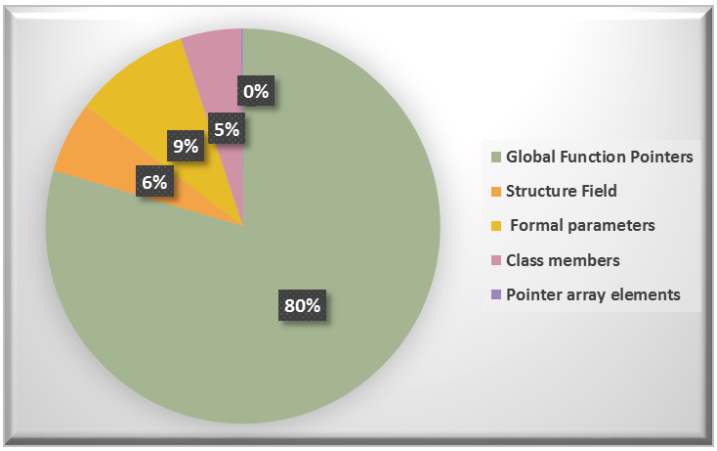

Figure 5 . Distribution of average of percentage of calls via function pointer types that cause in NP-hard analysis

We believe that such information could lead to recommendations for refactoring and improving large-scale software systems for more efficient and less complex inter-procedural and static analysis, if properly adapted. The change in the number of function calls conducted with virtual methods and function pointers along with the number of each type of the function pointers that cause in NP-hard analysis were computed for each version in the same manner as we described in the previous sections. These values were aggregated for each year so the system could be compared on a yearly basis. The system was updated to the last revision for each year. As before, all files with source code extensions (i.e., c, cc, cpp, cxx, h, and hpp) were examined and their classes, structures, functions, virtual methods and function pointers were then extracted. The Change in the number of function pointer calls of all categories and types for gcc is presented in Table 2. During the 10-year period, gcc shows a rising trend during the duration. A comparison of these two gcc versions (2002-2003) shows that the total number of function pointer calls in the system increased from 718 to 2,132, an almost 200% increase.

Table 2 also presents the number of virtual method calls for gcc, since it uses the object-oriented aspects of C++. During the 10-year period, gcc shows a fairly increasing trend during the duration. Figure 6 presents the percentage of function calls that are conducted using function pointer types that are known to cause in NP-hard analysis. During the 10-year period, gcc shows a relatively flat trend during the duration. The figure also presents the percentage of function calls via global function pointers. During the 10-year period, gcc shows a relatively flat trend during the duration. The figure shows the percentage of function calls via function pointers that are structure fields for gcc. During the 10-year period, gcc shows a relatively flat trend during the duration. The percentage of function calls via function pointers that are declared as formal parameters are presented. During the 10-year period, gcc shows a flat trend during the period.

Finally, Table 2 presents the number of function calls that are conducted by function pointer class members in gcc. During the 10-year period it shows a fairly increasing trend during the duration. A comparison of function pointer types shows that the total percentage of global function pointer usage for function calls in gcc increases over time and has been the dominant, thus the most prevalent, type. However, the other types have a fairly flat trend on average, except for function pointers declared as parameters.

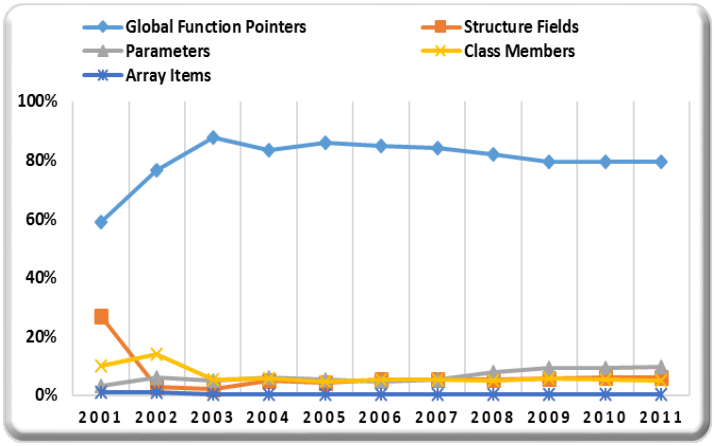

Figure 6. Evolution of calls via function pointer types that cause in NP-hard.

Much of this was due to an apparent large reengineering of the system to rely on global variables and data structure (most likely for efficiency purposes). Results, observations, and findings in regards to the observed trends are provided in the next section.

8 Discussion and Conclusion

This study empirically examines the difficulty expected from using calls via function pointers and virtual method, which most of the time cause in NP-hard analysis when static or inter-procedural approaches are applied [1, 7]. The study included a well-known, large-scale open-source software system, gcc. That is, the system is a general-purpose application. As expected, it needs to be statically analyzed both manually and automatically as it does evolve over time. There are no other recent studies of this type currently in the literature which include a historical study other than studies conducted by the main investigator in the parallelization context. We empirically addressed research questions. Obviously, the average percentage of calls performed through function pointer types that cause in NP-hard analysis is significantly larger than other types. We can easily infer that the studied systems require a lot of effort and work in order to facilitate its analysis. This means there could be a substantial increase in difficulty should any static and inter-procedural analysis be proposed for these systems, and thus poses problems for this type of analysis.

The implication here is that static analysis and inter-procedural analysis of general-purpose applications is getting more difficult to perform. That is, the cost of applying static analysis over an entire system would normally be prohibitive. Thus, the development and the design of tools and techniques that can reduce the usage of indirect calls via function pointers and virtual methods may be worth the cost and effort for more

efficient, yet less complex, static analysis. However, this is dependent on many factors including the architecture and programming style used in the system and the types of computations taking place in these systems, as well as types of function pointers used. Approaches, like the one used in [13], for eliminating function pointer and virtual method calls should also be considered.

Our findings have an important implication to the problem of statically analyzing general-purpose applications in the presence of indirect calls via special type function pointers. We found that the most prevalent type is global function pointers. This is an important finding because we can recommend developers to put more focus on this type in order to eliminate its usage for better static analysis. That is, the empirical findings show that global function pointers are the greatest roadblock to easier static and inter-procedural analysis of general-purpose applications. In fact, we see in Figure 5 that the vast majority of calls that are performed via global function pointers are amongst the types that cause in NP-hard analysis (80%).

It appears that developing methods and techniques for removing global function pointers, or changing their type, could potentially have a greater impact on the static analysis process. Additionally, it will reduce the expected challenges posed by global function pointers used in the system when tools and compilers are designed on the top of static analysis techniques. Minimally, this implies that software engineers and developers need to put more focus on addressing global function pointer type usage in general-purpose software applications. Coding practices aimed at avoiding the common types, as found in this study, can also be developed. Making developers more aware, via documentation or automated methods, of parts of code that use indirect calls via function pointers and virtual methods could also lead to more analyzable code. There are few pedagogical approaches that highlight these types of coding techniques and few documented approaches to decrease and eliminate virtual method and function pointer calls [13].

Our last research question (R3) addresses the prevalence of function pointers over the history of a system. In short, we wanted to know if the percentage of calls via virtual method and function pointer types that cause in NP-hard analysis are increasing or decreasing. We found that a great deal of function pointer and virtual method usages across most of the gcc releases show an increasing trend over time. Thus, we can surmise that developers do not pay attention to the complexity posed when trying to simplify code by providing a simple way to select a function to execute based on run-time values using function pointers and virtual methods.

Finally, these findings cannot be generalized for all open-source systems since more systems from different system domains should be analyzed and studied for a better understanding of this problem in open-source software systems.

References

[1] B. G. R. Anand Shah "Function Pointers in C - An Empirical Study," Technical Report1995.

[2] B.-C. Cheng and W. Hwu, "An Empirical Study of Function Pointers Using SPEC Benchmarks," presented at the Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, 2000.

[3] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," presented at the Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Jose, California, USA, 1996.

[4] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," presented at the Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, Orlando, Florida, USA, 1994.

[5] S. Alnaeli, J. Maletic, and M. Collard, "An empirical examination of the prevalence of inhibitors to the parallelizability of open source software systems," *Empirical Software Engineering,* pp. 1-30, 2015/05/28 2015.

[6] B. Stroustrup, "What is "Object-Oriented Programming"?," in *ECOOP' 87 European Conference on Object-Oriented Programming*. vol. 276, J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, Eds., ed: Springer Berlin Heidelberg, 1987, pp. 51-70.

[7] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs," presented at the Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Portland, Oregon, USA, 1994.

[8] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," presented at the Proceedings of the 9th European Conference on Object-Oriented Programming, 1995.

[9] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vall, #233, e-Rai*, et al.*, "Practical virtual method call resolution for Java," presented at the Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Minneapolis, Minnesota, USA, 2000.

[10] S. Zhang and B. G. Ryder, "Complexity of single level function pointer aliasing analysis," LCSR-TR-233, 1994.

[11] R. Muth and S. Debray, "On the complexity of function pointer may-alias analysis," in *TAPSOFT '97: Theory and Practice of Software Development*. vol. 1214, M. Bidoit and M. Dauchet, Eds., ed: Springer Berlin Heidelberg, 1997, pp. 381-392.

[12] K. H. Bennett, V. Rajlich, and I.-F. o. S. T. 73-87, "Software maintenance and evolution: a roadmap," in *International Conference on Software Engineering - The Future of Software Engineering Track*, 2000, pp. 73-87.

[13] G. Aigner and U. Holzle, "Eliminating Virtual Function Calls in C++ Programs," University of California at Santa Barbara1996.

[14] H. D. Pande and B. G. Ryder, "Data-Flow-Based Virtual Function Resolution," presented at the Proceedings of the Third International Symposium on Static Analysis, 1996.

[15] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," presented at the SCAM'11, Williamsburg, VA, USA, 2011.