EMPIRICALLY EXAMINING THE ROADBLOCKS TO THE AUTOMATIC PARALLELIZATION AND ANALYSIS OF OPEN SOURCE SOFTWARE SYSTEMS

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by
Saleh M. Alnaeli

May, 2015

© Copyright
All rights reserved
Dissertation written by

Saleh M. Alnaeli

B.S., Al Zawiya University, Libya, 1999
M.Sc., Technical University of Ostrava, Czech Republic, 2006
Ph.D., Kent State University, USA, 2015

Approved by

Dr. Jonathan Maletic
Chair, Doctoral Dissertation Committee

Dr. Mikhail Nesterenko
Members, Doctoral Dissertation Committee

Dr. Ye Zhao

Dr. Joseph D. Ortiz

Dr. Michael Collard

Dr. John Portman

Accepted by

Dr. Javed I Khan
Chair, Department of Computer Science

Dr. James L. Blank
Dean, College of Arts and Science
TABLE OF CONTENTS

TABLE OF CONTENTS ...........................................................................................................III
LIST OF FIGURES ..................................................................................................................VI
LIST OF TABLES ....................................................................................................................X
ACKNOWLEDGEMENTS .......................................................................................................XIII
CHAPTER 1 INTRODUCTION .................................................................................................14
  1.1 Research Overview ........................................................................................................20
  1.2 Research Questions .......................................................................................................21
  1.3 Contributions ................................................................................................................22
  1.4 Broader Impacts .............................................................................................................24
  1.5 Organization ..................................................................................................................24
CHAPTER 2 BACKGROUND AND RELATED WORK .......................................................26
  2.1 Parallel Programming for Multicore ..............................................................................26
  2.2 OpenMP .......................................................................................................................30
  2.3 Code Parallelization as an Adaptive Maintenance Task ..............................................34
  2.4 Automatic Parallelization .............................................................................................35
CHAPTER 3 PARALLELIZATION INHIBITORS ...............................................................42
  3.1 Data Dependency .........................................................................................................43
  3.2 Function Calls with Side Effects ..................................................................................48
    3.2.1 Determining Side Effects .......................................................................................49
    3.2.2 Dealing with Function Pointers and Virtual Methods ..........................................50
3.3 Jumps: Break, Goto ........................................................................................................... 53
3.4 Solvable Inhibitors: Shared/Private Data and Reduction Variables ................................. 53

CHAPTER 4 EMPIRICALLY EXAMINING THE PREVALENCE OF INHIBITORS .......................................................... 57

4.1 Methodology for Detecting Parallelization Inhibitors ...................................................... 58
4.1.1 Detecting data dependency ............................................................................................. 59
4.1.2 Detecting Function Calls With Side Effect .................................................................... 63
4.1.2.1 Call Graph ................................................................................................................... 63
4.1.2.2 Reducible Call Graph .................................................................................................. 67
4.1.3 Function pointers and virtual methods ............................................................................. 74
4.2 Data Collection ....................................................................................................................... 77
4.3 Findings of the Study ............................................................................................................. 80
4.3.1 Design of the Study .......................................................................................................... 80
4.3.2 Percentage of Free-Loops ............................................................................................... 83
4.3.3 Inhibitor Distribution ....................................................................................................... 83
4.3.4 Calls to Function Pointers/Virtual Methods ................................................................. 89
4.3.5 Historical Change of Inhibitor Frequency ..................................................................... 95
4.4 Discussion ............................................................................................................................... 100
4.5 Threats to Validity .................................................................................................................. 104
4.6 Conclusion .............................................................................................................................. 105

CHAPTER 5 PREVALENCE OF FUNCTION POINTER AND VIRTUAL METHOD CALLS .................................................... 108
LIST OF FIGURES

Figure 1. How Sequential Programs work on multicore architecture (running code in Figure 2) 19% CPU usage with extensive work on one core while others relatively idle. ........................................................... 17

Figure 2. Example, sequential for-loop. ............................................................ 17

Figure 3. How parallel code works on multicore architecture (running code in Figure 3) 100% CPU usage, all cores are working. ............................................................... 18

Figure 4. Example2, parallelized for-loop using OpenMP API.............................. 18

Figure 5. Sequential program vs. parallel program in multicore architecture [Roberts 2011] ..................................................................................................................... 19

Figure 6. Example demonstrates the use of Pthread (POSIX) with C/C++. ................. 29

Figure 7. Example of parallelizing C code using OpenMP API .................................. 32

Figure 8. Examples of types of data dependency: 1) flow-dependence (aka true dependence), 2) anti-dependence, and 3) output dependence. ........................................... 44

Figure 9. Example with potential data dependencies studied in this work .................... 45

Figure 10. Examples of Function Pointers’ Usage in C/C++. .................................... 52

Figure 11. Example shows how virtual methods are used in C++. .............................. 56

Figure 12. Example of the limitations of our approach used in our tool in dependence detection. The approach will label some situations as having a data dependency when none may actually exist due to the conservative approach for array access. .... 62

Figure 13. High level view of the reducible call graph generation by ParaStat. .......... 65

Figure 14. Call Graph Code Sample .................................................................. 66
Figure 15. Generating Graph Algorithm via Counting and Analyzing Calls within For-Loops .......................................................... 68

Figure 16. Complete (Full) call graph generated from the code presented in Figure 14. 69

Figure 17. Reduced Call Graph generated from the code presented in Figure 14........... 70

Figure 18. Adjacency Matrix Graph Functions List ...................................................... 71

Figure 19. Side Effect Detection and Propagation Algorithm ..................................... 71

Figure 20. Side Effect Propagation Approach Using Reduced Call Graph ..................... 73

Figure 21. Examples of function pointers and what Parastat detects via the approach we use in our work. ........................................................................................................ 76

Figure 22. Example of ParaStat limitations in determining side effects for virtual methods. ParaStat assumes both for-loops are not free loops, but the method definition shows that they are free................................................................. 78

Figure 23. The average percentage of non-free-for-loops that contain at least one inhibitor of the given type, over all 13 systems. Loops may contain more than one inhibitor type. .......................................................... 86

Figure 24. Percentage of for-loops in gcc that contain only a single type of inhibitor (exclusive). The remaining for-loops are either free or have multiple types of inhibitors .......................................................................................... 87

Figure 25. Percentage of for-loops in gcc that contain at least one inhibitor and percentage of the free-for-loops in gcc. If a for-loop has two inhibitors, it counts in both categories. Inclusive means that a for-loop can have a single or multiple types of inhibitors. (An accurate analysis with respect to function side effects) ............... 92
Figure 26. The evolution of the percentage of free-loops over a ten-year period for the ten systems.

Figure 27. The percentage of function-call inhibitors of for-loops over a ten-year period for the ten systems.

Figure 28. The percentage of different inhibitors of for-loops for releases during the time period of 2001 – 2003 for gcc. Note that while for-loops with function-call inhibitors increased greatly, other inhibitors do not show any increase for gcc.

Figure 29. Examples of function pointers that cause analysis to be NP-Hard, and are detected by the tool VirFptrStat.

Figure 30. Virtual Method examples as declared in classes and its inherited classes detected by VirFptrStat.

Figure 31. Average of percentage distribution of function pointer types statically declared in all 12 systems.

Figure 32. Average percentage distribution of function pointer types that are statically declared in all 12 systems.

Figure 33. Average percentage of calls via function pointer type that cause np-hard analysis vs. others.

Figure 34. Distribution of average of percentage of calls via function pointer types that cause in np-hard analysis.

Figure 35. Distribution of calls via function pointer types percentage for all 12 systems.
Figure 36. Evolution of calls via function pointer types that cause in np-hard analysis for 10 years period for all 12 systems ................................................................. 148

Figure 37. Evolution of calls via global function pointer for 10-year period for all 12 systems.......................................................... 151

Figure 38. Evolution of calls via function pointer structure fields for 10 year period for all 12 systems ........................................................................................................... 152

Figure 39. Evolution of calls via function pointer function parameters for 10 year period for all 12 system ........................................................................................................ 153

Figure 40. Average percentage of function pointer types that cause in np-hard analysis ................................................................................................................................. 155

Figure 41. Inhibitors distribution for all studied systems in Chapter 4 ................. 181

Figure 42. How a for-loop is parallelized with openMP directives ..................... 183
LIST OF TABLES

Table 1. Commonly used *OpenMP* directives ................................................................. 33

Table 2. The 13 open source systems used in the study. ..................................................... 79

Table 3. Number of loops, free loops, and inhibitors for each system found using ParaStat. The percentage is over the total number of for-loops detected for each system. A loop may have more than one inhibitor, so the total may be greater than 100%. ........................................................................................................................................ 82

Table 4. Percentage of for-loops in each system that contain inhibitors *exclusively*; divided by each type; the remaining for-loops are either free or have more than one type of inhibitors ........................................................................................................................................ 85

Table 5. comparison assuming all function calls have a potential side effect as in [Alnaeli, Alali, Maletic 2012b] and doing more complete analysis to determine actual side effect. The results on the percentage of free-loops is also given for each approach. The improvement in percentage of free-loops when analysis that is more accurate is conducted is obvious ........................................................................................................ 88

Table 6. The number of function pointers that occur in each system along with the number of for-loops that have only a function pointer call as the inhibitor. The percentage is over the total number of for-loops. For these systems the function pointer inhibitor occurs in less than 1 % of the overall for loops on average. .......................... 93

Table 7. The number of virtual functions for each C++ system along with the number of for-loops that have only a virtual function call as the inhibitor. The percentage is
over the total number of for-loops. For these systems, the virtual function-call
inhibitor occurs in less than 1% of the overall for loops on average.......................... 94

Table 8. AVERAGE INCREASE/DECREASE FROM 2001 TO 2011 OF FREE LOOPS
AND INHIBITORS IN THE 12 SYS-TEMS. THE CHROME SYSTEMS IS LEFT
OUT DUE TO ITS MUCH SHORTER HISTORY .................................................. 104

Table 9. The 12 open source systems used in the study. ........................................... 128

Table 10. Number of detected virtual methods declared in subsystem that have C++ files
for a ten-year period .................................................................................................. 130

Table 11. Number of virtual method and function pointer calls................................. 134

Table 12. NUMBER OF CALLS VIA FUNCTION POINTERS, all types, FOR A TEN-
YEAR PERIOD FOR THE TWELVE SYSTEMS .................................................. 145

Table 13. Number of calls via virtual methods for a ten-year period for sub set of the
twelve systems........................................................................................................ 146

Table 14. Number of calls via function pointer types that cause in np-hard analysis for a
ten-year period for the twelve systems ...................................................................... 147

Table 15. Percentage of calls via function pointer class members over a ten-year period
for seven systems........................................................................................................ 154

Table 16. Distribution of percentage of calls via function pointer types that cause in np-hard
analysis ....................................................................................................................... 169

Table 17. Evolution of calls via function pointer types that cause in np-hard analysis for
10 years period for all 12 systems ............................................................................... 170
Table 18. Evolution of calls via global function pointer for 10-year period for all 12 systems

Table 19. Evolution of calls via function pointer structure fields for 10 year period for all 12 systems

Table 20. Function calls in C++ systems via Virtual Methods vs. Function Pointers. Shows how some systems still use both.

Table 21. Number of Function Pointer Declared in all Systems

Table 22. Data dependency in for-loops for systems studied in Chapter 4.

Table 23. Goto statement in for-loops for systems studied in Chapter 4.

Table 24. Break statement in for-loops for systems studied in Chapter 4.

Table 25. Free for-loops for systems studied in Chapter 4.

Table 26. Function calls in for-loops for systems studied in Chapter 4.

Table 27. The percentage of different inhibitors of for-loops for releases during the time period of 2001 – 2003 for gcc. Note that while for-loops with function-call inhibitors increased greatly, other inhibitors do not show any increase for gcc.

Table 28. Inhibitors distribution for all studied systems in Chapter 4.
ACKNOWLEDGEMENTS

Most of all, I would like to thank my advisor, Dr. Jonathan Maletic, for his advice and guidance on the development and presentation of research, the results of which are also evident in this work. He has been always a friend before a supervisor, and I would like to thank him for all of his patience, support, and nice manner.

Second, I am grateful to all people who contributed in any form towards the completion of the dissertation. Without any significant order: friends, relatives, people of city of Aljmail, Libyan people, Kent State University community, computer science department’s family, my research advisors; Dr. Collard and Dr. Dragan, my teaching advisors Dr. Nesterenko, and all of SDML lab colleagues for their support.

Special appreciation for Dr. Abdulkareem Alali, Dr. Dianne Foreback and Ms. Kristi Campbell for their unlimited care and support. I am grateful to my brothers and sisters, whom I cannot imagine life without. For my brother who will always be my son Ali M. Alnaeli for his unlimited support. For Dad and Mum who taught me how to be a human and what sacrifice means, for my eyes in which I see the world through, and my children (Mohamed, Ali, Abdelmohaymen, and Raneem.)

Finally, thank you to my better second half, my wife Hanan for her everything.

Saleh M. Alnaeli

May 2015, Kent, Ohio, USA
CHAPTER 1

Introduction

Modern multicore architectures have become ubiquitous and are present in almost all of today’s desktops and laptops. In the past, this type of parallel architecture was only found in expensive specialized hardware in super computer centers. Parallelization was once reserved for such domains as weather modeling and other numerically intensive or scientific software applications. Now we have the hardware resources to parallelize all types of general-purpose software applications.

This fact has pushed the need for software engineers to rethink how the code they write can better utilize the underlying hardware. Because most of existing software systems were developed with sequential processors in mind they typically make inefficient use of the multicore technology (by using only one core in many cases). The problem gets worse as the number of cores increases which may reduce the cores speed causing slowing in sequential software speed.

The process of parallelizing a software system is typically done with one of the standard Application Programming Interfaces (API) such as OpenMP, or Pthread. These APIs provide the developer with a set of tools to parallelize loops and take advantage of multiple processors (cores) and shared memory. Current (C/C++) compilers can do a limited amount of automatic parallelization. That is, loops with fixed iteration bounds (i.e., for-loops) can, in certain situations, be directly parallelized by the compiler. Loops without fixed iteration bounds cannot, in general, be parallelized. The auto-parallelization can also
take place via a tool prior to compiling. These tools look for for-loops that do not contain any parallelization inhibitors.

Inhibitors are code within the body of a for-loop that prevents the loop from being parallelized. Data dependencies between statements in a loop are a well-studied inhibitor within the parallel computing field. Much of the literature on removal and detection of inhibitors is focused on data dependency. Other inhibitors include such things as conditional jumps to outside the loop and calls to functions with side effects. Software applications with complex scientific calculations (e.g., weather prediction) often have complex loops with large arrays and matrix computations. Data dependency can be an issue for the parallelization of such systems. However, since everyone now has a multicore processor on his or her desk, a new focus on the parallelization of general-purpose applications is at hand. Because of the potentially complex and difficult nature of automatic parallelization process, new techniques and tools must be developed to support programmers working in this domain.

This work is addressing the parallelization and analysis problem at the source code level, rather than at the compiler level. In addition, the process of parallelizing an existing software system is viewed here as an adaptive maintenance problem. That is, the software system is undergoing a platform change and needs to take advantage of, and function on, the new underlying hardware.

Figure 1 and Figure 2 show an example of how sequential code works on multicore architecture supported machine. The figure shows that the overall usage of the CPU was 19%. That is, one core was extensively doing the job while other cores are relatively in an
idle status. That explains how a sequential code cannot take advantage of the multi core technique. On the other hand, Figure 3 and Figure 4 demonstrate how the same code is working more efficiently when it is parallelized on the same device. Almost 100% of the CPU usage. That is, all available cores in fully used making the code faster and effectively getting the advantage of the multicore architecture. Code in Figure 4 is parallelized by one of the OpenMP directives which will be briefly explained in section 2.2.

Figure 5 presents how a sequentially designed system is run on a multicore architecture versus a parallelized software system. The computation processing is divided among cores in a balanced manner. Balancing is either controlled by developer if a flexible API is used or automatically if an API like OpenMP is used.
Figure 1. How Sequential Programs work on multicore architecture (running code in Figure 2) 19% CPU usage with extensive work on one core while others relatively idle.

```c
// Sequential
for (int i = 0; i < 1000000000000; i++)
{
    tmp = i + i / 2;
    for (int j = 0; j < 1000; j++)
        tmp = i / 2 + j / 4 + 2;
}
```

Figure 2. Example, sequential for-loop
Figure 3. How parallel code works on multicore architecture (running code in Figure 3) 100% CPU usage, all cores are working.

```c
// Parallel
omp_set_num_threads(8);
#pragma omp parallel for
  for (int i = 0; i < 100000000000; i++)
  {
    tmp = i + i/ + 2;
    for (int j = 0; j < 1000; j++)
      tmp = i /2 + j / 4 + 2;
  }
```

Figure 4. Example2, parallelized for-loop using OpenMP API.
Figure 5. Sequential program vs. parallel program in multicore architecture [Roberts 2011]
1.1 Research Overview

The goal of this research is to empirically investigate how we can better support the (semi) automated adaptation of existing software systems to utilize parallel hardware architectures. That is, to define new tools and techniques to support the process of software maintenance in parallelization context. This would be accomplished via transformational process of the source code. The transformations support the insertion of API (Application Programming Interface) directives (e.g., OpenMP) in the appropriate locations along with related modifications of the code as necessary.

More specifically, the investigation focuses on the development of techniques to conduct analysis of the parallelizability of software systems written in C, C++ programming languages. To accomplish this, a set of methods to efficiently analyze large-scale software systems is developed. These methods identify loops within the software and determine what types of inhibitors, if any, are contained in each loop. The objective here is to develop metrics to assess the parallelizability of a system. Additionally, methods to identify and measure the impact of function pointers and virtual method calls on static program analysis are developed.

A series of empirical studies is conducted for the examination of a variety of general-purpose open source software systems to better understand the roadblocks for automated and/or semi-automated parallelization tools. The main interest is determining the most prevalent inhibitors that occur in a wide variety of software applications and if there are general trends. While this work does not directly address the problem of
automated parallelization, it does serves as a foundation for understanding the problem requirements in the context of a broad set of applications.

Moreover, the focus of this research is on inhibitor detection and removal in the most common situations occurring within typical software systems. Furthermore, the research include the analysis of different versions of a system’s history to uncover evolutionary patterns related to parallelization process.

This work is unique in that it is at the source code level rather than compiler level optimizations. This better facilitates long-term maintenance and optimization of the software systems under consideration.

1.2 Research Questions

The dissertation addresses the following main research questions in context of automatic parallelization:

RQ1: What is a typical percentage of for-loops that are free loops (have no inhibitors) in general-purpose large-scale open source software systems?

RQ2: Which types of inhibitors are the most prevalent?

RQ3: Which types of inhibitors are the most prevalent exclusively?

- R3a: Data dependencies are a focus in the research literature. How prevalent are they as potential inhibitors?

- R3b: Complex analysis is needed for function pointers/virtual methods calls. How prevalent are they as potential inhibitors?

RQ4: Over the history of a system, is the presence of inhibitors increasing or decreasing?
RQ5: What is a typical distribution of function pointer types in open source system, and how much they are used, and Which type is the most prevalent and used?

RQ6: Over the history of a system, is the presence of function pointers and virtual methods calls increasing or decreasing?

1.3 Contributions

The primary contributions presented in this dissertation are, broadly, the study and description of inhibitors to automated parallelization and demonstrate which inhibitors are most prevalent. Additionally, the development of new source code analysis techniques and tools for analyzing large-scale software repositories in order to assess the potential of how well a software system, written for a sequential hardware, can potentially take advantage of multi-core platforms. Results of studies can be incorporated to other methods to propose better recommendations for developers in how to plan better refactoring or automatic parallelization technique if considered.

Specific contributions the study and description of software parallelizability include:

- It is one of the only large studies on the potential to parallelize general-purpose software applications,

- Demonstrate which inhibitors to automated parallelization are most prevalent in open source software systems.

- Describe how coding style can play a big role in advancing a system’s parallelizability by comparing systems from different domains.
- Recommend the software engineering community needs to develop standards and idioms that help developers in avoiding the inhibitors.
- Develop a new approach to automatically generate directives for the compiler to help it to recognize if the functions contain side effects for better parallelization and compilation time.
- Describe the impact of function calls via function pointers and virtual methods on the parallelization process, and the applicability of considering a conservative approximation solution.
- Examination of the prevalence and distribution of calls using function pointers and virtual functions in general-purpose software systems.
- Analysis of different versions of a system’s history to uncover evolutionary patterns related to parallelization process and usage of function calls through function pointers and virtual methods and how they evolve overtime in large-scale open source systems.

Additionally, the dissertation includes discussion on techniques for the design of approaches supports a type of preprocessing and labeling such as labeling methods with stereotypes [Dragan, Collard, Maletic 2009a], which could be of great use for compilers, as they typically avoid analyzing functions called within the bodies of for-loops that are considered for automatic parallelization.
1.4 Broader Impacts

The work presented in this dissertation will directly support the enhancement and quality of software systems through improved program comprehension and software evolution methods and tools. The modern multicore architectures have become ubiquitous and are present in almost all of today’s desktops and laptops, nearly every business, government office, and hospital, our home computing and entertainment systems, our mobile phones and personal computing devices, and in our transportation system and automobiles. The improvements engendered by the research presented in this dissertation will be reflected in the performance thus the quality of the many general-purpose systems that function as the backbone of our daily lives if they take advantage of multi-core platforms.

Additionally, aspects of the work presented in this dissertation can be leveraged to develop new academic curriculums and methods for the training and teaching of multicore programming, and source code that can be easily parallelized if needed. The work also supports the extension of software engineering topics and a very promising research direction to address problems related to automatic parallelization, adaptive maintenance, evolution and transformation of large-scale open source systems.

1.5 Organization

The dissertation is logically organized into three components: background reading, understanding source code parallelization process (manually and automatically) and inhibitors, techniques for detecting parallelization inhibitors and examining the potential to parallelize a software system.
Background reading is presented in Chapter 3. Chapter 3 gives an overview of parallel programming for multicore concepts and the proposed solutions using API’s. It also describe how the parallelization process in the source level is viewed as a type of adaptive maintenance changes in software engineering.

The second component focuses on automatic parallelization and inhibitors. It describes how some of the inhibitors are solvable while other inhibitors are not. Chapter 4 presents the inhibitors to parallelization and how parallelization Application Programming Interfaces (API) such as OpenMP can solve some of them while others needs different software engineering approaches and techniques to process them.

The third component focuses on tools and empirical studies to support automatic parallelization process and inhibitors detection. Chapter 4 present an empirical study that examines the potential to parallelize general-purpose software systems. It also describes tools and techniques for inhibitors detection. Chapter 5 describes an empirical study that examines the prevalence and distribution of function calls using function pointers and virtual methods in general-purpose software systems is presented. Conclusions and future work are given in Chapter 6, and Appendix A includes some extra results for additional reference.
CHAPTER 2

Background and Related Work

In this chapter, we cover a range of topics related to our work. We present an overview of the code parallelization and programming for multicore in shared memory architectures in Section 2.1. OpenMP, an Application Programming Interface (API) which is used by developers of shared memory general purpose C/C++ parallel applications is presented in Section 2.2. Section 2.3 explains how the process of source code parallelization is considered as an adaptive maintenance task in software engineering. After that, we talk about the process of automatic parallelization and its challenges in Section 2.4. Finally, we present a background and related work on automatic parallelization process.

2.1 Parallel Programming for Multicore

Recently, the introduction of chips with multicore architectures has become very prevalent for general computing needs. This introduction has created a revolution accompanied with challenges in the software industry and computing market. Almost all of today’s computers and mobile devices are leveraged with multicore architectures. This fact has pushed the need for software engineers to think more about how the code they write can better utilize the underlying hardware, Figure 1. The challenge gets harder as the number of cores integrated onto a single chip increases. This may poses a possible need to reduce the cores speed causing slowing in sequential software speed. Consequently,
developers and software engineers are left with no solution to increase hardware performance by parallelizing their code. Additionally, developers are required to invest in significant software modifications to transform on use sequential software systems into parallel ones. This makes parallel programming a concern for the entire computer science research community and computing industry.

Parallelization of programs is typically done with one of the standard APIs such as OpenMP, or Pthread. These APIs provide the developer with a set of tools (e.g. directives) to parallelize loops and take advantage of multiple processors (cores) and shared memory. The API is a set of standards and interfaces for parallelizing programs in a shared memory environment. It provides a set of compiler directives and pragmas for C/C++ that can be used in a program to instruct compilers to parallelize pieces of code. The sequential code is incrementally parallelized and the program can contain both serial and parallel code.

Developers have the choice among a variety of approaches for introducing parallelism into their code. Those approaches vary in their nature and capability. For example, developers can choose to use an API that offers the flexibility of threading development (e.g., PThread aka POSIX). With such APIs, developers have more flexibility and control over threading than with other APIs that are easy to use with limited control over threads[(Intel) 2012; Intel 2011].

In spite of the better control developers gain from using APIs like POSIX, it takes longer and requires more effort to implement a programming task. That is, the developer is responsible for creating and scheduling threads. Moreover, the developer needs to maintain load balancing among working threads as well as to use a proper number of cores
from the available cores supported by the expected devices that will be running the software system being developed. Consequently, developers will need to focus on the parallelization issue along with the role of a system designer. Another issue with API’s like PThread is the difficulty of using it in the automatic parallelization of existing software systems compared to other API’s that use a compiler directive to perform thread creation, balancing, and destruction automatically (e.g., OpenMP.). With OpenMP, developers use compiler directives and pragmas to describe parallelism to the compiler that handles the complex and time consuming details. In this work, we consider OpenMP as the parallelization approach that can be used for the automatic parallelization process. Section 2.2 presents an overview of OpenMP API usage and advantages.
```cpp
using namespace std;

#define NumberOfThreads 7

void *PrintMessage(void *threadid)
{
    long id;
    id = (long)threadid;
    cout << "Hello Thread Number : " << id << endl;
    pthread_exit(NULL);
}

int main()
{
    pthread_t Mythreads[NumberOfThreads];
    int R;

    for (int i = 0; i < NUM_THREADS; i++){
        cout << "New thread is created, " << i << endl;
        R = pthread_create(&Mythreads[i], NULL,
            PrintMessage, (void *)i);
        if (R){
            cout << "Can not create thread : " << R << endl;
            exit(-1);
        }
    }
    pthread_exit(NULL);
}

Figure 6. Example demonstrates the use of Pthread (POSIX) with C/C++.
```
2.2 OpenMP

OpenMP is an API used for writing parallel code for shared memory general-purpose software systems through a set of directives supported by a wide range of today’s C/C++ compilers. That is, groups of directives (pragmas) that can be instrumented to the source code to specify shared-memory parallelism in C/C++ source code. OpenMP is a powerful yet easy to use API to introduce parallelism in both loop and function levels for better hardware utilization [(Intel) 2012; Nawal Copty 2007]

With OpenMP, developers can incrementally parallelize their code. They just need to insert a proper pragma in a proper place in the source code, and then use a compiler that supports OpenMP specifications to run their code as shown in Figure 7. Additionally, developers can combine sequential and parallel code in their software systems. With OpenMP, developers do not need to worry about thread or core numbers. Additionally, developers need not worry about the thread work balance because it is done by the compiler. OpenMP is the proper approach if developers desire to remove the challenges caused by explicit threading and parallelization usage.

OpenMP makes code transformation fairly simple; any transformation can be achieved by embedding the appropriate pragmas at the proper locations. That is, the transformation of sequential code to parallel is conducted just by determining the parallelizable portions of code and then inserting a proper directive or pragma to a proper place with almost no changes to the original code. Additionally, if the transformed code is compiled on a compiler that does not support OpenMP, it simply ignores the directives and
can be executed as the original sequential code. Commonly used OpenMP directives, which we will be talking about in this dissertation, are presented in Table 1.
Figure 7. Example of parallelizing C code using OpenMP API
<table>
<thead>
<tr>
<th>OpenMP Directive</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>Parallel construct forms a team of threads and starts parallel execution</td>
</tr>
<tr>
<td>#pragma omp parallel for reduction(+ : Variable)</td>
<td>This Clause Makes the reduction variable shared to generate the correct results but private to avoid race conditions from parallel execution</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td>Combined parallel construct for for-loop. Parallelizes the loop that immediately follows the pragma.</td>
</tr>
<tr>
<td>#pragma omp parallel for private(x)</td>
<td>Variable is used within a for-loop and is meant by the developer to be private</td>
</tr>
<tr>
<td>#pragma omp critical</td>
<td>The critical construct restricts execution of the associated structured block to a single thread at a time</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>Ensures all the code occurring before the barrier has been completed by all the threads, before any thread can execute any of the code past the barrier directive</td>
</tr>
</tbody>
</table>
With all the facilities offered, OpenMP has a number of limitations, for example, pragmas can only be applied to for-loops with known iteration bounds. This implies that while-loops are not parallelized. Additionally, OpenMP does not analyze code correctness, and so it cannot detect un-parallelizable code. As a result, it will generate code that produces an unexpected result [binstock 2010]. In conclusion, OpenMP is a very important and easy to use API. However, it requires that developers determine the parts of code that are good candidates for parallelization.

2.3 Code Parallelization as an Adaptive Maintenance Task

At many organizations, some software systems play a crucial role in their business success. These systems are considered great assets and usually represent substantial investments. These systems have usually very long lifespans [Collard 2010]. Consequently, they grow as the business improves and different requirements are introduced by the market. Not only do these systems evolve due to new requirements that are introduced, they also need to be adapted to new changes of dependent platforms and libraries. These changes can be both software and hardware. Changes that are external to the system and organizations are known as adaptive maintenance changes that require adaptive maintenance tasks [Collard 2010].

The process of parallelizing an existing software system is viewed here as an adaptive maintenance problem. That is, the software system is undergoing a platform change and needs to take advantage of, and function on, the new underlying hardware. This approach has many benefits. First is that the adaptive maintenance problem can take advantage of automated tools along with developer intervention. The automated tools may
take care of a large percentage of the adaptive maintenance task but particularly difficult situations will require human input. It will also allow developers to optimize and refactor the source code so that the compiler can better optimize it for the underlying hardware. This explicit (in the source code) parallelization will also support targeted testing and should lead to higher quality and more dependable systems. Lastly, by adapting and modifying the code to support parallelization it supports long-term maintenance of these systems.

Parallelizing the existing software systems can be a tremendously time consuming and risky task because of the expected errors and bugs that can be introduced if a manual approach is considered. That is, the manual process is an error prone and difficult task to properly and safely complete in a practical time. This process requires developers to invest in significant software modifications to transform current sequential software systems into parallel ones. Different approaches are usually followed in order to conduct this task safely and in a timely manner. One of these approaches is the automatic parallelization.

The following section talks about the process of automatic parallelization with concentration on loop parallelization, which is our focus in this work.

2.4 Automatic Parallelization

Source code transformation from sequential to parallel is a time consuming and error prone task. For general-purpose applications where most of the extensive computation processing is performed in iterative statements, (e.g., for-loops), tools and techniques are needed to transform the sequential source code to a multithreaded, parallel version automatically and semi-automatically.
With automatic parallelization, developers are usually relieved from determining portions of code that are qualified for multithreading. Additionally, developers can, depending on tools or compilers that supports automatic parallelization, analyze the code and inject it with compiler directives that are used for parallelization within the source code (e.g., OpenMP API).

In this study, implicit parallelism with the shared memory parallel model [Barney 2012; Bik, Gannon 1997] is considered and we intend to support the parallelization of existing sequential code. While there are multiple APIs used for parallel programming (e.g., MPI, PThread) the most common is OpenMP and our discussion is within the context of using this API. OpenMP is a widely accepted standard and most compilers support it [Nadgir 2001; Nikolopoulos et al. 2001].

Current parallelization APIs have common for-loop parallelization inhibitors. Some are solvable by special OpenMP pragmas. For example, variable reduction and private variables are solvable in a straightforward manner. However, some are unsolvable with the API or too difficult to parallelize. These include data dependences, goto/break statements, and function calls with side effects. Here we are only interested in the latter problematic inhibitors.

The bulk of previous research on this topic has focused on detecting and dealing with data dependencies, particularly in the context of array indices [Psarris, Kyriakopoulos 1999] even though, as we will show, there are many other inhibitors that appear to be more frequent [Intel 2011]. There is a large body of work on parallelization. In the 1960s, parallel computers and research on parallel languages and compilers first began. The focus
was instruction-level parallelism [J.L. Hennessy 2006] and mainly involved detecting instructions in a program that could be executed concurrently to reduce computation time. Since this time, many compilers and tools have been developed for identifying parallelizable portions and fragments of code in the system. Loops are the main focus of auto-parallelization or vectorization. Parallelizing compilers divide loop iterations so that they can be concurrently executed on separate processors or cores.

Parallelizing compilers, such as Intel’s [Intel 2011] and gcc, have the ability to analyze loops to determine if they can be safely executed in parallel on multi-core systems, multi-processor computers, clusters, MPPs, and grids. The main limitation is effectively analyzing the loops. For example, compilers still cannot determine the thread-safety of a loop containing external function calls because they do not know whether the function call has side effects that would introduce dependences.

Tools separate from production compilers have been developed to assist in situations where the parallelizing compiler cannot assist. Kim et al. [Kim 2010] introduce Prospector, a profile-based parallelism identification tool that uses dynamic data dependence profiling. It advises on how to parallelize selected sections of code. The authors demonstrated that Prospector is able to discover potentially parallelizable loops that are missed by state-of-the-art production compilers. Dig et al [Dig et al. 2009] present ReLooper, an Eclipse-based refactoring tool, that performs two important tasks required for refactoring regular arrays into parallel arrays in Java. The main tasks are analyzing whether the loop iterations are safe for parallel execution, and automatically replacing loops with the equivalent parallel operations.
The SUIF parallelizer [Robert P. Wilson 1994] translates sequential programs into parallel code for shared address space machines. SUIF does several passes to determine the optimizations. First, a number of scalar optimizations help to expose parallelism. These include constant propagation, forward propagation, induction-variable detection, constant folding, and scalar-privatization analysis. Second, unimodular loop transformations guided by array-dependence analysis restructure the code to optimize for both parallelism and locality. Finally, the parallel-code generator produces parallel code with calls to the parallel run-time library.

The PGI compilers [Group 2012] support C++ and FORTRAN and offer features including auto-parallelization for multicore and OpenMP directive-based parallelization. The source code is parsed for good candidate parallelizable loops. Selected loops are then parallelized and the developers are informed. In spite of the sophisticated analysis and transformations performed by the PGI compiler, limitations still exist including innermost loops and timing loops.

For function pointer and virtual function analysis there are multiple algorithms used for call graph construction. Our concern in this study is to support function side effect detection and to provide insight on the impact of function pointers on the parallelization process and the distribution of types of function pointers in each system. Virtual functions were also considered for the analysis to show some insights of their usage and impact on the parallelization of existing sequential code.

The majority of previous research on this topic has focused on detecting and resolving virtual function and function pointers dealing with interprocedural analysis and
call graph construction, particularly in the context of static program analysis [Bacon, Sweeney 1996b; Calder, Grunwald 1994; Grove, DeFouw, Dean, Chambers 1997; Shah Anand 1995a; Sundaresan et al. 2000]. However, no study has been conducted on the evolution of the open-source systems over time in terms of function pointer and virtual function usage in a parallelization context [Bliss 2007]. In this work, we are examining the history of each release of a system in this regard.

Ben-Chung et al. [Cheng, Hwu 2000b] 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. Real examples and samples of function pointers usage in the benchmark studied are shown, as an attempt to explore the issues that might be critical in the design of a complete interprocedural pointer analysis algorithm. The observation is made that the call graph construction problem has become an interprocedural pointer analysis problem, as all pointers need to be analyzed and resolved for correct results.

Ryder et al. [Shah Anand 1995a] examined multiple systems from different domains by statically gathering empirical data on C function pointer usage in an attempt to better predict the appropriate interprocedural analyses required for C programs. They classified and categorized the programs based on ease of call multi-graph construction with the presence of function pointers. They observed that calls to global function pointer variables far outnumber the calls to any other kind of function pointers, which agrees with our observations on most of the systems we studied. However, the study was done on relatively small to medium scale C systems quite some time ago, and virtual functions were
not considered. A newer technique presented by Sundaresan et al. [Sundaresan et al. 2000] can be used to estimate the possible types of receivers for virtual method and interface calls in Java. Their design goal was to develop a competitive technique that can be solved with only iteration, and thus scales linearly with the size of the program, while at the same time providing more accurate results than what existed at that time. Two popular linear techniques, class hierarchy analysis and rapid type analysis [Dean, Grove, Chambers 1995], were applied to Java programs. They observed that the extra call sites resolved by variable-type analysis account for a significant number of calls in the dynamic trace, and demonstrated that inlining could make use of these extra call sites, giving performance improvement for two benchmarks. Their results were better than what was achieved by rapid type analysis.

The work presented here differs from previous work on parallelization in that we conduct an empirical study of inhibitors to parallelization. We empirically examine a number of systems to determine what roadblocks exist to developing better tools for automatic parallelization as well as call-graph construction and inter-procedural analysis, and show how these systems evolve overtime in terms of analysis difficulty based on the usage of virtual functions and function pointers.

In this work, we are particularly interested in for-loop automatic parallelization. The objective of this study was to better understand what obstacles are in place for advancing the reengineering of systems to better take advantage of parallelization through code transformation (refactorings), at the compiler level, and by programmers (coding standards). We are particularly interested in tools that assist developers in an automated
or semi-automated manner to refactor or transform parts of a code base to facilitate parallelization by other tools (i.e., compilers).

In order for a for-loop to be parallelized, it needs to be free of any inhibitors to parallelization. That is, a loop must not contain any inhibitors in its body. Inhibitors are all source code or data relations that can make the parallelized code does not behave or work as supposed to be and resulting in inadequate results. Tools and automatic techniques cannot transform for-loops to be parallelized unless it is determined to be clear or free of any parallelization inhibitor. The existence of inhibitors in the source code posed more challenges and complexity to the development and the design of any emerged automatic parallelization technique. There are many parallelization inhibitors presented and introduced in the next chapter in details. This dissertation proposes new tool and techniques that can be used for the detection of free for-loops that can be safely parallelized.
CHAPTER 3
Parallelization Inhibitors

This chapter describes the different inhibitors to parallelization. Particularly, we interested in for-loop parallelization inhibitors. That is because, in most applications, the extensive computation is carried out in loops and parallelization APIs like OpenMP can parallelize only for-loops. However, not all for-loops are parallelizable. For example, for-loops whose results are used by other iterations of the same loop will not properly work, and can lead to unexpected and incorrect results [binstock 2010]. Some inhibitors can prevent for-loop parallelization, and some of them are solvable and some are not, these include (1) data dependency; (2) function calls with side effects (3) jumps (goto and break) (5) shared and private data (6) reduction variables (7) exit and exceptions.

Some inhibitors has a direct solution in Application Programming Interfaces such as OpenMP, and others cannot be solved and demand more complex (conservative) approaches. In this study, a for-loop is considered a free-loop if it does not contain any parallelization inhibitors that are not already solvable with OpenMP.

In this chapter, the data dependency is discussed first followed by function calls with side effects, and then jump statements (e.g., break, goto). Finally, inhibitors that are solvable by OpenMP are discussed in last the section.
3.1 **Data Dependency**

One of the most well studied conditions that inhibit for-loop parallelization is data dependency. In many situations, the order of statement execution within the body of the for-loop must be preserved to gain the same results as when executed in sequential order. That is, all loop iterations must be independent and no dependency relation should exist between two different iterations. Data dependency analysis is a major concern and an essential stage for compilers doing optimization and automatic parallelization, as well as many software engineering activities [Orso, Sinha, Harrold 2004; Psarris, Kyriakopoulos 1999].

Data dependency analysis is used to detect and identify portions and fragments of the code as well as the for-loops that can be safely executed in parallel. Several tests and algorithms have been developed based on approximation or integer programming algorithms. They are well covered in the literature [Banerjee 1988; Kong, Klappholz, Psarris 1991; Petersen, Padua 1996]. Typically, the more precise the technique is the worse the efficiency. All methods are conservative in the case of dependency suspension, or when it is difficult to prove the opposite, so that no unsafe parallel transformation is done [Petersen, Padua 1993; 1996].

The main purpose of data-dependency analysis is to detect if the same memory position is used in more than one loop iteration. Indices that are used in iterating and controlling the for-loops are often used in array references (subscripts). Multiple indices may be used to reference an array element, as in the case of nested loops. The subscripts are usually presented as functions that can be linear or nonlinear; Analysis of nonlinear
functions is very complex. The majority of dependency analysis tests are focused on array references [Jacobson 2003; Psarris, Kyriakopoulos 1999].

1) for (i=1; i<100; i++){
   S: M[i*2] = Data1[i-1]*0.25 ;
   T: Data2[i]= 0.5 + M[2*i-4];
}

2) for (i=1; i<100; i++){
   S: Data2[i] = 0.5 + M[2*i-4];
   T: M[i*2] = Data1[i-1]*0.25 ;
}

3) for (i=1; i<100; i++){
   S: M[i-1] = Data1[i-1]*0.25;
   T: M[i] = 0.5 + Data2[2*i-4];
}

**Figure 8. Examples of types of data dependency: 1) flow-dependence (aka true dependence), 2) anti-dependence, and 3) output dependence.**

We now briefly discuss the various types of data dependency. Also, we explain data dependency as a relation between assignment-statements contained within the for-loop with array references. Assignment statements of scalar variables can easily be parallelized using OpenMP. Control dependency is not an issue here since it does not directly affect the automated parallelization task. When a statement refers to the data modified by a previous statement there is a data dependency. This is particularly problematic for array accesses.
for (i=1; i<100; i++){
    M[i*2] = Data1[i-1]*0.25;
    Data2[i] = M[2*i-4]+ Data3[i+1];
    t = i + 4;
    Data2[i-1] = i*i;
    Data3[Data4[t]] = fun(Data1[Data2[t-1]]);
    Temp = Data4[i];
    constRef[7] = constRef[3];
}

Cases that are reported as data dependency:
1) M[i*2] and M[2*i-4] → Flow-dependence
2) Data3[Data4[t]] and Data3[i+1] → Anti-dependence, assuming that Data4[t] is less than i+1
3) Data2[i] and Data2[i-1] → Output-dependence

Figure 9. Example with potential data dependencies studied in this work.
There are three types of dependency based on the manner and sequence of accessing a memory location. They are 1) flow-dependence (aka true dependence), 2) anti-dependence, and 3) output dependence. A fourth type, input dependence is not considered here because it does not meet the condition that at least one access is a write to memory [Banerjee 1988; Jacobson 2003; Kennedy 2002; Kulkarni 1993]. Figure 8 shows examples of data dependency types that can be detected by our tool, ParaStat, that is introduced in Chapter 4.

Figure 9 presents examples of the three different types of data dependency. For each example there are two assignment statements S and T in a for-loop body. Both S and T reference a memory location M, such that one of S or T is a write access. Line S is executed before T in the sequential program that is being analyzed. Flow-dependence (Figure 9 part 1) occurs when S modifies M while T reads M. That is, one statement is dependent on the result of a previous statement. When i is 3, the memory location presented by M[2*i] (i.e., M[6]) and computed in S, is the same memory position referenced by M[2*i-4] in T, when i is 5, M[10-4]. Anti-dependence (Figure 9 part 2) occurs if S reads M while T modifies it when a statement requires a value that is later updated. An anti-dependence occurs between S and T whenever i in S is y and i in T is x where y = x+2. So, when i is 3, the memory location presented by M[2*i] (i.e., M[6]) and computed in T, is the same memory position referenced and used by M[2*i-4] in S, when i is 5. Output-dependence (Figure 9 part 3) occurs when the ordering of the statements affect the final result [Banerjee 1988; Jacobson 2003]. Here S is output-dependent on T.
That occurs when $M[i]$ in $T$ points to the same memory position that is later used and presented by $M[i-1]$ in $S$.

In this work, we take a conservative approach to detecting data dependency and detect all potential dependencies. We say there is data dependence in a for-loop if two statements refer to the same array with one statement referencing the array as a l-value and the other referencing the array as an r-value, or both are l-values. Additionally, if the same array appears as both a l-value and a r-value in the same statement, we also include this as a potential data dependence. Examples of these situations are given in Figure 9.

3.2 Function Calls with Side Effects

Another situation that can inhibit parallelization is calling functions or routines that have side effects within a for-loop. Today's compilers cannot parallelize any loop containing a call to a function or a routine that has side effects. A side effect can be produced by function call in multiple ways. Basically, any modification of the non-local environment is referred to as side effect [Ghezzi, Jazayeri 1982; Spuler, Sajeev 1994] (e.g., modification of a global variable or passing arguments by reference). Moreover, a function call in a for-loop or in a call from that function can introduce data dependence that might be hidden [Oracle 2010]. The static analysis of the body of the function increases compilation time; hence this is to be avoided. As such, it is usually left up to the programmer to ensure that no function calls with side effects are used and the loop is parallelized by explicit markup using an API. There are many algorithms proposed for side-effect detection [Banning 1979; Spuler, Sajeev 1994], with varying efficiency and complexity.
In general, a function has a side effect due to one or more of the following:

1. Modifies a global variable
2. Modifies a static variable
3. Modifies a parameter passed by reference
4. Performs I/O
5. Calls another function that has side effects

3.2.1 Determining Side Effects

In order to determine if a function/method has a side effect we do static analysis of the code within the function/method. Any variables that are directly modified via an assignment statement (e.g., \( x = x + y \)) are detected by finding the l-value of an expression that contains an assignment operator, i.e., =, +=, etc. For each l-value variable it is determined if it has a local, non-static declaration, or is a parameter that is passed by value. If there are any l-value variables that do not pass this test, then the function is labeled as having a side effect. That is, the function is modifying either a global, static, or reference parameter. This type of side effect can be determined with 100% accuracy since the analysis is done local to the function only.

Of course, pointer aliasing can make detecting side effects on reference parameters and global variables quite complex. Our approach detects all direct pointer aliases to reference parameters and globals such as a pointer being assigned to a variable’s address (\( \text{int } *\text{ptr}; \text{ptr} = \&x; \)). If any alias is an l-value we consider this to cause a side effect. However, we currently do not support full type resolution and will miss some pointer variables. Also, there are many complicated pointer aliasing situations that are extremely
difficult to [Mock, Atkinson, Chambers, Eggers 2005] address even with very time consuming analysis approaches. For example, the flow-sensitive and context-sensitive analysis algorithms can produce precise results but their complexity, at least $O(n^3)$, makes them impractical for large systems [Mock, Atkinson, Chambers, Eggers 2005]. As such, our approach to detection of side effects is not completely accurate in the presence of pointer aliasing. However, this type of limited static pointer analysis has shown [Alomari et al. 2014] to produce very good results on large open source systems.

3.2.2 Dealing with Function Pointers and Virtual Methods

It is very challenging to statically analyze programs that make function calls using function pointers [Cheng, Hwu 2000b; Shah Anand 1995b] and virtual methods [Aigner, Holzle 1996; Bacon, Sweeney 1996a; Calder, Grunwald 1994]. A single function pointer or virtual method can alias multiple different functions/methods and determining which one is actually called can only be done at run time. An imprecise, but still valid, analysis is to resolve all function pointers in the system and then assume that a call using each function pointer/virtual method reaches all possible candidate functions in the program. This, of course, adds more complexity and inaccuracy to static analysis. In general, the problem has been shown to be NP-hard based on the ways function pointers are declared and manipulated in the system [Aigner, Holzle 1996; Calder, Grunwald 1994; Cheng, Hwu 2000b; Shah Anand 1995b; Zhang, Ryder 1994b].

It has always been assumed that for accurate interprocedural analysis, function alias analysis is a very important step that should be always seriously considered for better
results [Emami, Ghiya, Hendren 1994a; Zhang, Ryder 1994b]. For safe analysis and parallelism, the set of possible targets of a function pointer call must be determined.
Figure 10. Examples of Function Pointers’ Usage in C/C++. 

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:};
If one item in this set has a side effect, parallelization may not be safe resulting in the conservative decision to inhibit parallelization [Bacon, Sweeney 1996a]. Our approach for calls using function pointers and virtual methods is to assume that all carry side effects. At the onset, this may appear to be a problematic, however conservative, limitation. However, this assumption is supported by empirical analysis we undertook. We present the complete details of this data later in Chapter 4 and the data is summarized in
Table 5.

3.3 Jumps: Break, Goto

Breaks and goto statements are inhibitors to parallelization of for-loops. That is, the loop must be a basic block, meaning no jumps outside the loop are permitted. As such, the occurrence of one of these statements prevents parallelization of the loop. It is very simple to detect any and all occurrences of break and goto statements in source code so counting them is accurate.

A call to exit() can be handled by OpenMP so we do not consider these as loop inhibitors. Also, the same applies to exception handling. Exceptions thrown in a parallel region and caught within the same region are safe for parallelization. Catches can be inserted into those regions automatically if they do not exist. Since there is a know solution for exceptions we do not consider them as inhibitors.

3.4 Solvable Inhibitors: Shared/Private Data and Reduction Variables

In addition to the three inhibitors discussed, other possible inhibitors include shared data, some types of private data, and reduction variables. However, each of these inhibitors has a direct solution in OpenMP. Variables that are shared among all threads are problematic if one thread is reading from a shared variable, at the same time as another thread is writing to the same shared variable. Certain types of private data cause issues as they are not shared, but require initialization from the non-parallel code before the loop starts (i.e., the OpenMP directive firstprivate), or the final result is required after the loop ends (i.e., the OpenMP directive lastprivate). In this study we do not consider any of these
since OpenMP has complete solutions to deal with these situations. Control of shared data can be coordinated by using the OpenMP directive critical (e.g., #pragma omp critical), or OpenMP lock routines. If a variable is used within a for-loop and is meant by the developer to be private, OpenMP can accommodate that situation by using the private directives (e.g., #pragma omp parallel for private(x) firstprivate(j) lastprivate(k)).

Reduction variables are also an inhibitor. A reduction variable is one whose partial values are individually computed by each of the cores processing iterations in the same loop, and whose total value can be computed from all the partial values (computed individually) at the end of the parallel region. Reduction variables can also be addressed using OpenMP (e.g., #pragma omp parallel for reduction(+ : sum). This clause makes the reduction variable shared to generate the correct results but private to avoid race conditions from parallel execution.

Since these types of inhibitors are solvable by OpenMP, they are not considered in our empirical study in Chapter 4 and any loops that contain only these inhibitors are considered to be free for-loops. In summary, this leaves the three inhibitors (i.e., data dependency, function calls with side effects, and jumps) that are not solvable by OpenMP as the effective inhibitors to parallelization. For-loops that do not contain any of these inhibitors are free-loops and able to be parallelized with OpenMP.
class Base{
public:
    virtual const string action() { return "Default"; }
};

class Obj1: public Base{
public:
    virtual const string action() { return "predefined name"; }
};
class Obj2: public Base{
public:
    virtual const string action() {
        cout << " side effect ";
        return "user entered";
    }
};

int main(){
    Obj1 iObj1; Obj2 iObj2;
    for(int i = 0; i < 10; ++i)
        cout << iObj1.action();
    for(int i = 0; i < 10; ++i)
        cout << iObj2.action();
}

Figure 11. Example shows how virtual methods are used in C++.
CHAPTER 4

Empirically Examining the Prevalence of Inhibitors

In this chapter, we present an empirical study that examines the potential to parallelize general-purpose software systems. The study is conducted on 13 open source systems comprising over 14 MLOC. Each for-loop is statically analyzed to determine if it can be parallelized or not. A for-loop that can be parallelized is termed a free loop. Free-loop can be easily parallelized using tools such as OpenMP. For the loops that cannot be parallelized, the various inhibitors to parallelization are determined and tabulated.

The data shows that the most prevalent inhibitor by far, is functions called within for-loops that have side effects. This single inhibitor poses the greatest challenge in adapting and re-engineering systems to better utilize modern multi-core architectures. This fact is somewhat contradictory to the literature, which is primarily focused on the removal of data dependencies within loops. Results of this work also show that function calls via function pointers and virtual methods have very little impact on the for-loop parallelization process. Historical data over a ten-year period of inhibitor counts for the set of systems studied is also presented. It shows that there is little change in the potential for parallelization of loops over time.

The tools used to conduct the study are described in this chapter. This chapter describes two major contributions to the dissertation: The data shows that the most prevalent inhibitor by far, is functions called within for-loops that have side effects. This
study also demonstrates that function calls via function pointers and virtual methods have very little impact on the for-loop parallelization process.

4.1 Methodology for Detecting Parallelization Inhibitors

We now describe the methodology we used to detect the parallelization inhibitors and collect the data for our case study. Data dependency is discussed first followed by function calls with side effects, and then jump statements (e.g., break, goto). In this study, a for-loop is considered a free-loop if it does not contain any parallelization inhibitors that are not already solvable with OpenMP. That is, a free-loop does not contain any of the following inhibitors: data dependency, function calls with side effects, or jumps outside of the loop.

Techniques to determine data dependency and function calls with side effects are generally conservative and label situations as having a data dependency or a side effect when in fact there may not be one. This is referred to potential data dependency and function call with potential side effects [Petersen, Padua 1993; 1996]. The static analysis required to identify all the actual cases from the potential cases can be quite expensive. In some cases it cannot be done by static analysis and requires some form of dynamic analysis. Here we limit our detection approach to simple static analysis. Using complicated (deep) analysis or dynamic analysis would prove to time consuming to conduct the case study. Also, based on the literature [Goff, Kennedy, Tseng 1991] and our results there would be only a limited improvement of accuracy.

We developed a tool, ParaStat, to analyze loops and determine if they contain any inhibitors as defined in this section. 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 [Collard, Decker, Maletic 2011b; Collard, Kagdi, Maletic 2003; Collard 2002] to parse and analyze each file. srcML is an open source software infrastructure to support the exploration, analysis, and manipulation of source code. We use srcML because it is very efficient and allows us to construct specialized static analysis tools very easily. The srcML format wraps the statements and structures of the source-code syntax with XML elements, allowing tools, such as ParaStat, to use XML APIs and tools (e.g., XPath) to locate such things as for-loops and to analyze expressions. The srcML toolkit provides for fast translation to the srcML format at speeds of 35KLOC/second, and can convert large source-code projects to the srcML format in minutes. Once in the srcML format, ParaStat iteratively finds each for-loop and then analyzes the expressions in the for-loop to find the different inhibitors. A count of each inhibitor per loop is recorded. It also records the number of free-loops found. The final output is a report of the number of free-loops and for-loops with one or more types of inhibitors.

Now, we will discuss and describe how the ParaStat tool finds and counts each inhibitor along with any limitations of the approach.

4.1.1 Detecting data dependency

Here we take a conservative approach to detecting data dependency and detect all potential dependencies. We say there is data dependency in a for-loop if two statements refer to the same array with one statement referencing the array as a l-value and the other referencing the array as an r-value, or both are l-values. Additionally, if the same array
appears as both a l-value and a r-value in the same statement, we also include this as a potential data dependence. Examples of these situations are given in Figure 9.

ParaStat starts by separating both sides of assignment-statements. Then, all the array references found in each side are saved to a table with the array name and the subscripts of each reference. Then, the subscripts are analyzed and all the array items with a constant subscript are excluded from any further analysis (e.g., array constRef in the fourth for-loop in Figure 12) as these can be easily parallelized using OpenMP. If any array name is found on both sides of a statement (i.e., as both an l-value and r-value), or if the same array name is used more than one time on the left hand side, a data dependency is reported. The approach is applied to both single and multi-dimensional arrays and vectors, regardless of the type.

However, no sophisticated data dependency test is applied and our approach is conservative in that it detects potential data dependencies. As such, the approach will find all actual data dependencies but will count a situation as having a data dependency when one may not actually exist. Examples of this are given in Figure 12. The tool reports a data dependency between (array1[subscr] and array1[subscr + n]) regardless of the value of n, simply because the subscripts are not deeply analyzed and value of n is not dynamically evaluated. While it is clear that the value of n should be considered, because if it is greater than the value of the variable half_size, there will not be any data dependency. We feel this is a reasonable tradeoff since only simple static analysis is required. In addition, there are always situations where determining if an actual data dependency exists is computationally impractical. The approach will label some situations as having a data
dependency when none may actually exist due to the conservative approach for array access.
for (int i = 1; i < half_size; ++i)
    array1[i] = array1[i + n];

int e;
cin >> e;
for (int j = 1; j < 10000; ++j)
    array2[j] += array2[e];

int z, n = 2; std::cin >> z;
for (int k = 1; k < 10000; ++k)
    array3[k * z] = array3[(k + n) * z];

int x = 0;
for (int m = 1; m < 10000; ++m) {
    constRef[7] = x;
    x = constRef[3];
}

---

**LeftArrayNames**

array1

**LeftArrayIndices (subscripts)**

i

array2

k

array3

j*z

constRef

7

---

**RightArrayNames**

array1

**RightArrayIndices (subscripts)**

i + n

array2

e

array3

(j + n)*z

constRef

3

---

**Cases that are reported as data dependency but should not be:**

1) `array1[i]` and `array1[i + n]` → Flow-dependence
   No dependency if `n > half_size`

2) `array2[j] += array2[e]` → Flow-dependence
   No dependency since `e` is fixed in all iterations

3) `array3[j*z] = array3[(j + n)*z]` → Flow-dependence
   No dependency when `z = 0`

---

*Figure 12.* Example of the limitations of our approach used in our tool in dependence detection. The approach will label some situations as having a data dependency when none may actually exist due to the conservative approach for array access.
4.1.2 Detecting Function Calls With Side Effect

The detection of I/O operations is accomplished by identifying any calls to standard library functions (e.g., printf, fopen). A list of known I/O calls from the standard libraries of C and C++ was created. Our tool checks for any occurrence of these and if a function contains one it is labeled as having a side effect. Also, standard (library) functions can be labeled as side effect free or not. As such, a list of safe and unsafe functions is kept and our tool checks against this list to further identify side effects.

Our detection approach identifies all function/method calls within a loop. The functions directly called are located and statically analyzed for possible side effects through the chain of calls. This is done for any functions in the call graph originating from the calls in the loop. Specifically, a subset of the full call graph is constructed. This subset contains only the function calls that are involved and relevant to parallelization. That is, only functions directly called from for-loops are included in their call chain. This reduced call graph is then used to propagate any side effect detected among all callers of the function. This technique is presented in details in Section 4.1.2.1.

4.1.2.1 Call Graph

A call graph is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a procedure and each edge \((f, g)\) indicates that procedure \(f\) calls procedure \(g\). Thus, a cycle in the graph indicates recursive procedure calls.

A call graph is a basic program analysis tool that is usually used for human understanding of programs, or as a basis for further analyses, such as an analysis that tracks
the flow of values between procedures. One simple application of call graphs is finding functions that are never called. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is undecidable, so static call graph algorithms are generally over approximations. That is, every call relationship that occurs is represented in the graph, and possibly some call relationships that would never occur in actual runs of the program.

In this study, we have developed an approach that can generate a reducible call graph from srcML documents. Our tool utilizes srcML features of transforming a whole system to one XML file. The size of this file varies based on the targeted system; let us note that the size has no effect on the performance of our tool. From the XML file, we generate an adjacency matrix, which will be used later on for the side effect detection and propagation.

We have observed that not all the user defined functions are involved in the parallelization process. Two classes of functions would be considered for side effect detection analysis. The called functions within the for-loops are collected and counted first and then introduced for further analysis. All the called functions within the functions detected in the for-loops will be analyzed as well since they can affect the called functions within the for-loops indirectly (function calls other functions with side effect also has a side effect.) So by building the reducible call graph, the system resources will be better utilized and even the compilation time will be reduced.
Figure 13. High level view of the reducible call graph generation by ParaStat.

Figure 13 presents the process of generating the call graph using our tool from srcML file. The srcML file contains all the transformed source files as units in one master unit. Figure 15 shows the algorithm that generates the call graph from the srcML file. Note that the call graph is reduced. The algorithm starts by iterating through the units in the srcML file and in each unit only the for-loops are counted and retrieved as XML nodes. The units with no for-loop statement are ignored. The for-loops then iterated in order to detect any function calls within each for-loop body. The loop with no function calls is ignored and considered for other inhibitor detection processes. If a function call is found within the for-loop, it is added to the graph and the adjacency matrix is updated. All the detected calls will be inserted into a stack and then passed to the Checkstack method for further parsing. Each call detected in the body of the stack item will be added to the graph and the adjacency matrix is updated, again. The process continues until all the units are processed and the adjacency matrix is produced. Figure 16 and Figure 17 present the generated graphs that represent the functions in the source files in Figure 14.
//Starter.cpp
#include<iostream>
#include <h3.h>
using namespace std;

void func2(){ func3(); }
void func1(){ func2(); }

void main()
{
    int x;
    for (int i=0;i<=10;i++) {func1();}
    func11();
    func8();
}

//h1.h
void func12();
void func11(){ func12(); }
void func10();

//h2.h
#include<h1.h>
void func9(){ func10(); }
void func8(){ func9(); }
void func7()
    { cout<<"Side effect"; }
void func6();

//h3.h
#include<h2.h>
void func5(){func7();func12();}
void func4(){func7();func6();}
void func3(){func5();func4();}

Figure 14. Call Graph Code Sample
The reducible call graph in Figure 17 represents only the involved function in the parallelizability inspection, while Figure 16 is a complete call graph. This reducible call graph will be used later to propagate any detected side effect in any node. The next section will contain further details about the side effect detection and propagation.

4.1.2.2 Reducible Call Graph

As mentioned in Section 3.2, calling a function that contains any side effect is also effected. On the top of that, we developed a way for our tool to be able to propagate any detected side effect throughout the generated reducible call graph. The nodes first are sorted by their in-degrees in non-increasing order, so that the one with highest degree will be parsed first. If any side effect is detected, then all nodes that have edges to the effected function are annotated as effected functions as well. Additionally, this process will be applied again until the effect is propagated throughout the reducible call graph. This approach helps in saving time that can be better spent on detecting functions that are effected by the propagation. Thus, processing time is decreased tremendously in many cases especially in large scale systems where the number of functions is relatively high.

From the example in Figure 14, we can see that the function func7 in h2.h file does contain a side effect I/O operation and is the function with the highest in-degree of 2. The algorithm works starting from func7 and once the side effect is detected, it will be propagated. func5 and func4 will be updated as effected as well, and then in succession, func3, func2, func1 and finally the main function.
1: **ParseUnit** (XMLDOC )
2: For each unit in srcMLfile
3:   For each function in unit.Functions
4:      If (function.fors.count>0)
5:         For each forloop in function.fors
6:            If (forloop.calls.count>0)
7:               For each call in forloop.calls
8:                  Graphfunctions.add(call.FunctionPath)
9:                  Graph [function.pos,call.path.pos]=1
10:                 Stack.push(call.path);
11:                  Cheackstack(Stack);

1: **Cheackstack** (Stack)
2:  While (!Stack.empty)
3:    FullPath=Stack.pop()
4:    FunNode=GetNode(Fullpath)
5:    If (FunNode.Calls.Count>0)
6:       For each call in FunNode.Calls
7:          Graphfunctions.add(call.FunctionPath)
8:          Stack.push(call.path)
9:          Graph [FunNode.pos,call.pos]=1

Figure 15. Generating Graph Algorithm via Counting and Analyzing Calls within For-Loops
Figure 16. Complete (Full) call graph generated from the code presented in Figure 14.
Figure 17. Reduced Call Graph generated from the code presented in Figure 14
Figure 18. Adjacency Matrix Graph Functions List.

| C:\~\Starter.cpp\main       | 0 1 0 0 0 0 0 0 |
| C:\~\Starter.cpp\func1     | 0 0 1 0 0 0 0 0 |
| C:\~\Starter.cpp\func2     | 0 0 0 1 0 0 0 0 |
| C:\~\h3.h\func3            | 0 0 0 0 1 1 0 0 |
| C:\~\h3.h\func5            | 0 0 0 0 0 0 1 0 |
| C:\~\h3.h\func4            | 0 0 0 0 0 0 1 1 |
| C:\~\h2.h\func7            | 0 0 0 0 0 0 0 0 |
| C:\~\h2.h\func6            | 0 0 0 0 0 0 0 0 |

1: SideEffectPropagation(Graph[],GraphFunctions)
2: {
   // sorting the graph nodes decreasly based on
   // their in_degree.
   3: SortNodes(Graph[],GraphFunctions);
   4: while (UNEXPLOREDNODES.count>0)
   5:     {
   6:         node=getNextUnexploredNode();
   7:         if (isEffectd(node))
   8:             {
   9:                 node.explored(True);
 10:                 Propagate(node);
 11:             }
 12:         else
 13:             node.explored(False);
 14:     }
 15: }

Figure 19. Side Effect Detection and Propagation Algorithm
Fun6 is left so it will be parsed next. Note that our algorithm could save us the time needed for parsing the effected functions of the propagation one by one. So the function isEffected(), for all types of side effect detection with high complexity will be called just two times instead of 8 times which is considered a reasonable time savings. Figure 20 shows the reducible call graph after propagating the detected side effect. By conducting this static analysis of function calls within for-loops, we produced more accurate result than our previous study [Alnaeli, Alali, Maletic 2012a]. There, we assumed all function calls (other than standard library calls) had side effects. By conducting the additional static analysis to determine if calls actually have side effects approximately 25% fewer for-loops were found to have side effects.

Even with our analysis there could still be some functions that appear to have side effects when none actually exists or that the side effect would not be a problem for parallelization. These cases typically require knowledge of the context and problem being addressed and may require human judgment (i.e., may not be automatically determinable). However, our approach does not miss detecting any potential side effects. As such, we may over count side effects but not under count them.
Figure 20. Side Effect Propagation Approach Using Reduced Call Graph
4.1.3 Function pointers and virtual methods

Our approach for calls using function pointers and virtual methods is to assume that all carry side effects. At the onset, this may appear to be a problematic, however conservative, limitation. However, this assumption is supported by empirical analysis we undertook. We present the complete details of this data later in Section Chapter 54.3.4 and the data is summarized in
Table 6 and Table 7. In short, we found that in the 13 systems we studied, calls using function pointers or virtual methods rarely occur in for loops that do not already have an existing inhibitor. That is, if a call using a function pointer or a virtual method is used within a loop, there is almost always another inhibitor preventing the parallelization of that loop.

Hence, our claim is that assuming that all such calls have side effects has a very small impact on the actual number of loops that can be parallelized. As such, our assumption (limitation) only degrades the accuracy of our results by a small amount while avoiding an extremely large amount of analysis (which in itself has potential inaccuracies). Function pointers can come in various forms: global and local function pointers.

Global forms are further categorized into defined, class members, array of function pointers, and formal parameters [Shah Anand 1995a]. Our tool, ParaStat, detects all of these types of function pointers whenever they are present in a for-loop. Figure 21 contains examples of the detection of these types of function pointers. Pointers to member functions declared inside C++ classes are detected as well. Classes that contain at least one function pointer and instances derived from them are detected. 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.

Detecting calls to virtual methods is a fairly simple lookup. We identify all virtual methods in a class and any subsequent overrides in derived classes. We do not perform analysis on virtual methods, instead it is assumed that any call to a virtual method has a
side effect. Again, this is a conservative assumption and we will label some methods that in actuality do not have a side effect to be a problem. A slightly more accurate approach would be to analyze all variations of a virtual method and if none have side effects then it would be a safe call. However, this would require quite a lot of extra analysis with little overall improvement in accuracy. Figure 9 presents an example where ParaStat assumes that a side effect exists in the method action() in both for-loops. This is not actually correct, because the implementation of the virtual method action() in the derived class Obj1 has no side effect in the other implementation of the equivalent virtual method action() in the sibling derived class Obj2.
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 21. Examples of function pointers and what Parastat detects via the approach we use in our work.
4.2 Data Collection

As mentioned previously, we developed a tool, ParaStat, which analyzes loops and determines if they contain any inhibitors. The srcML toolkit produces an XML representation of the parse tree for the C/C++ systems we examined. ParaStat, which was developed in C#, analyzes the srcML produced using XML tools to search the parse tree information using system.xml from the .NET framework.

In C# we use XmlDocument to identify every for-loop in a system. The body of each loop is then extracted and examined for each type of inhibitor. If no inhibitors exist in a for-loop it is counted as a free loop otherwise the existence of each inhibitor is recorded. Converting the source code for the systems into srcML took between 1 and 16 minutes for an individual system on a typical desktop computer. Next, for-loops were detected and each was then analyzed in order to detect inhibitors in their bodies. The analysis phase took a little over 2 minutes for gcc and the other systems took less.
class Base{
public:
    virtual const string action(){ return "Default";}
};

class Obj1: public Base{
public:
    virtual const string action(){return "predefined name"; }
};
class Obj2: public Base{
public:
    virtual const string action(){
        cout<< " side effect ";  return "user entered";
    }
};

int main(){
    Obj1 iObj1; Obj2 iObj2;
    for(int i = 0; i < 10; ++i)
        cout << iObj1.action();

    for(int i = 0; i < 10; ++i)
        cout << iObj2.action();
}

Figure 22. Example of ParaStat limitations in determining side effects for virtual methods. ParaStat assumes both for-loops are not free loops, but the method definition shows that they are free
Table 2. The 13 open source systems used in the study.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Language</th>
<th>KLOC</th>
<th>Files</th>
<th>While Loops</th>
<th>For Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>4.5.3</td>
<td>C/C++</td>
<td>4,029</td>
<td>40,638</td>
<td>6,102</td>
<td>27,688</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>2010</td>
<td>C/C++</td>
<td>1,591</td>
<td>5,161</td>
<td>2,493</td>
<td>4,646</td>
</tr>
<tr>
<td>KOffice</td>
<td>2.3</td>
<td>C++</td>
<td>1,185</td>
<td>4,927</td>
<td>1,640</td>
<td>4,859</td>
</tr>
<tr>
<td>Subversion</td>
<td>1.6.17</td>
<td>C</td>
<td>922</td>
<td>687</td>
<td>511</td>
<td>1,475</td>
</tr>
<tr>
<td>Open MPI</td>
<td>1.4.4</td>
<td>C/C++</td>
<td>888</td>
<td>3,606</td>
<td>1,684</td>
<td>4,907</td>
</tr>
<tr>
<td>LLVM</td>
<td>2011</td>
<td>C/C++</td>
<td>736</td>
<td>1,796</td>
<td>1259</td>
<td>7,776</td>
</tr>
<tr>
<td>Python</td>
<td>2.5.6</td>
<td>C</td>
<td>695</td>
<td>1,538</td>
<td>969</td>
<td>1,896</td>
</tr>
<tr>
<td>Ruby</td>
<td>186p399</td>
<td>C</td>
<td>565</td>
<td>389</td>
<td>1,054</td>
<td>1,333</td>
</tr>
<tr>
<td>OSG</td>
<td>3.0.1</td>
<td>C++</td>
<td>503</td>
<td>1,992</td>
<td>1,311</td>
<td>5,803</td>
</tr>
<tr>
<td>QuantLib</td>
<td>1.1</td>
<td>C++</td>
<td>449</td>
<td>3,398</td>
<td>472</td>
<td>4,476</td>
</tr>
<tr>
<td>httpd</td>
<td>2.2.17</td>
<td>C</td>
<td>391</td>
<td>370</td>
<td>946</td>
<td>1,005</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>2014</td>
<td>C/C++</td>
<td>2,356</td>
<td>11436</td>
<td>3,078</td>
<td>8,787</td>
</tr>
<tr>
<td>Xapain</td>
<td>2011</td>
<td>C/C++</td>
<td>159</td>
<td>781</td>
<td>343</td>
<td>870</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>14,469</td>
<td>76,719</td>
<td>21,862</td>
<td>75,521</td>
</tr>
</tbody>
</table>
4.3 Findings of the Study

We now study the parallelizability of thirteen open-source software projects. Table 2 presents the list of systems examined along with the version, number of files, and LOCs for each. Also included is a count of how many for-loops were found in each system and for comparison the number of while loops.

These systems were chosen because they represent a variety of applications including compilers, desktop applications, libraries, a web server, and a version control system. They represent a set of general-purpose open-source applications that are widely used. These are systems that are not specifically aimed at parallel architectures but may benefit from parallel/multicore hardware. We feel that they represent a good reflection of the types of systems that would undergo reengineering or migration to better take advantage of modern hardware.

One item of interest in Table 2 is that with the exception of Httpd and Ruby, all of the systems show a much larger use of for-loops than while-loops. Also, a few of the systems utilized for-loops to a much greater degree than while-loops (e.g., gcc has almost four times as many for-loops). This gives promise for potential parallelization through the use of APIs such as OpenMP.

4.3.1 Design of the Study

Our study focuses on four aspects of for-loops. First, the percentage of for-loops containing one or more inhibitors; this gives a handle of how much of the system could be readily parallelized by a compiler or other automated tool. Second, we examine which inhibitors are most prevalent. Third, we seek to understand when inhibitors are the sole
cause in preventing parallelization. That is, loops can have multiple inhibitors preventing parallelization and therefore would require a large amount of effort to remove all the inhibitors. Thus we are interested in understanding how often only one type of inhibitor occurs in a loop. These types of loops would hopefully be easier to refactor into something that is parallelizable. Lastly, we examine how the presence of inhibitors changes 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 percentage of for-loops that are free loops (have no inhibitors)?

R2: Which types of inhibitors are the most prevalent?

R3: Which types of inhibitors are the most prevalent exclusively?

R3a: Data dependencies are a focus in the research literature. How prevalent are they as potential inhibitors?

R3b: Complex analysis is needed for function pointers/virtual methods calls. How prevalent are they as potential inhibitors?

R4: Over the history of a system, is the presence of inhibitors increasing or decreasing?

Question R3 gives rise to two sub-questions. The first concerns data dependencies as we are interested in understanding if the strong focus in the literature on data dependency is reflected in actual prevalence of this inhibitor in systems we examined. The second addresses our assumption that calls using function pointers or virtual methods are rarely the sole inhibitor of for-loops. We now examine our findings within the context of these research questions.
Table 3. Number of loops, free loops, and inhibitors for each system found using ParaStat. The percentage is over the total number of for-loops detected for each system. A loop may have more than one inhibitor, so the total may be greater than 100%.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of For-loops</th>
<th>Number of Free-loops</th>
<th>Function Call with Side Effects</th>
<th>Jumps</th>
<th>Data Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>27,688</td>
<td>15,634 (57%)</td>
<td>7,738 (28%)</td>
<td>3,563 (13%)</td>
<td>2,777 (10%)</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>4,646</td>
<td>2,900 (62%)</td>
<td>1,183 (26%)</td>
<td>606 (13%)</td>
<td>205 (4%)</td>
</tr>
<tr>
<td>KOffice</td>
<td>4,859</td>
<td>3,028 (62%)</td>
<td>1,363 (28%)</td>
<td>483 (10%)</td>
<td>170 (4%)</td>
</tr>
<tr>
<td>Subversion</td>
<td>1,475</td>
<td>367 (25%)</td>
<td>998 (68%)</td>
<td>221 (15%)</td>
<td>31 (2%)</td>
</tr>
<tr>
<td>Open MPI</td>
<td>4,907</td>
<td>2,016 (41%)</td>
<td>2,094 (43%)</td>
<td>1155 (24%)</td>
<td>528 (11%)</td>
</tr>
<tr>
<td>LLVM</td>
<td>7,776</td>
<td>4,336 (56%)</td>
<td>2,741 (35%)</td>
<td>920 (12%)</td>
<td>252 (3%)</td>
</tr>
<tr>
<td>Python</td>
<td>1,896</td>
<td>771 (41%)</td>
<td>810 (43%)</td>
<td>597 (32%)</td>
<td>143 (8%)</td>
</tr>
<tr>
<td>Ruby</td>
<td>1,333</td>
<td>518 (39%)</td>
<td>631 (47%)</td>
<td>281 (21%)</td>
<td>87 (7%)</td>
</tr>
<tr>
<td>OSG</td>
<td>5,803</td>
<td>3,904 (67%)</td>
<td>1,309 (23%)</td>
<td>323 (6%)</td>
<td>424 (7.3%)</td>
</tr>
<tr>
<td>QuantLib</td>
<td>4,476</td>
<td>2,899 (65%)</td>
<td>712 (16%)</td>
<td>184 (4%)</td>
<td>875 (20%)</td>
</tr>
<tr>
<td>httpd</td>
<td>1,005</td>
<td>575 (57%)</td>
<td>218 (22%)</td>
<td>241 (24%)</td>
<td>53 (5%)</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>8,787</td>
<td>3,607 (41%)</td>
<td>4,043 (46%)</td>
<td>1,227 (14%)</td>
<td>609 (7%)</td>
</tr>
<tr>
<td>Xapain</td>
<td>870</td>
<td>486 (59%)</td>
<td>271 (31%)</td>
<td>101 (12%)</td>
<td>67 (8%)</td>
</tr>
</tbody>
</table>
Percentage of Free-Loops

Table 3 presents the results collected for the 13 systems. We give the total number of for-loops along with the number of free-loops we detected. The percentage of free-loops is computed over the total number of for-loops. As can be seen, free loops account for between 25% and 67% of all for loops in these systems, with an overall average of 52%. That is, on average, half of all the for-loops in these systems could potentially be parallelized. This addresses R1.

4.3.2 Inhibitor Distribution

We now address R3 and R3a and present the details of our findings on the distribution of inhibitors.

Table 3 also presents the counts of each inhibitor that occur within for-loops. Many of the for-loops have multiple inhibitors (e.g., a data dependency and a jump). As can be seen, function-call inhibitors are by far the most prevalent across all systems. For most of the systems this is then followed by jumps and then data dependency, thus addressing R2. We lumped the jumps together but we note that break statements are much more prevalent than goto statement.

Table 3 gives the percentage of for-loops that contain only one type of inhibitor for each category (addressing R3). The average percentage is also given and this indicates that function-call inhibitors are clearly far more prevalent. We see that Subversion has the
largest percentage of function-call inhibitors 58%, followed by Chrome at 39%. QuantLib has the lowest, at 12%. The percentage of the for-loops that contain only a potential data dependency across all the systems is quite small by comparison. QuantLib has the large percentage of data dependencies while the smallest belongs to Subversion.

Clearly there are a number of for-loops with multiple inhibitors thus complicating the parallelization process. But no matter how we present the data, it is apparent that function-call inhibitors present the most serious roadblock to parallelization. Moreover, while there is a lot of literature devoted to addressing the problems of data dependencies (i.e., R3a), it appears that resolving that problem will have a very limited impact on the parallelization of common software applications (such as those examined in this study).

Figure 23 presents the average percentage, over all 13 systems, of for-loops that are not free and that contain one or more inhibitors. This gives a clear view of which inhibitor occurs most frequently. We see that the trend is likewise similar via this perspective. Function-call inhibitors are by far the most prevalent. On average we see the next most prevalent inhibitor is the jumps followed next by data dependency.

Additionally, in Table 5 we compare the results of our previous work [Alnaeli, Alali, Maletic 2012b] in computing function-call side effects with the work presented here. In that previous work a very conservative approach was taken and all function calls within loops were considered to have a potential side effect. Here we used the call graph to compute if a call actually has a side effect or not. The additional analysis is clearly necessary to gather accurate data on inhibitors of loops. There are approximately 25% (on average) fewer detected free loops when using the more conservative assumption.
Table 4. Percentage of for-loops in each system that contain inhibitors \textit{exclusively}; divided by each type; the remaining for-loops are either free or have more than one type of inhibitors

<table>
<thead>
<tr>
<th>System</th>
<th>Function Call with Side Effects</th>
<th>Jumps</th>
<th>Data Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>22%</td>
<td>9%</td>
<td>8%</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>21%</td>
<td>8%</td>
<td>3%</td>
</tr>
<tr>
<td>KOffice</td>
<td>25%</td>
<td>7%</td>
<td>3%</td>
</tr>
<tr>
<td>Subversion</td>
<td>58%</td>
<td>7%</td>
<td>1%</td>
</tr>
<tr>
<td>Open MPI</td>
<td>27%</td>
<td>10%</td>
<td>5%</td>
</tr>
<tr>
<td>LLVM</td>
<td>29%</td>
<td>7%</td>
<td>2%</td>
</tr>
<tr>
<td>Python</td>
<td>23%</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Ruby</td>
<td>35%</td>
<td>9%</td>
<td>4%</td>
</tr>
<tr>
<td>OSG</td>
<td>20%</td>
<td>4%</td>
<td>6%</td>
</tr>
<tr>
<td>QuantLib</td>
<td>12%</td>
<td>3%</td>
<td>16%</td>
</tr>
<tr>
<td>httpd</td>
<td>15%</td>
<td>17%</td>
<td>4%</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>39%</td>
<td>8%</td>
<td>4%</td>
</tr>
<tr>
<td>Xapain</td>
<td>26%</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>Average</td>
<td>27%</td>
<td>8%</td>
<td>5%</td>
</tr>
</tbody>
</table>
Figure 23. The average percentage of non-free-for-loops that contain at least one inhibitor of the given type, over all 13 systems. Loops may contain more than one inhibitor type.
Figure 24. Percentage of for-loops in gcc that contain only a single type of inhibitor (exclusive). The remaining for-loops are either free or have multiple types of inhibitors.
Table 5. Comparison assuming all function calls have a potential side effect as in [Alnaeli, Alali, Maletic 2012b] and doing more complete analysis to determine actual side effect. The results on the percentage of free-loops is also given for each approach. The improvement in percentage of free-loops when analysis that is more accurate is conducted is obvious.

<table>
<thead>
<tr>
<th>System</th>
<th>Functions Call with Side Effect</th>
<th>Free Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Actual (more accurate)</td>
<td>Potential</td>
</tr>
<tr>
<td>gcc</td>
<td>28%</td>
<td>48%</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>26%</td>
<td>90%</td>
</tr>
<tr>
<td>KOffice</td>
<td>38%</td>
<td>76%</td>
</tr>
<tr>
<td>Subversion</td>
<td>68%</td>
<td>77%</td>
</tr>
<tr>
<td>Open MPI</td>
<td>43%</td>
<td>49%</td>
</tr>
<tr>
<td>Python</td>
<td>43%</td>
<td>46%</td>
</tr>
<tr>
<td>Ruby</td>
<td>47%</td>
<td>52%</td>
</tr>
<tr>
<td>OSG</td>
<td>23%</td>
<td>79%</td>
</tr>
<tr>
<td>QuantLib</td>
<td>16%</td>
<td>63%</td>
</tr>
<tr>
<td>httpd</td>
<td>22%</td>
<td>50%</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>46%</td>
<td>54%</td>
</tr>
<tr>
<td>Average</td>
<td>36%</td>
<td>62%</td>
</tr>
</tbody>
</table>
Now let us examine one system, gcc, in a bit more detail. Figure 24 presents the number of for-loops with only a single type of inhibitor over the total number of for-loops in gcc. The remaining for-loops are either free-loops or have multiple types of inhibitors. This presentation of the data is useful since it addresses each type of inhibitor separately. That is, if we have a means to resolve one inhibitor it can be systematically applied to for-loops with only that type. We found that only about 8% of loops in gcc contain only data dependency inhibitors. As can be seen, the occurrence of function-call inhibitor is by far the most prevalent inhibitor (21.6%). Figure 23 presents the findings from a different perspective and gives the percentage of for-loops that contain any inhibitor. It shows that 27.9% of all for-loops contain a function-call inhibitor and 12.9% contain a data dependency.

It is interesting to note that goto-statements are used in for-loops even though it is considered a bad programming practice [Dijkstra 1979]. We did some spot inspections and found a number of cases of goto-statements being used for optimization purposes or to simplify the logic of loop conditions for exits.

4.3.3 Calls to Function Pointers/Virtual Methods

To address R3b, all thirteen systems were examined with respect to function pointer and virtual method usage. Table 6 presents the number of function pointers found in each system. This includes all definitions of function pointers as parameters, in arrays, and as global declarations [Shah Anand 1995b]. The number of calls using function pointers is also given in the table. This provides perspective on the overall usage of calls using function pointers within the examined systems. The number of such calls in these systems
varies greatly. For example, in gcc, 1219 of these calls were detected (0.1% out of total calls in the whole system). Chrome also has a significant number, 1664, of calls to function pointers/virtual methods. The remaining systems have far fewer.

The last column in Table 6 presents the number of for-loops that contain only a call function pointer or virtual method as an inhibitor. The actual number of such calls that occur in for-loops is greater. However, many of those occur in conjunction with another inhibitor. We only present the cases where calls via function pointers/virtual methods are the only inhibitor within a for-loop to understand the actual impact of function pointers as inhibitors. There are only a small percentage of for-loops that can potentially be blocked by calls via function pointers/virtual methods (1% on average for across these systems). Chrome has the largest percentage for-loops blocked by such calls, 4% followed by Open MPI at 3%. The systems gcc, subversion, and ruby have 2% and the remaining systems have 1% or less.

Virtual functions were also examined in this study. We have counted the number of virtual functions in the systems, including in the count inherited virtual functions. Table 7 presents the number of virtual functions for seven systems out of the thirteen systems that use the object-oriented aspects of C++. It also gives the number for-loops blocked by only a virtual function calls for each system. Here we see that all the systems have quite a small percentage (1% on average with the largest being 3%) of for-loops potentially blocked by virtual function calls. These two empirical results are the basis for our argument that we can safely assume that all calls involving function pointers or virtual methods have side effects. This assumption has only a very small impact on the overall potential to
parallelize a system. In the worst case, only 1% to 2% (on average) of all for-loops would not be able to be parallelized. However, given the inherent nature of function pointers it is most likely many would indeed have some side effect making the actual impact on parallelizability much less.

We now examine the data in a different perspective by looking at the historical history of the systems to determine the trends overtime of for-loop inhibitor usage.
Figure 25. Percentage of for-loops in gcc that contain at least one inhibitor and percentage of the free-for-loops in gcc. If a for-loop has two inhibitors, it counts in both categories. Inclusive means that a for-loop can have a single or multiple types of inhibitors. (An accurate analysis with respect to function side effects)
Table 6. The number of function pointers that occur in each system along with the number of for-loops that have only a function pointer call as the inhibitor. The percentage is over the total number of for-loops. For these systems the function pointer inhibitor occurs in less than 1% of the overall for loops on average.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of Function Pointers</th>
<th>For-loops with Function Pointer Inhibitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>2194</td>
<td>511</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>267</td>
<td>1</td>
</tr>
<tr>
<td>KOffice</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Subversion</td>
<td>336</td>
<td>22</td>
</tr>
<tr>
<td>Open MPI</td>
<td>1075</td>
<td>163</td>
</tr>
<tr>
<td>LLVM</td>
<td>146</td>
<td>15</td>
</tr>
<tr>
<td>Python</td>
<td>512</td>
<td>10</td>
</tr>
<tr>
<td>Ruby</td>
<td>180</td>
<td>28</td>
</tr>
<tr>
<td>OSG</td>
<td>60</td>
<td>14</td>
</tr>
<tr>
<td>QuantLib</td>
<td>164</td>
<td>23</td>
</tr>
<tr>
<td>httpd</td>
<td>66</td>
<td>11</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>5330</td>
<td>736</td>
</tr>
<tr>
<td>Xapain</td>
<td>61</td>
<td>6</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7. The number of virtual functions for each C++ system along with the number of for-loops that have only a virtual function call as the inhibitor. The percentage is over the total number of for-loops. For these systems, the virtual function-call inhibitor occurs in less than 1% of the overall for loops on average.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of Virtual Functions</th>
<th>For-loops with Virtual Function Inhibitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>32978</td>
<td>2</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>1337</td>
<td>116</td>
</tr>
<tr>
<td>KOffice</td>
<td>1141</td>
<td>128</td>
</tr>
<tr>
<td>Open MPI</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LLVM</td>
<td>1374</td>
<td>37</td>
</tr>
<tr>
<td>OSG</td>
<td>1420</td>
<td>123</td>
</tr>
<tr>
<td>QuantLib</td>
<td>166</td>
<td>10</td>
</tr>
<tr>
<td>Chrome src14</td>
<td>1168</td>
<td>11</td>
</tr>
<tr>
<td>Xapain</td>
<td>342</td>
<td>1</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3.4 Historical Change of Inhibitor Frequency

Each of the systems, with the exception of *Chrome*, has been under development for 10 years or more. To address R4 we examined the most recent 10-year period those 12 systems. *Chrome* is a relatively new project and has a very short version history; as such it was excluded from this comparison. Our goal is to uncover how each system evolves in the context of potential parallelizability. Here we measure this by examining the change of inhibitors within loops. Our feeling is that this information could lead to recommendations for utilizing and adapting to the current multicore processing trends.

The change in the number of for-loops, free-loops, and presences of each inhibitor was computed for each version in the same manner as we described in the previous sections. These values were aggregated for each year so the systems could be compared on a yearly basis. The systems were 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 for-loops were then extracted.

Figure 26 presents the change in the percentage of free-loops for each of the 12 systems. During the 10-year period, all systems show a fairly flat trend. Two systems, *gcc* and *Ruby*, have a steep decline early on and then are relatively flat in proceeding years. Figure 27 presents the percentage of for-loops that contain a function-call inhibitor. It is approximately a mirror of Figure 26. That is, the systems that increased in the number of free-loops decreased in the number of function call-inhibitors (e.g., gcc, Ruby).
Figure 26. The evolution of the percentage of free-loops over a ten-year period for the ten systems.
Figure 27. The percentage of function-call inhibitors of for-loops over a ten-year period for the ten systems.
As can be observed in Figure 27, gcc shows a large variation in function-call inhibitors during the observed ten-year time period, with a particular large jump from 2002 to 2003. Figure 28 presents all categories of inhibitors for only gcc during this time period (broken down by release), and shows that the change in function-call inhibitors does not correlate to changes in the other inhibitors, which remained stable during these releases.

To better understand this issue, we took a closer look at the versions of gcc involved in this large increase in function-call inhibitors. The gcc version for 2002 was gcc 3.2.1, and the gcc version for 2003 was gcc 3.3.2. A comparison of these two gcc versions shows that the total number of functions in the system increased from 21,758 to 34,588: an almost 55% increase. However, the number of functions with side effects was close to double, from 10,554 to 20,645 functions with side effects.

Further examination of the individual causes for the function-call inhibitor showed a doubling in most cases. The number of function-call inhibitors due to pointers to parameters increased 118% (from 4,552 to 9,945), modifications to global variables increased 82% (from 4,776 to 8,728), and calls to other functions with side effect increased 80% (from 50,706 to 91,522). The only category that did not increase was due to I/O operations, which remained at 1,612 in both versions. Much of this was due to an apparent large reengineering of the system to rely on a global data structure (most likely for efficiency purposes).
Figure 28. The percentage of different inhibitors of for-loops for releases during the time period of 2001 – 2003 for gcc. Note that while for-loops with function-call inhibitors increased greatly, other inhibitors do not show any increase for gcc.
4.4 Discussion

The first research question (R1) addresses the percentage of free-loops within the systems studied. On average, approximately half of all for-loops in these systems could be parallelized (with the accurate function analysis for actual side effects). This means there could be a substantial increase in performance for these systems. Additionally, overall utilization resources on multicore hardware would be increased. The implication here is that parallelization of general-purpose applications may be worth the cost and effort. However, this is dependent on many factors including the bounds of the loops and the types of computations taking place in these free-loops.

While using more accurate analysis methods would most like identify a small number of additional free-loops, the costs of applying that analysis over an entire system would normally be prohibitive. One approach to address this would be to identify complex situations where deeper analysis has a good chance to have a positive outcome.

The next two research questions (R2 and R3) addressed the makeup and distribution of inhibitors within for-loops. We found that the most prevalent inhibitor is function calls with side effects. This is an important finding because while the literature has heavily focused on solving and detecting data dependencies [Maydan, Hennessy, Lam 1991; Petersen, Padua 1993; Petersen 1991], the empirical findings show that function calls within loops are the greatest roadblock to parallelization of general purpose applications. In fact, we see that (Figure 23) the vast majority of for-loops with one or more inhibitor contain a call to a function with a side effect (70%), followed by a jump (31%), and lastly data dependency (16%). Using more accurate analysis methods would in all likelihood
reduce the number (by a small amount) of both data dependencies and function calls with side effects. However, this would not impact the overall magnitude of the findings.

It appears that developing methods to remove break and goto statements from for-loops could potentially have a greater impact on improving parallelizability than addressing data dependency alone. Minimally, this implies that tools and techniques for automating the parallelization of general-purpose applications must be focused on addressing function-call inhibitors. Identifying functions that have side effects is critical knowledge for development teams with the goal of optimizing systems for automated parallelization. Coding practices aimed at avoiding the common inhibitors, as found in this study, can also be developed. Making developers better aware, via documentation or automated methods, of functions with side effects could also lead to more parallelizable code. There are few pedagogical approaches that highlight these types of coding techniques and few documented approaches to decrease inhibitors.

Our study also shows that developers are still using goto statements even while it has been considered a harmful statement [Dijkstra 1979]. For example, about 13.7% of the for-loops in Python contain a goto statement. Open MPI and Ruby have 7% and 5%, respectively. While there are appropriate times that a careful use of goto can be practical, they remove an opportunity for parallelizing those for-loops.

Research question R3b is focused on calls via function pointers and virtual methods. This is a special case of the function call with side effect inhibitor. Our interest in this special case has to do with the complexity of analysis. Our findings have an important implication to the problem of parallelizing general-purpose applications. We
found that while function pointers and virtual methods pose a serious problem to many static analysis problems, they do not represent a roadblock for the parallelization of the systems studied. In fact, for the much narrower problem of parallelizing for-loops, calls using function pointers/virtual methods are involved in a very small percentage (2%) of potentially inhibited loops. While some of the loops with calls to function pointers/virtual methods maybe parallelizable, it is clearly not worth the costs incurred from the additional complex analysis to sort them out.

Our last research question (R4) addresses the prevalence of inhibitors over the history of a system. In short, we wanted to know if the percentage of inhibitors are increasing or decreasing. On average we found a small decrease in the number of free loops along with a corresponding increase in the number of inhibitors (i.e., function calls with side effects).

Thus, we can surmise that developers do not spend much effort trying to improve the parallelizability of a system. The fact that development teams do not focus on improving parallelizability is particularly telling in
Table 8. None of the systems’ history demonstrates any systematic decrease in the number of inhibitors overtime. Even though systems such as gcc implemented features for automatically optimize code for parallelization [Novillo 2006], there is little evidence that the development team take direct advantage of using these features within the gcc code base.

Analysis methods can be used to document functions and methods that have any type of side effect. Approaches such as labeling methods with stereotypes [Dragan, Collard, Maletic 2009b] is one example. Methods that are access-only (i.e., get or predicate methods), have no side effects and as such are not inhibitors. This type of preprocessing and labeling could be of great use for compilers, as they typically avoid analyzing functions called within the bodies of for-loops that are considered for automatic parallelization. Upfront function analysis could greatly decrease parallelization development time.

Refactorings could be developed to deal with break and goto statements. Systematic removal of these statements manually or through automated tools is most likely the only practical approach to avoiding these inhibitors. Coding standards and idioms also need to be developed to avoid inhibitors. As mentioned earlier, in our hypothesis we believe that the tendency to assume that function pointers have a significant impact on the parallelization process is not realistic. In other words, the frequent usage of function pointers in the systems may not have a big impact on the for-loop parallelization in the system and thus can be excluded in developing any automatic parallelization technique to save time and reduce complexity.
Table 8. AVERAGE INCREASE/DECREASE FROM 2001 TO 2011 OF FREE LOOPS AND INHIBITORS IN THE 12 SYSTEMS. THE CHROME SYSTEMS IS LEFT OUT DUE TO ITS MUCH SHORTER HISTORY.

<table>
<thead>
<tr>
<th>System</th>
<th>Number of Free-loops Change</th>
<th>Inhibitor Change from 2001 to 2011 (+/-) %</th>
<th>Function Call with Side Effect</th>
<th>Jumps</th>
<th>Data Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>break</td>
<td>goto</td>
</tr>
<tr>
<td>gcc</td>
<td>-21%</td>
<td>+28%</td>
<td>-5%</td>
<td>-1%</td>
<td>+3%</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>-10%</td>
<td>+14%</td>
<td>+2%</td>
<td>+1%</td>
<td>-3%</td>
</tr>
<tr>
<td>KOffice</td>
<td>-10%</td>
<td>+11%</td>
<td>-1%</td>
<td>0%</td>
<td>-2%</td>
</tr>
<tr>
<td>Subversion</td>
<td>-10%</td>
<td>+13%</td>
<td>-6%</td>
<td>+1%</td>
<td>-4%</td>
</tr>
<tr>
<td>Open MPI</td>
<td>-7%</td>
<td>+8%</td>
<td>+3%</td>
<td>+4%</td>
<td>+4%</td>
</tr>
<tr>
<td>LLVM</td>
<td>-5%</td>
<td>+7%</td>
<td>-2%</td>
<td>+1%</td>
<td>+2%</td>
</tr>
<tr>
<td>Python</td>
<td>-3%</td>
<td>-3%</td>
<td>+4%</td>
<td>+4%</td>
<td>+2%</td>
</tr>
<tr>
<td>Ruby</td>
<td>-32%</td>
<td>+44%</td>
<td>-3%</td>
<td>-1%</td>
<td>+2%</td>
</tr>
<tr>
<td>OSG</td>
<td>0%</td>
<td>+9%</td>
<td>-4%</td>
<td>0%</td>
<td>-7%</td>
</tr>
<tr>
<td>QuantLib</td>
<td>-14%</td>
<td>+15%</td>
<td>+4%</td>
<td>-1%</td>
<td>+2%</td>
</tr>
<tr>
<td>httpd</td>
<td>-5%</td>
<td>+4%</td>
<td>+3%</td>
<td>-3%</td>
<td>-1%</td>
</tr>
<tr>
<td>Xapain</td>
<td>+10%</td>
<td>-7%</td>
<td>+2%</td>
<td>-5%</td>
<td>+6%</td>
</tr>
<tr>
<td>Average</td>
<td>-9%</td>
<td>12%</td>
<td>0%</td>
<td>1%</td>
<td>0%</td>
</tr>
</tbody>
</table>

4.5 Threats to Validity

We use the potential for data dependency rather than exact data dependency. Calculating a more accurate picture of data dependency can involve costly analysis. However, our approach is more conservative and as such will tend to over count data dependency rather than under count it. As such, the actual percentages of free for-loops will most likely be lower. This does not change our findings or observations in any substantial manner.

We excluded standard C functions that are known to have no side effects. Otherwise, we use an algorithm that detects function calls that have clear side effects. However, if the function was not proved to be clear of side effects we consider it as a
potential side effect holder that makes our approach conservative and therefore safe. This is of concern to the findings and observations. We have seen in previous studies [Dragan, Collard, Maletic 2009b; Dragan, Collard, Maletic 2010] that the majority of functions appear to have side effects of one type or another.

The tools we developed for this study only work with any language supported by srcML (C/C++/Java). This has restricted us from using some existing benchmarks for parallelizability (e.g., Perfect Club Benchmark) or projects written in languages such as FORTRAN. It may be that certain computationally intensive applications have a much larger prevalence of data dependency. We attempted to offset this issue by including projects such as OSG and Open MPI. However, our focus was on the application of parallelization and use of multicore architecture for a wide variety of general-purpose applications.

Upon examination of the for-loops in the study we found that some of them were part of dead code, i.e., code that would never be executed. As part of the static analysis there was no distinction made in the study between for-loops in dead code or active code that might affect the accuracy of the results we present in terms of the systems parallelizability. In the future we are planning to refine the percentages to only include active code.

4.6 Conclusion

This study empirically examined the potential parallelizability of thirteen open source software systems. The systems are all general-purpose applications not written specifically for parallel architectures. There are no other studies of this type currently in
the literature. We found that the greatest inhibitor to automated parallelization of for-loops is the presence of function calls with side effects. This is somewhat contrary to the published literature on methods for parallelizability. That is, the vast majority of literature focuses on resolving the issue of data dependency inhibitors rather than function-call inhibitors. As such, more attention needs to be placed on dealing with function-call inhibitors if a large amount of parallelization is to occur in general purpose software systems so they can take better advantage of modern multicore hardware. While we cannot completely generalize this finding to all software systems (across all domains) there is some indication that this is a common trend.

Most development teams and organizations have not focused on developing software in a way that could one day take advantage of parallel architectures. However, the recent ubiquity of multicore processors gives rise to the need to educate developers and make them more aware of the problems and inhibitors to automatically parallelizing their code. Coding style can play a big role in advancing a system’s parallelizability. The software engineering community needs to develop standards and idioms that help developers in avoiding the inhibitors and these standards should most likely be based on the nature of the API’s (e.g., OpenMP, PThread, MPI) used for parallelizing the code.

The objective of this study was to better understand what obstacles are in place for advancing the reengineering of systems to better take advantage of parallelization through code transformation (refactorings), at the compiler level, and by programmers (coding standards). We are particularly interested in tools that assist developers in an automated or semi-automated manner to refactor or transform parts of a code base to facilitate
parallelization by other tools (i.e., compilers). From the results of this work we are developing methods to assist in removing break and goto statements along with the identification of function with detrimental side effects (in the context of parallelization).

We also have empirically proven that indirect calls conducted by virtual methods and function pointers have no huge impact on the parallelization process based on the current distribution of inhibitors. That is, there are only a small percentage of for-loops that can potentially be blocked by calls via function pointers/virtual methods (1% on average for across these systems). However, this situation may not continue to be practical and valid if refactoring tasks were conducted to remove some of the inhibitors in the studied system so that for-loops that have calls through pointers/virtual methods can potentially increase.

Thus, we were motivated to conduct further investigation and solutions proposal which as a second phase. Additionally, we decided to conduct a deeper study that empirically examines the prevalence and distribution of calls using function pointers and virtual methods in general-purpose software systems. This study is presented in the next chapter.
CHAPTER 5

Prevalence of Function Pointer and Virtual Method Calls

In this chapter, we present an empirical study that examines the prevalence and distribution of calls using function pointers and virtual methods in general-purpose software systems is presented. The study is conducted on 12 open source systems comprising over 12 MLOC. Each system is analyzed and the number of function pointer and virtual method calls is determined. Additionally, function pointers are categorized based on their types and the complexity they pose in conducting inter-procedural static analysis. The results show that in a majority of the time 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 these systems shows that there is an increase in the usage of both calls using function pointers and virtual method over the lifetime the systems, thus posing further problems for inter-procedural analysis.

5.1 Introduction

It is very challenging to statically analyze programs that make calls using function pointers [Anand Shah 1995; Cheng, Hwu 2000a] or virtual methods [Bacon, Sweeney 1996b]. 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. Thus,
determining which specific function/method is actually called can only be done at run time. For the purposes of static analysis, an imprecise, but still valid, approach is to resolve all function pointers/virtual methods in the system and then assume that a call to these reach all possible candidate functions/methods in the program. This, of course, adds more complexity and inaccuracy to static analysis. In general, the problem is shown to be NP-hard based on the ways function pointers are declared and manipulated in a system [Anand Shah 1995; Cheng, Hwu 2000a]. For accurate inter-procedural analysis, function-alias analysis is a very important step that should be seriously considered [Anand Shah 1995; Emami, Ghiya, Hendren 1994b].

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 have safe analysis 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 [Alnaeli, Alali, Maletic 2012a; Bacon, Sweeney 1996b].

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. Hence, for some problems in order to preserve the expected accuracy of the results, it is not practical to use any approximation that attempts to avoid an extremely large amount of analysis. Function pointers can typically come in the form of local and global pointers. Global pointers are
further categorized into defined, class members, array of function pointers, and formal parameters [Anand Shah 1995].

Additionally, programs written in object-oriented languages such as C++ or Java may also have challenges in inter-procedural analysis. That is, static analysis is even more difficult because of object inheritance and function overloading through methods [Cheng, Hwu 2000a]. Virtual methods in an object oriented programming languages can be overridden by derived classes [Stroustrup 1987]. For adequate analysis, virtual method calls need to be resolved, and the literature is rich with algorithms for this purpose [Bacon, Sweeney 1996b; Calder, Grunwald 1994; Dean, Grove, Chambers 1995; Fern, #225, ndez 1995; Grove, DeFouw, Dean, Chambers 1997; Sundaresan et al. 2000].

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 [Bacon, Sweeney 1996b]. The complexity of class hierarchy can play a big role in the accuracy and difficulty of determining the targets of a virtual method call.

Based on the ways function pointers are used in the system and the extensive use of virtual methods, it is most likely that the analysis complexity and difficulty can be expected and determined.

As we previously presented in Chapter 4, calls via function pointers/virtual methods block only a small percentage of for loops (1% on average across these systems). However, this situation is changing overtime. In the future, if refactoring and adaptive maintenance
tasks remove some of the inhibitors in the studied system (e.g., goto statements) then for-loops that are blocked by calls through pointers/virtual methods will potentially increase

Thus, this motivated us to conduct a deeper study that empirically examines the prevalence and distribution of calls using function pointers and virtual methods in general-purpose software systems.

To the best of our knowledge, no historical study have been conducted on the evolution of function pointer and virtual method on 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 its obstacles that must be considered when static analysis is conducted.

In this study we examine 12 large-scale open source software systems from different domains. The history of each system is examined based on multiple metrics. The number of function pointers, virtual methods, and indirect calls are determined for each release. Then classification of function pointer types is determined (Global, local, Formal parameter, Class member, and array of function pointers). This data is presented and a comparison of the different systems uncovers trends and general observations. The analysis shows if there is a trend toward increasing or decreasing growth on the number of function pointer type and virtual methods.

In short, our findings show that function pointers that make the complexity of static analysis’ NP-hard, represent the vast majority of function-pointer types and thus pose a great deal of difficulty in any proposed inter-procedural analysis used for software-engineering processes (e.g., transformation, slicing, call graph construction, and automatic
parallelization). Results of this paper also show that virtual methods (calls and implementations) are still increasing overtime in most of the studied systems written with C++. For some systems designed using an OOP style, the analysis show that function pointers are still used in addition to virtual methods. The historical data analysis conducted shows that there is an increase in the usage of both function pointers and virtual methods. Thus, overtime the usage of indirect function calls leads to difficulties and hardness of inter-procedural static analysis.

The chapter is organized as follows. Background and related work is presented in section 5.2. Section 5.45.3 presents function pointer types and then followed by Section 5.4 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 5.5. A case study we designed and conducted is presented in Section 5.6 and results and findings are presented in section 5.7. 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 5.8. Sections 5.9 and 5.10 are discussions about our findings and limitations in this study, and the work is finally concluded in 5.11.

5.2 Related Work

There are multiple 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 overtime for better understanding and uncovering any trends or evolutionary patterns. That
is, we believe that can be a valuable information in determining and predicting the solutions and effort required to better statically analyze those systems written in C/C++ languages.

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 [Bacon, Sweeney 1996b; Calder, Grunwald 1994; Dean, Grove, Chambers 1995; Fern, #225, ndez 1995; Grove, DeFouw, Dean, Chambers 1997; Sundaresan et al. 2000]. However, no study has been conducted on the evolution of the open source systems overtime in the inter-procedural analysis context in the presence of function pointers and virtual methods. So, in this work, we are conducting a study on the history of each system for each release.

Ben-Chung et al. [Cheng, Hwu 2000a] 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 are need to be analyzed and resolved for correct results.

Ryder et al [Anand Shah 1995] 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.

A new technique is presented by Sundaresan et al. [Sundaresan et al. 2000] to
estimate the possible types of receivers for virtual method and interface calls in Java. Their
design goal was to develop a competitive technique that requires only one iteration, and
thus scales linearly with the size of the program, while at the same time providing more
accurate results than two popular linear techniques, class-hierarchy analysis and rapid-type
analysis [Bacon, Sweeney 1996b; Dean, Grove, Chambers 1995]. They observed that the
extra call sites resolved by variable-type analysis account for a significant number of calls
in the dynamic trace, and demonstrated that inline calls could make use of these extra call
sites giving performance improvement for two benchmarks. Their results were better than
what was achieved by rapid type analysis.

Here, we do not propose specific solution to the pointer-alias, rather, we empirically
examine a number of systems to determine what roadblocks exist to develop automated
tools for better static and inter-procedural analysis. Additionally, we show how those
systems evolve overtime in terms of expected difficulty of analysis posed by function
pointer/virtual method usage.
5.3 Function Pointers

It has been shown [Anand Shah 1995] that the complexity of inter-procedural analysis is negatively affected by the ways function pointers are declared and used in software. For instance, the broader the scope of a function pointer (e.g., global in the worst case) within the system, the complexity is higher (NP-hard in many cases) in conducting static analysis. In this section, we discuss the different types of function pointers.

The different types of function pointers that cause in NP-hard analysis are presented in the following sections [Muth, Debray 1997; Zhang, Ryder 1994a].
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 29. Examples of function pointers that cause analysis to be NP-Hard, and are detected by the tool VirFptrStat
5.3.1 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 29 in line 2. These include external function pointers, e.g., the function line 1 in Figure 29. Global function pointers are one of the types that if used, causes in NP-hard analysis [Anand Shah 1995].

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 systems in order to discover how they evolve overtime in terms of this type usage and distribution.

5.3.2 Array of Function Pointers

Array of function pointers are usually used in general purpose software systems especially the systems developed in the language C. It is also known amongst function pointer types that cause in NP-hard analysis as well [Anand Shah 1995]. An example is shown in Figure 29 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 arrays usage in the open source systems we targeted in this study.

5.3.3 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 are studied which includes the calls done through function pointers that are structure fields. Examples of this are shown in Figure 29 in line 16 as a field in the local structure srct.
is also known amongst function pointer types cause in NP-hard analysis as well [Anand Shah 1995] and it is commonly used with C/C++ code.

5.3.4 Function Pointer Class Members

Function pointers 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 somehow similar to the structure members in their nature 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 [Anand Shah 1995; Bennett, Rajlich, 73-87]. Lines 7, 6 in Figure 29 are examples of this type.

5.3.5 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.

5.4 Virtual Methods

Calls to virtual methods increase the complexity of conducting static analysis in a similar manner as calls to function pointers [Pande, Ryder 1996] [Aigner, Holzle 1996;
Bacon, Sweeney 1996b; Calder, Grunwald 1994; Pande, Ryder 1994]. 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) [Aigner, Holzle 1996; Bacon, Sweeney 1996b; Calder, Grunwald 1994]. In some cases, the analysis may not even be possible since the target of a call is unknown at compile time.

Figure 30 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 overtime is also presented in Section 5.8.
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 userEntry;
        cout << " enter a name > ";
        cin >> userEntry;
        ob_Name = userEntry;
        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 30 Virtual Method examples as declared in classes and its inherited classes detected by VirFptrStat
5.5 Detection of Function Pointers and Virtual Methods

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

In this study, function pointers are divided to two categorized. The first category type are known to cause in NP-hard analysis [Anand Shah 1995] (e.g. Global function pointers). That is, 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 and they are known to cause in NP-hard analysis[Aigner, Holzle 1996; Calder, Grunwald 1994; Pande, Ryder 1996]. Virtual methods are identified using their declarative directive virtual. Our tool does not miss any virtual methods, resulting in both complete precision and recall.

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 [Collard, Decker,
Maletic 2011a; Collard, Kagdi, Maletic 2003; Collard, Maletic, Marcus 2002] 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 ElementTree API [Behnel 2014]. That is, it has a similar implementation of 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.

*srcML* is an open source software infrastructure to support the exploration, analysis, and manipulation of source code. We use *srcML* because it is very efficient and allows us to construct specialized static analysis tools very easily. The *srcML* format wraps the statements and structures of the source-code syntax with XML elements, allowing tools, such as *VirFptrStat*, to use XML APIs and tools (e.g., XPath) to locate such things as function pointers and calls and to analyze expressions. The *srcML* toolkit provides for fast translation to the *srcML* format at speeds of 35KLOC/second, and can convert large source-code projects to the *srcML* format in minutes.

Once in the *srcML* format, *VirFptrStat* iteratively finds each virtual method and function pointer 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 each analyzed system.

Our tool, VirFptrStat, detects all types of function pointers whenever they are present in the code. Figure 29 contains examples of the detection of these types of function pointers. Pointers to member functions declared inside C++ classes are detected as well. Classes that contain at least one function pointer. 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.

Now, we will discuss each of the different function pointer types in details. We also describe how we find and counts each function pointer and virtual method along with limitations of our approach.
Figure 31. Average of percentage distribution of function pointer types statically declared in all 12 systems
5.6 Case Study

As mentioned previously, we developed a tool, VirFptrStat, which analyzes source code and determines all types of aforementioned function-pointer types as well as virtual methods used in the studied systems. The purpose of the case study was to quantify the occurrence of function pointers and virtual methods function calls that cause analysis to be NP-Hard. The systems that were chosen in this study were carefully selected to represent a variety of applications including compilers, desktop applications, libraries, a web server, and a version-control system. They represent a set of general-purpose open-source applications that are widely used. These are systems that are well-known to both academia and research communities, and have been used for software-engineering research purposes and themselves are targets of self-analysis for evolution processes expected in software systems. That is, they benefit from static analysis processes.

We feel that they represent a good reflection of the types of systems that would undergo reengineering or migration to better take advantage of available techniques and subject to adaptive changes demand analysis. The first step was to produce the srcML format of these systems. Converting the source code for the systems into srcML took between 1 and 16 minutes for an individual system on a typical desktop computer. Next, methods were detected and each was then analyzed in order to detect the ones that are declared as virtual methods. The analysis phase took a little over 3 minutes for the system gcc, while other systems took less time.


5.7 Results

We now study the presence, usage, and the distribution of function pointers types and virtual methods of 12 open-source software projects. Usage means the calls conducted via function pointers and virtual methods. Table 9 presents the list of systems examined along with the version, number of files, and LOCs for each.

Note that in Table 9 some systems (i.e., Subversion, Python, Ruby, and httpd) were developed in the C language, with no classes, let alone virtual methods. However, some of the C++ systems use function pointers in addition to virtual methods (e.g., gcc). Some information about the distribution of all function pointer types that are statically declared in the studied system as well as number of virtual methods in the are presented in Figure 31 and Table 10 respectively.

Our study focuses on three aspects regarding function pointer and virtual methods. The first aspect we focus on is the usage of function pointers and virtual methods and their distribution amongst the studied systems. 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 systems when static analysis is conducted. That is, if a system uses the function pointer types that cause in NP-hard analysis more frequently than the other types that do not cause NP-hard analysis, then the indication is that the system is potentially difficult to be statically analyzed by a compiler or other automated tools. Second, we examine which
function-pointer type is most prevalent amongst types that cause NP-hard analysis. 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 the function calls that are conducted via virtual methods and function pointer types that cause NP-hard analysis in open source systems?

**R2:** Which function pointer types amongst those that cause NP-hard analysis are the most prevalent?

**R3:** Over the history of a system, is the presence of virtual methods and function pointers that cause NP-hard analysis increasing or decreasing?

Question R3 concerns the growth of virtual methods and function pointers that cause NP-hard analysis as we believe that they pose difficulty 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.
Table 9. The 12 open source systems used in the study.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Language</th>
<th>KLOC</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>4.5.3</td>
<td>C/C++</td>
<td>4,029</td>
<td>40,638</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>2010</td>
<td>C/C++</td>
<td>1,591</td>
<td>5,161</td>
</tr>
<tr>
<td>KOffice</td>
<td>2.3</td>
<td>C++</td>
<td>1,185</td>
<td>4,927</td>
</tr>
<tr>
<td>Subversion</td>
<td>1.6.17</td>
<td>C</td>
<td>922</td>
<td>687</td>
</tr>
<tr>
<td>Open MPI</td>
<td>1.4.4</td>
<td>C/C++</td>
<td>888</td>
<td>3,606</td>
</tr>
<tr>
<td>LLVM</td>
<td>2011</td>
<td>C/C++</td>
<td>736</td>
<td>1,796</td>
</tr>
<tr>
<td>Python</td>
<td>2.5.6</td>
<td>C</td>
<td>695</td>
<td>1,538</td>
</tr>
<tr>
<td>Ruby</td>
<td>186p399</td>
<td>C</td>
<td>565</td>
<td>389</td>
</tr>
<tr>
<td>OSG</td>
<td>3.0.1</td>
<td>C++</td>
<td>503</td>
<td>1,992</td>
</tr>
<tr>
<td>QuantLib</td>
<td>1.1</td>
<td>C++</td>
<td>449</td>
<td>3,398</td>
</tr>
<tr>
<td>httpd</td>
<td>2.2.17</td>
<td>C</td>
<td>391</td>
<td>370</td>
</tr>
<tr>
<td>Xapain</td>
<td>2011</td>
<td>C/C++</td>
<td>159</td>
<td>781</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td>14,469</td>
<td>76,719</td>
</tr>
</tbody>
</table>
Figure 32. Average percentage distribution of function pointer types that are statically declared in all 12 systems.
Table 10. Number of detected virtual methods declared in subsystem that have C++ files for a ten-year period

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>410</td>
<td>412</td>
<td>406</td>
<td>418</td>
<td>457</td>
<td>429</td>
<td>33,015</td>
<td>33,180</td>
<td>32,778</td>
<td>32,878</td>
<td>32,978</td>
</tr>
<tr>
<td>OSG</td>
<td>166</td>
<td>392</td>
<td>508</td>
<td>599</td>
<td>644</td>
<td>771</td>
<td>779</td>
<td>893</td>
<td>1,012</td>
<td>1,086</td>
<td>1,124</td>
</tr>
<tr>
<td>Quantlip</td>
<td>135</td>
<td>166</td>
<td>198</td>
<td>234</td>
<td>300</td>
<td>377</td>
<td>767</td>
<td>931</td>
<td>1,103</td>
<td>1,118</td>
<td>1,187</td>
</tr>
<tr>
<td>KOffice</td>
<td>2,547</td>
<td>3,662</td>
<td>3,958</td>
<td>4,524</td>
<td>5,513</td>
<td>6,389</td>
<td>5,829</td>
<td>6,264</td>
<td>6,403</td>
<td>6,244</td>
<td></td>
</tr>
<tr>
<td>XAPAIN</td>
<td>133</td>
<td>135</td>
<td>151</td>
<td>136</td>
<td>135</td>
<td>112</td>
<td>399</td>
<td>427</td>
<td>463</td>
<td>461</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>243</td>
<td>416</td>
<td>378</td>
<td>577</td>
<td>687</td>
<td>711</td>
<td>821</td>
<td>1,115</td>
<td>1,526</td>
<td>2,047</td>
<td>2,459</td>
</tr>
</tbody>
</table>
5.7.1 Number and Percentage of Calls

The results collected for the 12 systems are presented in
Table 11. We give the number of function pointer calls along with the number of virtual method calls. Figure 33 shows the average percentage of function pointers that cause NP-hard analysis computed over the total number of detected function calls. As can be seen, the overall average of detected function pointers that cause NP-hard analysis is 96%. That is, on average, most of the function pointer calls in these systems 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 the systems, including in the count inherited virtual functions.

Table 13 presents the number of virtual function calls for seven systems out of the twelve systems that use the object-oriented aspects of C++. Here we see that four out of seven systems have a larger number of virtual method calls than function pointer calls. As an example, Koffice has 10,845 function calls conducted by virtual methods, whereas, just 11 function calls via function pointers. Likewise, KDElibs has 10,555 virtual method calls vs. 1,222 function calls via function pointers. LLVM and OSG have more frequent virtual method calls than function pointer calls as well. This also answers R1 as well.

All the systems have quite a small percentage (1% on average with the largest being 3%) of for-loops potentially blocked by virtual function calls. These two empirical results are the basis for our argument that we can safely assume that all calls involving function pointers or virtual methods have side effects. This assumption has only a very small impact on the overall potential to parallelize a system. In the worst case, only 1% to 2% (on
average) of all for-loops would not be able to be parallelized. However, given the inherent nature of function pointers it is most likely many would indeed have some side effect making the actual impact on parallelizability much less.

We now examine the data in a different perspective by looking at the historical history of the systems to determine the trends over time of for-loop inhibitor usage.
Figure 33. Average percentage of calls via function pointer type that cause np-hard analysis vs. others
### Table 11. Number of virtual method and function pointer calls

<table>
<thead>
<tr>
<th>System</th>
<th>Number Of Function Pointer Calls</th>
<th>Number Of Virtual Method Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>4131</td>
<td>328</td>
</tr>
<tr>
<td>KDELIBS</td>
<td>1222</td>
<td>10555</td>
</tr>
<tr>
<td>KOffice</td>
<td>11</td>
<td>10845</td>
</tr>
<tr>
<td>Subversion</td>
<td>908</td>
<td>-</td>
</tr>
<tr>
<td>Open MPI</td>
<td>7462</td>
<td>-</td>
</tr>
<tr>
<td>LLVM</td>
<td>308</td>
<td>715</td>
</tr>
<tr>
<td>Python</td>
<td>871</td>
<td>-</td>
</tr>
<tr>
<td>Ruby</td>
<td>814</td>
<td>-</td>
</tr>
<tr>
<td>OSG</td>
<td>101</td>
<td>550</td>
</tr>
<tr>
<td>QuantLib</td>
<td>323</td>
<td>123</td>
</tr>
<tr>
<td>httpd</td>
<td>227</td>
<td>-</td>
</tr>
<tr>
<td>Xapain</td>
<td>73</td>
<td>26</td>
</tr>
</tbody>
</table>
5.7.2 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 34 presents the average percentage of each function pointer type’s usage (calls) that occur in all studied systems. Many of the studied systems have multiple function pointer types. As can be seen, global function pointers are by far the most prevalent across all systems, at 70% for most of the systems. Then, function pointers follow it as formal parameters at 11.9%, and then data function pointers as class members at 10.6% followed by function pointers that are structure fields, thus addressing R2.

Additionally, we lumped the function pointer types that cause NP-hard analysis in order to show the presence of function pointer types that cause NP-hard analysis versus other function pointer types, and results are presented in Figure 33. We see that most of the studied systems have a larger average percentage of function pointer types that cause NP-hard analysis, at 96%, than other types 4%. Figure 35 shows the percentage of function pointer types that cause in NP-hard analysis for each system independently (addressing R2 as well). The figure indicates that for ten systems, the global function pointer type is prevalent. However, function calls that are done using function pointers that are declared as formal parameters are more prevalent in only one OSG. Additionally, function pointer class members are most prevalent in KOffice. For function calls that are conducted by global function pointers, we see that OpenMPI has the largest percentage at 99%, followed by Httpd at about 97%. OSG has the lowest, at 2%. On the other hand, OSG has the largest percentage of Calls through formal parameter function pointers at 72%, followed by
KOffice at 10%. Lowest percentage was owned by KDElibs with less than 1%. The percentage of the array of function pointers across all the systems is quite small in comparison with the other types.
Figure 34. Distribution of average of percentage of calls via function pointer types that cause in np-hard analysis.
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 (such as those examined in this study).

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 studied systems extensively use function pointers that are known to cause in NP-hard analysis in the indirect calls compared to the other types as shown in Figure 33. That is, if we have a means to resolve this problem, those systems will obviously be greatly affected making static analysis easier.

5.7.3 Virtual Methods and Function Pointers declarations

To better address R2 aside from the distribution of calls via virtual methods and function pointers distribution, all twelve systems were examined with respect to virtual method and function pointer declarations.

Table 10 and Figure 32 present the number of virtual methods and function pointers statically declared in each system. This includes all definitions of function pointers as parameters, in structures, in arrays, and as global declarations [Anand Shah 1995]. The number of virtual methods and such function pointer types in these systems varies greatly. For example, in gcc, 2,194 function pointer static declarations were detected; 1873 of these
were types that cause in NP-hard analysis. OpenMPI also has a significant number, 1,078 and 1,032 of them were types that cause NP-hard analysis. The remaining systems have relatively far fewer.
Figure 35. Distribution of calls via function pointer types percentage for all 12 systems
On the other hand, virtual methods are relatively high in almost all the systems that have C++ code. That is, gcc has 32,978 which is the highest, and then followed by KOffice and KDElibs 6,244 and 5,199 respectively. LLVM has 2,459 and the minimum was in Xapian with 461.

5.8 Historical Trends

We now examine how the presences of function calls conducted via function pointers and virtual methods changed over a 10-year period for the studied systems. Each of the systems, with the exception of OpenMPi, has been under development for 10 years or more. To address R3 we examined the most recent 10-year period of those systems. OpenMPi is relatively a newer project and has a shorter version history starts 2004; it was included in this comparison though. Our goal is to uncover how each 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 each system. Additionally, we put more focus on the change in the presence of calls via function pointer types that cause in NP-hard analysis. Our feeling is that this information could lead to recommendations for refactoring and improving large-scale software systems for more efficient and less complex inter-procedural and static analysis.

The change in the number of function calls conducted with, virtual methods and function pointers, and presences of each type of the function pointers that cause in NP-hard analysis was computed for each version in the same manner as we described in the previous sections. These values were aggregated for each year so the systems could be compared
on a yearly basis. The systems were 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 each of the 12 systems is presented in

Table 12. During the 10-year period all systems show a rise trend during the duration except KOffice. KOffice increased in the number of function pointer calls from 2001 to 2007. However, the number started to decrease until 2010 where it dropped to almost 40 times comparing to what it was in 2007.

A comparison of these two gcc versions shows that the total number of function pointer calls in the system increased from 718 to 2,132, an almost 200% increase. However, the number of functions with side effects was close to double, from 10,554 to 20,645 functions with side effects. KDElibs has a 100% jump in 2008 compared to 2007, and the number was increased three times in OpenMPI from 599 to 1709 in 2005. Generally, the trend was increase in most of the systems, which is steady increase with little variation in the increase rate except for KOffice, which drops dramatically from 424 in 2006 to only 11 in 2010.
Table 13 presents the number of virtual method calls for seven systems out of the twelve systems that use the object-oriented aspects of C++. During the 10-year period most of the systems show a fairly increase trend during the duration except for Xpain. Two systems, OSG and KOffice, have a steep decline in 2006 and then are relatively steady increase in proceeding years. Xpain has continuous decline started early on until 2008 and then are relatively flat in proceeding years.

Figure 36 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 most of the systems show a relatively flat trend during the duration. Quantlib has a steep decline in 2005 and then is relatively flat in proceeding years. OSG dramatically jumped in 2009 from 50% to 96% and then flattens in proceeding years. LLVM shows a large variation during the first three years of the observed ten-year time period, with a particular large jump from 2003 to 2004 and then shows the same flat trend in the remaining years during the whole time period. Precise numbers are shown in Table 14.

Figure 37 presents the percentage of function calls via global function pointers for all studied systems. During the 10-year period most of the systems show a relatively flat trend during the duration except for KOffice. KOffice increased in the number of function pointer calls via global function pointers from 2001 to 2007. However, the number started to decrease until 2010 where it dropped to almost 40%. Concerning KOffice, it is approximately a mirror of
Table 12. That is, the number of the calls via function pointers has the same trend in KOffice, increasing until 2007 and then dropping. Xapian also dramatically dropped in 2003 from 40% to 0% and then has a large jump in 2005 back to 50%, and then fairly flattens in proceeding years.
Table 12. NUMBER OF CALLS VIA FUNCTION POINTERS, all types, FOR A TEN-YEAR PERIOD FOR THE TWELVE SYSTEMS

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>951</td>
<td>718</td>
<td>2132</td>
<td>2593</td>
<td>2913</td>
<td>2896</td>
<td>3076</td>
<td>3463</td>
<td>3429</td>
<td>3850</td>
<td>4131</td>
</tr>
<tr>
<td>Ruby</td>
<td>59</td>
<td>297</td>
<td>377</td>
<td>429</td>
<td>451</td>
<td>816</td>
<td>893</td>
<td>768</td>
<td>789</td>
<td>818</td>
<td>814</td>
</tr>
<tr>
<td>Httpd</td>
<td>200</td>
<td>278</td>
<td>238</td>
<td>253</td>
<td>244</td>
<td>241</td>
<td>242</td>
<td>236</td>
<td>237</td>
<td>249</td>
<td>227</td>
</tr>
<tr>
<td>OpenMPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>27</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>Python</td>
<td>407</td>
<td>518</td>
<td>549</td>
<td>587</td>
<td>623</td>
<td>693</td>
<td>797</td>
<td>831</td>
<td>861</td>
<td>871</td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>11</td>
<td>294</td>
<td>305</td>
<td>323</td>
<td>323</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>114</td>
<td>41</td>
<td>98</td>
<td>106</td>
<td>126</td>
<td>208</td>
<td>459</td>
<td>1055</td>
<td>1180</td>
<td>1222</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>16</td>
<td>27</td>
<td>76</td>
<td>164</td>
<td>325</td>
<td>356</td>
<td>424</td>
<td>334</td>
<td>155</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>494</td>
<td>572</td>
<td>636</td>
<td>916</td>
<td>1044</td>
<td>1304</td>
<td>1396</td>
<td>1018</td>
<td>1480</td>
<td>351</td>
<td>908</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>15</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>63</td>
<td>67</td>
<td>72</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>2</td>
<td>13</td>
<td>19</td>
<td>44</td>
<td>44</td>
<td>49</td>
<td>54</td>
<td>60</td>
<td>89</td>
<td>173</td>
<td>308</td>
</tr>
</tbody>
</table>
Table 13. Number of calls via virtual methods for a ten-year period for sub set of the twelve systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>194</td>
<td>195</td>
<td>210</td>
<td>212</td>
<td>225</td>
<td>206</td>
<td>214</td>
<td>220</td>
<td>236</td>
<td>295</td>
<td>328</td>
</tr>
<tr>
<td>OSG</td>
<td>166</td>
<td>1389</td>
<td>394</td>
<td>401</td>
<td>471</td>
<td>531</td>
<td>371</td>
<td>429</td>
<td>499</td>
<td>546</td>
<td>550</td>
</tr>
<tr>
<td>Quantlip</td>
<td>73</td>
<td>39</td>
<td>38</td>
<td>69</td>
<td>85</td>
<td>94</td>
<td>90</td>
<td>114</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>3347</td>
<td>3622</td>
<td>5151</td>
<td>5799</td>
<td>6712</td>
<td>6300</td>
<td>6763</td>
<td>8762</td>
<td>9167</td>
<td>10555</td>
<td>10555</td>
</tr>
<tr>
<td>KOffice</td>
<td>1741</td>
<td>1935</td>
<td>2523</td>
<td>3039</td>
<td>8410</td>
<td>8630</td>
<td>7542</td>
<td>8894</td>
<td>10643</td>
<td>10845</td>
<td>10845</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>128</td>
<td>125</td>
<td>93</td>
<td>50</td>
<td>42</td>
<td>35</td>
<td>29</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>LLVM</td>
<td>13</td>
<td>73</td>
<td>73</td>
<td>185</td>
<td>269</td>
<td>297</td>
<td>384</td>
<td>435</td>
<td>566</td>
<td>767</td>
<td>715</td>
</tr>
</tbody>
</table>
Table 14. Number of calls via function pointer types that cause in np-hard analysis for a ten-year period for the twelve systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>900</td>
<td>665</td>
<td>2,014</td>
<td>2,469</td>
<td>2,801</td>
<td>2,774</td>
<td>2,940</td>
<td>3,319</td>
<td>3,273</td>
<td>3,570</td>
<td>3,852</td>
</tr>
<tr>
<td>Ruby</td>
<td>58</td>
<td>287</td>
<td>355</td>
<td>405</td>
<td>439</td>
<td>798</td>
<td>874</td>
<td>750</td>
<td>770</td>
<td>798</td>
<td>795</td>
</tr>
<tr>
<td>Httpd</td>
<td>196</td>
<td>274</td>
<td>235</td>
<td>250</td>
<td>241</td>
<td>238</td>
<td>239</td>
<td>230</td>
<td>230</td>
<td>242</td>
<td>220</td>
</tr>
<tr>
<td>OpenMPi</td>
<td></td>
<td></td>
<td>586</td>
<td>1,675</td>
<td>3,646</td>
<td>3,562</td>
<td>2,501</td>
<td>2,873</td>
<td>6,986</td>
<td>7,423</td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>26</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Python</td>
<td>307</td>
<td>417</td>
<td>448</td>
<td>478</td>
<td>514</td>
<td>584</td>
<td>685</td>
<td>713</td>
<td>741</td>
<td>751</td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>11</td>
<td>290</td>
<td>301</td>
<td>319</td>
<td>319</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>103</td>
<td>30</td>
<td>86</td>
<td>88</td>
<td>106</td>
<td>186</td>
<td>446</td>
<td>1,045</td>
<td>1,169</td>
<td>1,211</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>16</td>
<td>27</td>
<td>72</td>
<td>158</td>
<td>318</td>
<td>349</td>
<td>417</td>
<td>327</td>
<td>146</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>468</td>
<td>546</td>
<td>616</td>
<td>894</td>
<td>1,018</td>
<td>1,280</td>
<td>1,370</td>
<td>994</td>
<td>1,444</td>
<td>340</td>
<td>889</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>15</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>60</td>
<td>64</td>
<td>69</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>36</td>
<td>36</td>
<td>43</td>
<td>48</td>
<td>54</td>
<td>80</td>
<td>163</td>
<td>294</td>
</tr>
</tbody>
</table>
Figure 36. Evolution of calls via function pointer types that cause in np-hard analysis for 10 years period for all 12 systems
Figure 38 shows the percentage of function calls via function pointers that are structure fields for all studied systems. During the 10-year period most of the systems show a relatively flat trend during the duration. Two systems, OSG and LLVM, have a dramatic decline early on and then are relatively flat in proceeding years. The figure also shows that both subversion and python have the highest percentages 30%, 20% respectively over the other systems in the last seven years.

The percentage of function calls via function pointers that are declared as formal parameters are presented in Figure 39 for all studied systems. During the 10-year period most of the systems show a flat trend during the period. Two systems, OSG and Xapian, have a noticeable jump in 2010 and 2007 respectively and then are relatively flat in proceeding years. Additionally, the figure also shows that they both have the highest percentages 72% and 31% respectively over the other studied systems.

Finally,
Table 15 presents the number of function calls that are conducted by function pointer class members for seven systems out of the twelve systems that use the object-oriented aspects of C++. During the 10-year period most of the systems show a fairly increase trend during the duration except for KOffice. KOffice has shown a steep decline in 2007 and then a relatively steady decrease in proceeding years. To better understand the overall trend of the function pointer types’ evolution over the ten years for all the systems, we studied the average percentage of each type amongst function pointer types that cause in NP-hard analysis in all systems for all analyzed versions and present the results in Figure 37. A comparison of these averages shows that the total percentage of global function pointer usage for function calls in the systems increase overtime and is been the dominant thus the most prevalent. However, the other types have fairly a flat trend in average.

Much of this was due to an apparent large reengineering of the system to rely on a 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.
Figure 37. Evolution of calls via global function pointer for 10-year period for all 12 systems
Figure 38. Evolution of calls via function pointer structure fields for 10 year period for all 12 systems
Figure 39. Evolution of calls via function pointer function parameters for 10 year period for all 12 system
Table 15. Percentage of calls via function pointer class members over a ten-year period for seven systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>89</td>
<td>93</td>
<td>103</td>
<td>136</td>
<td>131</td>
<td>142</td>
<td>151</td>
<td>165</td>
<td>184</td>
<td>189</td>
<td>191</td>
</tr>
<tr>
<td>OSG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Quantlip</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>15</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>20</td>
<td>12</td>
<td>31</td>
<td>32</td>
<td>29</td>
<td>30</td>
<td>74</td>
<td>139</td>
<td>140</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>11</td>
<td>9</td>
<td>17</td>
<td>43</td>
<td>66</td>
<td>60</td>
<td>52</td>
<td>53</td>
<td>29</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>XAPAIN</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>19</td>
<td>30</td>
<td>64</td>
<td>93</td>
<td></td>
</tr>
</tbody>
</table>
Figure 40. Average percentage of function pointer types that cause in np-hard analysis
5.9 Discussion

The first research question (R1) addresses the numbers and percentage of calls via function pointers and virtual methods within the systems studied. It can be clearly seen that the amount of function calls that are conducted by both function pointers and virtual methods used by developers varies considerably across the studied systems. That is, numbers vary based on the systems size (Lines of code) and domain.

Obviously, from Figure 33 the average percentage of calls performed through function pointer types that cause NP-hard in analysis is significantly larger than other types. We can easily infer that the studied systems require much 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 is proposed for these systems, and thus poses problems for this type of analysis. Additionally, some studied systems amongst those that use the object-oriented aspects of C++ are still using function pointers although the virtual method feature is available and known as one of the OOP privileges.

The implication here is that static analysis and inter-procedural of general-purpose applications are getting more difficult to be performed. That is, the costs 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 and 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 [Aigner, Holzle 1996] to eliminating function pointers and virtual method calls should be also considered.

The next research questions (R2) addressed the makeup and distribution of calls via function pointer types that known to cause in NP-hard analysis. This is a special case of the function pointer calls. Our interest in this special case has to do with the complexity of analysis. Our findings have an important implication to the problem of statically analyzing general-purpose applications in the presence of indirect calls via special types 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 the global function pointers are the greatest roadblock to easier static and inter-procedural analysis of general-purpose applications. In fact, we see that Figure 34 the vast majority of calls that are performed via global function pointers that is amongst the types that cause in NP-hard analysis (70%), followed by parameters (11.9%), and lastly arrays of function pointers (0.1%).

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. Technically speaking, identifying the types of function pointers that cause in NP-hard analysis is critical knowledge for development teams with the goal of optimizing systems for automated static analysis. Coding practices aimed at avoiding the common types, as found in this study, can also be developed. Making developers better 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 [Aigner, Holzle 1996].

Our study also shows that developers are still using function pointers as class members although the virtual method feature is considered a replacement to the function pointers in C++ [Aigner, Holzle 1996; Bacon, Sweeney 1996b; Calder, Grunwald 1994; Dean, Grove, Chambers 1995; Pande, Ryder 1994]. For example, about 72% of the indirect function calls are via function pointers and 28% are conducted via virtual methods in Quantlib. OSG, Xapian, and LLVM have about 93%, 74%, and 30% function pointer calls out of the total number of indirect calls respectively.

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. On average, we found a great deal of function pointer and virtual method usage across most of the studied systems that show an increasing trend overtime. There is a small
decrease in the number of function pointers in Koffice along with a corresponding significant increase in the number of calls via virtual methods.

Thus, we can surmise that developers do not pay attention to the complexity posed when trying to simplifying code by providing a simple way to select a function to execute based on run-time values using function pointers and virtual methods. The fact that development teams do not focus on improving the analyzability of source code is particularly telling in

Table 13,

Table 14, and Figure 36. None of the systems’ history demonstrates any systematic decrease in the number of indirect calls overtime. Even though systems such as KOffice seem to reduce the usage of function pointers, there is still an increase in the usage of virtual methods. Unfortunately, that does not help too much since virtual methods are known to cause in NP-hard analysis as well [Aigner, Holzle 1996; Calder, Grunwald 1994; Pande, Ryder 1994; Sundaresan et al. 2000].

Again, analysis methods can be used to document functions and methods that have side effects. That is, when a function pointer is used to invoke a function or a method it is already known in advance if it has any kind of analysis obstacles. An approach such as labeling methods with stereotypes [Dragan, Collard, Maletic 2009b] is one example.
Refactorings could be developed to deal with the most prevalent function pointer types (e.g., Global function pointers) and virtual methods. Wise usage of indirect calls and systematic transformation of these types manually or through automated tools is most likely the only practical approach to avoiding challenges caused by these types. There are some techniques proposed in literature [Aigner, Holzle 1996; Calder, Grunwald 1994] that can be improved to work better with large scale software systems, and further investigations and research should be directed to this avenue. Coding standards and idioms also need to be developed to avoid usage of function pointers that cause in NP-hard analysis.

For an example, in parallelization context, in our hypothesis earlier, we stated that we believe that the tendency to assume that function pointers have a significant impact on the parallelization process is not realistic; validation of our hypothesis may change overtime. That is, if developers and software engineers come up with a technique to help remove inhibitors in parallelization, some for-loops will end up having only indirect function calls as an inhibitor thus blocked with indirect function calls. This can lead to potential increase in the impact of indirect calls via function pointers and virtual methods on the parallelization process. Because our findings support our hypothesis that says if an indirect exists within a for-loop, it usually coexists with another inhibitor to parallelization. So, if that inhibitor was safely removed, then the loop became blocked with an indirect call. Thus, indirect calls cannot be excluded any more in developing any automatic parallelization technique to save time and reduce complexity as we claimed earlier.
In conclusion, concerning R3, over the history of all studied systems, the presence of virtual methods and function pointers that cause NP-hard analysis are increasing posing more challenges in the software engineering research.

5.10 Threats to validity

The tools we developed for this study only work with languages supported by srcML (C/C++/Java). This has restricted us from using some existing benchmarks for parallelizability (e.g., Perfect Club Benchmark) or projects written in languages such as FORTRAN. It may be that certain computationally intensive applications have a much larger prevalence of these types of function pointers. We attempted to offset this issue by including projects such as OSG and Open MPI. However, our focus was on the usage of function pointers and virtual methods for a wide variety of general-purpose applications.

Upon examination of the function pointers in the study, we found that some of them were part of dead code, i.e., code that would never be executed. As part of the static analysis there was no distinction made in the study between function pointers and virtual methods in dead code or active code that might affect the accuracy of the results we present in terms of the systems complexity caused by their usage. In the future we are planning to refine the percentages to only include active code.

5.11 Conclusion

This study empirically examines the difficulty expected from using calls via function pointers and virtual method, which most of the time cause NP-hard analysis when
static or inter-procedural approaches are applied [Anand Shah 1995; Calder, Grunwald 1994]. The study included twelve open source software systems from different domains. That is, the systems are all general-purpose applications, as expected; they need to be statically analyzed both manually and automatically as they evolve overtime. There are no other studies of this type currently in the literature includes a historical study.

We found that the most used function pointer types are the ones that are known to cause more challenges to static analysis based techniques [Anand Shah 1995]. As such, more attention needs to be placed on dealing with function pointers and virtual methods if a large amount of static analysis is to occur in general purpose software systems in whatever context so they can take better advantage of available software engineering that use static analysis techniques. While we cannot completely generalize this finding to all software systems (across all domains), there is some indication that this is a common trend.

Most development teams and organizations have not focused on developing software in a way that could one day take better advantage of available software engineering tools built on the top of static analysis. For example, in context of automatic parallelization, the recent ubiquity of multicore processors gives rise to the need to educate developers and make them more aware of the problems and inhibitors to automatically parallelizing their code. Coding style can play a big role in advancing a system’s analyzability for purposes such as code parallelization. The software engineering community needs to develop standards and idioms that help developers in avoiding the unwise and improper use of virtual functions and function pointers, and these standards should most likely be based on the nature of the inter-procedural and static analysis
techniques used for processes like adaptive maintenance, parallelization, transformation and software comprehension in large-scale software systems.

Analysis of the historical data over a ten-year period of these systems shows that there is an increase in the usage of both calls using function pointers and virtual method over the lifetime of the systems, thus posing further problems for inter-procedural analysis techniques.

The associated data related to this study are available for download at www.sdml.info/downloads/ as well as the VirFptrStat tool. VirFptrStat uses srcML, which is available on www.srcML.org.
CHAPTER 6

Conclusions

This dissertation addresses several practical concerns of parallelization and analysis of large open source software systems. We address the problem at the source code level, rather than at the compiler level. Current C/C++ compilers can do a limited amount of automatic parallelization. That is, loops with fixed iteration bounds (i.e., for-loops) can, in certain situations, be directly parallelized by the compiler. Loops without fixed iteration bounds cannot, in general, be parallelized. The auto-parallelization can also take place via a tool prior to compiling. These tools look for for-loops that do not contain any parallelization inhibitors. Most development teams and organizations have not focused on developing software in a way that could one day take advantage of parallel architectures. However, the recent ubiquity of multicore processors gives rise to the need to educate developers and make them more aware of the problems and inhibitors to automatically parallelizing their code.

The research investigations presented in this thesis yield a number of research and development contributions in the areas of program comprehension and software maintenance for C/C++ source code parallelization and evolution. Specific research contributions in the domain of program comprehension include the development of methods to study for-loop parallelization inhibitors in open source C/C++ software systems, the observations derived from this study [Alnaeli, Alali, Maletic 2012b], and the
study on the prevalence of function pointer and virtual method calls in open source software in Chapter 5.

Contributions to the field of software adaptive maintenance for includes the definition of a new source code analysis technique that directly supports the evaluation of the parallelizability of C/C++ software systems, and the use of that technique to experimentally identify and diagnose problems caused by the presence of parallelization inhibitors.

Beyond these contributions to the software engineering and maintenance of C++ software systems, the research and experiments presented in this dissertation have led to a great deal of insight on the design and implementation of concepts as a part of the software design process, code parallelization, and will have a direct impact on its evolution. The empirical study conducted in this work presented in Chapter 5 demonstrates that the most prevalent inhibitor by far, is functions called within for-loops that have side effects. This single inhibitor poses the greatest challenge in adapting and re-engineering systems to better utilize modern multi-core architectures. This fact is somewhat contradictory to the literature, which is primarily focused on the removal of data dependencies within loops. Results of this work also show that function calls via function pointers and virtual methods have very little impact on the for-loop parallelization process.

The empirical studies presented in Chapter 5 can also be reinterpreted as a study of complexity and difficulty that should be expected by evaluating the amount of function pointer and virtual method usage (indirect function calls) in the context of automatic parallelization process. The results show that in a majority of the time 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 these systems shows that there is an increase in the usage of both calls using function pointers and virtual method over the lifetime the systems, thus posing further problems for inter-procedural analysis.

The work presented in this dissertation is provides a solid foundation for the further exploration of problems related to the design and implementation of tools that can led to safe and easy automatic parallelization at the source code level. Future work supported by the research presented in this dissertation includes:

- Leveraging the results of this work to design a new set of techniques for automatic parallelization and inhibitors detection.
- Leveraging the results of this work to design and develop a new set of coding standards and idioms to avoid inhibitors.
- Developing a methodology for using empirical studies of software usage to motivate or constrain the evolution of software engineering research and techniques to fit software parallelization needs, and
- Creating techniques and courses for teaching programming concepts for parallel programming, and software engineering for parallel programming.

Parallel programming and automatic parallelization will become increasingly important in the software development community, especially in C++ because they provide features that few other paradigms can offer: parallelization API’s support, simple
transformation methods for representing the code in descriptive languages like XML through tools such as srcML. As interest in sequential code parallelization problem and techniques for parallel programming and the number of libraries increase, must be put in place to ensure the continuing quality of these techniques and tools overtime. The work presented in this dissertation provides initial steps in those directions.

Functions for Parallelization Stereotypes will become increasingly important in the software development community, especially in C/C++. Analysis methods can be used to document for both for-loops that have any type of parallelization inhibitors, and functions and methods that have any type of side effect. Approaches such as labeling methods with stereotypes (Dragan, Collard and Maletic 2009) is one example. Methods that are access-only (i.e., get or predicate methods), have no side effects and as such are not inhibitors. This type of preprocessing and labeling could be of great use for compilers, as they typically avoid analyzing functions called within the bodies of for-loops that are considered for automatic parallelization. Upfront function analysis could greatly decrease parallelization development time. That is because it takes excessive computation which cause longer compilation time.

For support this problem, we are proposing a method for helping the compiler determining the safe calls to parallelization. We believe that by automatically re-document each for-loop as well as function and method by annotating the original source code with simple information that helps the compiler identifying the free for-loops and functions and methods that have no side effect in $O(1)$ time.
We propose to develop an approach to automatically identify the safe functions and methods that can be called safely by parallelized loop in an entire system. That can be error prone and time consuming, costly, manual solution is considered. We are proposing to use a lightweight static analysis method for the side effect identification and annotation production. The method must be naturally lightweight so that it can be very efficient and scalable. This dissertation results suggest several areas for future research. One possible research area is the study of function side effects evolution in open source systems and real programs written in both procedural and object-oriented languages like C++, C#, and Java.

We will be investigating the possibility to extending this approach as attempt for generalizing it on all functions of the system however in different classification, since our concern is the side effect detection in the context of software parallelization. Additionally, this similar approach will be also considered for stereotyping for-loops as well to supporting compilers for automatic parallelization.
### Table 16. Distribution of percentage of calls via function pointer types that cause in np-hard analysis

<table>
<thead>
<tr>
<th>Systems</th>
<th>QuantLib</th>
<th>GCC</th>
<th>Python</th>
<th>Ruby</th>
<th>Subversion</th>
<th>LLVM</th>
<th>xapian</th>
<th>Ht tp</th>
<th>KOffice</th>
<th>KDELibs</th>
<th>OpenSceneGraph</th>
<th>OpenMpi</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global</strong></td>
<td>80%</td>
<td>80%</td>
<td>75%</td>
<td>84%</td>
<td>69%</td>
<td>65%</td>
<td>63%</td>
<td>97%</td>
<td>40%</td>
<td>86%</td>
<td>2%</td>
<td>99%</td>
</tr>
<tr>
<td><strong>Structure Field</strong></td>
<td>6%</td>
<td>6%</td>
<td>19%</td>
<td>10%</td>
<td>30%</td>
<td>1%</td>
<td>3%</td>
<td>2%</td>
<td>0%</td>
<td>2%</td>
<td>9%</td>
<td>1%</td>
</tr>
<tr>
<td><strong>Formal parameters</strong></td>
<td>4%</td>
<td>10%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
<td>3%</td>
<td>31%</td>
<td>1%</td>
<td>10%</td>
<td>0%</td>
<td>72%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Class members</strong></td>
<td>9%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>32%</td>
<td>3%</td>
<td>0%</td>
<td>50%</td>
<td>12%</td>
<td>17%</td>
<td>0%</td>
</tr>
<tr>
<td><strong>array elements</strong></td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Table 17. Evolution of calls via function pointer types that cause in np-hard analysis for 10 years period for all 12 systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>95%</td>
<td>93%</td>
<td>94%</td>
<td>95%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>95%</td>
<td>93%</td>
<td>93%</td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td>98%</td>
<td>97%</td>
<td>94%</td>
<td>94%</td>
<td>97%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
</tr>
<tr>
<td>Httpd</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>97%</td>
<td>97%</td>
<td>97%</td>
<td>97%</td>
<td>97%</td>
</tr>
<tr>
<td>OpenMPi</td>
<td></td>
<td></td>
<td></td>
<td>98%</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>100%</td>
<td>99%</td>
</tr>
<tr>
<td>OSG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>75%</td>
<td>81%</td>
<td>82%</td>
<td>81%</td>
<td>83%</td>
<td>84%</td>
<td>86%</td>
<td>86%</td>
<td>86%</td>
<td>86%</td>
<td>86%</td>
</tr>
<tr>
<td>Quantlip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>90%</td>
<td>73%</td>
<td>88%</td>
<td>83%</td>
<td>84%</td>
<td>89%</td>
<td>97%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>KOffice</td>
<td>100%</td>
<td>100%</td>
<td>95%</td>
<td>96%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>97%</td>
<td>91%</td>
</tr>
<tr>
<td>Subversion</td>
<td>95%</td>
<td>95%</td>
<td>97%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>98%</td>
<td>97%</td>
<td>98%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>95%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
<td>96%</td>
</tr>
<tr>
<td>LLVM</td>
<td>50%</td>
<td>85%</td>
<td>63%</td>
<td>82%</td>
<td>82%</td>
<td>88%</td>
<td>89%</td>
<td>90%</td>
<td>90%</td>
<td>94%</td>
<td>95%</td>
</tr>
</tbody>
</table>
Table 18. Evolution of calls via global function pointer for 10-year period for all 12 systems

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>59%</td>
<td>77%</td>
<td>88%</td>
<td>83%</td>
<td>86%</td>
<td>85%</td>
<td>84%</td>
<td>82%</td>
<td>79%</td>
<td>79%</td>
<td>80%</td>
</tr>
<tr>
<td>Ruby</td>
<td>45%</td>
<td>98%</td>
<td>97%</td>
<td>95%</td>
<td>84%</td>
<td>93%</td>
<td>90%</td>
<td>79%</td>
<td>79%</td>
<td>81%</td>
<td>84%</td>
</tr>
<tr>
<td>Httpd</td>
<td>93%</td>
<td>88%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>89%</td>
<td>89%</td>
<td>89%</td>
<td>89%</td>
<td>89%</td>
<td>97%</td>
</tr>
<tr>
<td>OpenMPi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>98%</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>OSG</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>2%</td>
<td>2%</td>
</tr>
<tr>
<td>Python</td>
<td>61%</td>
<td>70%</td>
<td>67%</td>
<td>69%</td>
<td>72%</td>
<td>71%</td>
<td>75%</td>
<td>76%</td>
<td>76%</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>83%</td>
<td>79%</td>
<td>80%</td>
<td>80%</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>67%</td>
<td>53%</td>
<td>62%</td>
<td>61%</td>
<td>71%</td>
<td>83%</td>
<td>72%</td>
<td>84%</td>
<td>86%</td>
<td>86%</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>13%</td>
<td>26%</td>
<td>54%</td>
<td>51%</td>
<td>68%</td>
<td>75%</td>
<td>82%</td>
<td>77%</td>
<td>48%</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>86%</td>
<td>85%</td>
<td>75%</td>
<td>81%</td>
<td>76%</td>
<td>78%</td>
<td>75%</td>
<td>66%</td>
<td>71%</td>
<td>64%</td>
<td>69%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>67%</td>
<td>40%</td>
<td>0%</td>
<td>0%</td>
<td>50%</td>
<td>50%</td>
<td>63%</td>
<td>59%</td>
<td>62%</td>
<td>63%</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>0%</td>
<td>27%</td>
<td>42%</td>
<td>69%</td>
<td>72%</td>
<td>77%</td>
<td>69%</td>
<td>44%</td>
<td>55%</td>
<td>55%</td>
<td>65%</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>GCC</td>
<td>27%</td>
<td>3%</td>
<td>2%</td>
<td>5%</td>
<td>4%</td>
<td>5%</td>
<td>5%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>Ruby</td>
<td>48%</td>
<td>1%</td>
<td>1%</td>
<td>1%</td>
<td>7%</td>
<td>2%</td>
<td>1%</td>
<td>11%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
</tr>
<tr>
<td>Httpd</td>
<td>7%</td>
<td>12%</td>
<td>9%</td>
<td>9%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>2%</td>
</tr>
<tr>
<td>OpenMPI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2%</td>
</tr>
<tr>
<td>OSG</td>
<td>100%</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>35%</td>
<td>9%</td>
<td>9%</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>36%</td>
<td>27%</td>
<td>29%</td>
<td>28%</td>
<td>26%</td>
<td>23%</td>
<td>20%</td>
<td>19%</td>
<td>19%</td>
<td>19%</td>
<td>19%</td>
</tr>
<tr>
<td>Quantlip</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>14%</td>
<td>7%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td>1%</td>
<td>0%</td>
<td>2%</td>
<td>2%</td>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>19%</td>
<td>41%</td>
<td>17%</td>
<td>16%</td>
<td>8%</td>
<td>5%</td>
<td>4%</td>
<td>5%</td>
<td>10%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>13%</td>
<td>14%</td>
<td>24%</td>
<td>19%</td>
<td>24%</td>
<td>22%</td>
<td>25%</td>
<td>34%</td>
<td>29%</td>
<td>36%</td>
<td>30%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td>3%</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>100%</td>
<td>9%</td>
<td>25%</td>
<td>8%</td>
<td>8%</td>
<td>0%</td>
<td>6%</td>
<td>6%</td>
<td>0%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
Table 20. Function calls in C++ systems via Virtual Methods vs. Function Pointers. Shows how some systems still use both.

<table>
<thead>
<tr>
<th>Systems</th>
<th>Total indirect function calls</th>
<th>Via Virtual Methods</th>
<th>via Function Pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>4,459</td>
<td>328</td>
<td>4,131</td>
</tr>
<tr>
<td>OpenSceneGraph</td>
<td>8012</td>
<td>550</td>
<td>7,462</td>
</tr>
<tr>
<td>Quantlip</td>
<td>446</td>
<td>123</td>
<td>323</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>11,777</td>
<td>10,555</td>
<td>1,222</td>
</tr>
<tr>
<td>Koffice</td>
<td>10,856</td>
<td>10,845</td>
<td>11</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>99</td>
<td>26</td>
<td>73</td>
</tr>
<tr>
<td>LLVM</td>
<td>1,023</td>
<td>715</td>
<td>308</td>
</tr>
<tr>
<td>Systems</td>
<td>QuantLib</td>
<td>GCC</td>
<td>Python</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>Global</td>
<td>46</td>
<td>602</td>
<td>128</td>
</tr>
<tr>
<td>Structure Field</td>
<td>68</td>
<td>579</td>
<td>175</td>
</tr>
<tr>
<td>Formal parameters</td>
<td>7</td>
<td>320</td>
<td>97</td>
</tr>
<tr>
<td>Class members</td>
<td>19</td>
<td>329</td>
<td>0</td>
</tr>
<tr>
<td>array elements</td>
<td>2</td>
<td>43</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 22. Data dependency in for-loops for systems studied in Chapter 4.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>6.2%</td>
<td>6.3%</td>
<td>6.3%</td>
<td>6.7%</td>
<td>7.4%</td>
<td>9.8%</td>
<td>10.3%</td>
<td>11.3%</td>
<td>9.2%</td>
<td>9.5%</td>
<td>9.4%</td>
</tr>
<tr>
<td>Ruby</td>
<td>5.3%</td>
<td>7.3%</td>
<td>7.3%</td>
<td>7.2%</td>
<td>5.8%</td>
<td>5.0%</td>
<td>6.1%</td>
<td>6.7%</td>
<td>6.7%</td>
<td>6.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>Httpd</td>
<td>5.6%</td>
<td>6.3%</td>
<td>5.4%</td>
<td>5.6%</td>
<td>5.9%</td>
<td>6.1%</td>
<td>5.8%</td>
<td>5.7%</td>
<td>5.2%</td>
<td>5.4%</td>
<td>5.4%</td>
</tr>
<tr>
<td>OpenMPi</td>
<td></td>
<td>6.6%</td>
<td>10.0%</td>
<td>12.7%</td>
<td>11.0%</td>
<td>10.4%</td>
<td>10.2%</td>
<td>10.1%</td>
<td>10.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>13.8%</td>
<td>8.5%</td>
<td>6.5%</td>
<td>5.8%</td>
<td>5.5%</td>
<td>5.3%</td>
<td>4.7%</td>
<td>4.9%</td>
<td>7.5%</td>
<td>7.4%</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>5.8%</td>
<td>6.4%</td>
<td>6.6%</td>
<td>6.6%</td>
<td>6.5%</td>
<td>7.7%</td>
<td>7.6%</td>
<td>8.1%</td>
<td>7.9%</td>
<td>7.9%</td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>17.6%</td>
<td>19.1%</td>
<td>20.9%</td>
<td>26.3%</td>
<td>23.1%</td>
<td>24.4%</td>
<td>22.3%</td>
<td>20.6%</td>
<td>20.2%</td>
<td>20.0%</td>
<td>19.8%</td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>7.4%</td>
<td>6.3%</td>
<td>5.3%</td>
<td>5.5%</td>
<td>5.6%</td>
<td>6.0%</td>
<td>4.7%</td>
<td>4.7%</td>
<td>4.5%</td>
<td>4.5%</td>
<td></td>
</tr>
<tr>
<td>KOffice</td>
<td>5.1%</td>
<td>4.7%</td>
<td>6.5%</td>
<td>6.9%</td>
<td>6.8%</td>
<td>6.2%</td>
<td>7.8%</td>
<td>6.8%</td>
<td>6.5%</td>
<td>3.6%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>5.8%</td>
<td>2.6%</td>
<td>2.7%</td>
<td>3.0%</td>
<td>2.3%</td>
<td>2.7%</td>
<td>2.3%</td>
<td>2.1%</td>
<td>2.4%</td>
<td>2.2%</td>
<td>2.2%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>11.2%</td>
<td>9.5%</td>
<td>10.0%</td>
<td>11.0%</td>
<td>10.0%</td>
<td>10.0%</td>
<td>7.8%</td>
<td>6.8%</td>
<td>7.6%</td>
<td>7.7%</td>
<td>5.5%</td>
</tr>
<tr>
<td>LLVM</td>
<td>1.5%</td>
<td>2.5%</td>
<td>2.2%</td>
<td>3.6%</td>
<td>4.1%</td>
<td>3.8%</td>
<td>3.5%</td>
<td>4.1%</td>
<td>4.0%</td>
<td>3.5%</td>
<td>3.3%</td>
</tr>
</tbody>
</table>
Table 23. Goto statement in for-loops for systems studied in Chapter 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>2.7%</td>
<td>2.5%</td>
<td>2.4%</td>
<td>2.7%</td>
<td>2.6%</td>
<td>2.6%</td>
<td>2.4%</td>
<td>2.3%</td>
<td>2.2%</td>
<td>2.3%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Ruby</td>
<td>6.3%</td>
<td>5.4%</td>
<td>5.1%</td>
<td>3.7%</td>
<td>4.8%</td>
<td>4.8%</td>
<td>4.8%</td>
<td>4.8%</td>
<td>5.1%</td>
<td>5.2%</td>
<td></td>
</tr>
<tr>
<td>Httpd</td>
<td>2.4%</td>
<td>2.1%</td>
<td>2.1%</td>
<td>2.2%</td>
<td>2.1%</td>
<td>2.1%</td>
<td>1.7%</td>
<td>1.8%</td>
<td>1.7%</td>
<td>1.9%</td>
<td></td>
</tr>
<tr>
<td>OpenMPi</td>
<td></td>
<td>2.3%</td>
<td>7.2%</td>
<td>6.3%</td>
<td>8.7%</td>
<td>6.9%</td>
<td>7.9%</td>
<td>7.5%</td>
<td>6.4%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>0.4%</td>
<td>0.2%</td>
<td>0.5%</td>
<td>0.3%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Python</td>
<td>0.2%</td>
<td>0.4%</td>
<td>1.1%</td>
<td>1.5%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>0.9%</td>
<td>0.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>10.5%</td>
<td>11.0%</td>
<td>11.5%</td>
<td>12.1%</td>
<td>11.9%</td>
<td>11.7%</td>
<td>11.3%</td>
<td>13.9%</td>
<td>14.0%</td>
<td>13.7%</td>
<td></td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>0.0%</td>
<td>0.7%</td>
<td>0.3%</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.3%</td>
<td>0.2%</td>
<td>0.4%</td>
<td>0.4%</td>
<td>0.3%</td>
<td>0.3%</td>
</tr>
<tr>
<td>KOffice</td>
<td>0.4%</td>
<td>0.5%</td>
<td>0.9%</td>
<td>0.9%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>1.1%</td>
<td>1.0%</td>
<td>1.0%</td>
<td>0.9%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>0.8%</td>
<td>2.0%</td>
<td>2.2%</td>
<td>1.9%</td>
<td>2.2%</td>
<td>2.0%</td>
<td>2.0%</td>
<td>1.9%</td>
<td>1.7%</td>
<td>1.8%</td>
<td>2.0%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>0.0%</td>
<td>0.6%</td>
<td>0.6%</td>
<td>1.0%</td>
<td>0.8%</td>
<td>0.5%</td>
<td>0.4%</td>
<td>0.4%</td>
<td>0.6%</td>
<td>0.7%</td>
<td></td>
</tr>
<tr>
<td>LLVM</td>
<td>5.5%</td>
<td>5.0%</td>
<td>5.0%</td>
<td>5.3%</td>
<td>3.5%</td>
<td>3.5%</td>
<td>1.6%</td>
<td>0.4%</td>
<td>0.6%</td>
<td>0.5%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>
Table 24. Break statement in for-loops for systems studied in Chapter 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>18.9%</td>
<td>18.1%</td>
<td>18.1%</td>
<td>17.3%</td>
<td>16.4%</td>
<td>15.8%</td>
<td>15.0%</td>
<td>14.0%</td>
<td>14.3%</td>
<td>14.4%</td>
<td>14.5%</td>
</tr>
<tr>
<td>Ruby</td>
<td>20.6%</td>
<td>24.4%</td>
<td>22.4%</td>
<td>23.1%</td>
<td>20.2%</td>
<td>20.0%</td>
<td>18.8%</td>
<td>18.7%</td>
<td>18.5%</td>
<td>18.1%</td>
<td>18.3%</td>
</tr>
<tr>
<td>Httpd</td>
<td>21.4%</td>
<td>21.2%</td>
<td>23.8%</td>
<td>24.9%</td>
<td>25.1%</td>
<td>24.9%</td>
<td>25.5%</td>
<td>22.6%</td>
<td>22.1%</td>
<td>22.3%</td>
<td>23.6%</td>
</tr>
<tr>
<td>OpenMPI</td>
<td>14.8%</td>
<td>14.7%</td>
<td>11.6%</td>
<td>14.3%</td>
<td>17.3%</td>
<td>17.0%</td>
<td>17.9%</td>
<td>18.2%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>8.6%</td>
<td>7.4%</td>
<td>6.6%</td>
<td>6.0%</td>
<td>5.7%</td>
<td>5.9%</td>
<td>6.0%</td>
<td>5.8%</td>
<td>5.1%</td>
<td>5.5%</td>
<td>5.3%</td>
</tr>
<tr>
<td>Python</td>
<td>9.9%</td>
<td>9.2%</td>
<td>12.9%</td>
<td>13.4%</td>
<td>10.9%</td>
<td>10.3%</td>
<td>10.3%</td>
<td>11.7%</td>
<td>10.9%</td>
<td>9.8%</td>
<td></td>
</tr>
<tr>
<td>Quantlip</td>
<td>18.1%</td>
<td>20.6%</td>
<td>21.0%</td>
<td>21.1%</td>
<td>19.2%</td>
<td>23.2%</td>
<td>22.7%</td>
<td>22.5%</td>
<td>22.4%</td>
<td>21.9%</td>
<td></td>
</tr>
<tr>
<td>KDELIBS*</td>
<td>1.1%</td>
<td>1.4%</td>
<td>1.9%</td>
<td>1.3%</td>
<td>1.3%</td>
<td>1.1%</td>
<td>1.3%</td>
<td>4.3%</td>
<td>4.1%</td>
<td>4.1%</td>
<td>3.9%</td>
</tr>
<tr>
<td>KOffice</td>
<td>11.0%</td>
<td>11.5%</td>
<td>12.2%</td>
<td>11.9%</td>
<td>12.1%</td>
<td>11.7%</td>
<td>13.0%</td>
<td>13.1%</td>
<td>12.8%</td>
<td>12.5%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>19.3%</td>
<td>10.6%</td>
<td>11.9%</td>
<td>12.6%</td>
<td>13.2%</td>
<td>13.4%</td>
<td>11.4%</td>
<td>11.6%</td>
<td>11.4%</td>
<td>11.2%</td>
<td>13.0%</td>
</tr>
<tr>
<td>XAPAIN</td>
<td>13.1%</td>
<td>7.3%</td>
<td>9.0%</td>
<td>12.0%</td>
<td>11.6%</td>
<td>12.4%</td>
<td>11.4%</td>
<td>10.6%</td>
<td>11.0%</td>
<td>11.6%</td>
<td>11.8%</td>
</tr>
<tr>
<td>LLVM</td>
<td>10.7%</td>
<td>9.0%</td>
<td>9.2%</td>
<td>5.7%</td>
<td>10.0%</td>
<td>10.0%</td>
<td>10.2%</td>
<td>11.6%</td>
<td>11.9%</td>
<td>11.4%</td>
<td>12.7%</td>
</tr>
</tbody>
</table>
Table 25. Free for-loops for systems studied in Chapter 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>70%</td>
<td>70%</td>
<td>48%</td>
<td>49%</td>
<td>51%</td>
<td>51%</td>
<td>53%</td>
<td>52%</td>
<td>55%</td>
<td>56%</td>
<td>56%</td>
</tr>
<tr>
<td>Ruby</td>
<td>70%</td>
<td>67%</td>
<td>66%</td>
<td>64%</td>
<td>41%</td>
<td>39%</td>
<td>40%</td>
<td>39%</td>
<td>39%</td>
<td>39%</td>
<td>39%</td>
</tr>
<tr>
<td>Httpd</td>
<td>61%</td>
<td>61%</td>
<td>59%</td>
<td>58%</td>
<td>57%</td>
<td>57%</td>
<td>57%</td>
<td>59%</td>
<td>59%</td>
<td>60%</td>
<td>57%</td>
</tr>
<tr>
<td>OpenMpi</td>
<td></td>
<td></td>
<td>53%</td>
<td>45%</td>
<td>50%</td>
<td>45%</td>
<td>45%</td>
<td>44%</td>
<td>42%</td>
<td>41%</td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>56%</td>
<td>68%</td>
<td>65%</td>
<td>66%</td>
<td>67%</td>
<td>65%</td>
<td>68%</td>
<td>67%</td>
<td>67%</td>
<td>65%</td>
<td>67%</td>
</tr>
<tr>
<td>KOffice</td>
<td>67%</td>
<td>72%</td>
<td>60%</td>
<td>57%</td>
<td>60%</td>
<td>61%</td>
<td>59%</td>
<td>55%</td>
<td>59%</td>
<td>62%</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>42%</td>
<td>41%</td>
<td>40%</td>
<td>39%</td>
<td>40%</td>
<td>40%</td>
<td>41%</td>
<td>40%</td>
<td>41%</td>
<td>41%</td>
<td>41%</td>
</tr>
<tr>
<td>Quantlip</td>
<td>72%</td>
<td>72%</td>
<td>69%</td>
<td>60%</td>
<td>63%</td>
<td>63%</td>
<td>64%</td>
<td>65%</td>
<td>65%</td>
<td>65%</td>
<td>65%</td>
</tr>
<tr>
<td>KDELibs</td>
<td>67%</td>
<td>66%</td>
<td>64%</td>
<td>64%</td>
<td>64%</td>
<td>62%</td>
<td>64%</td>
<td>61%</td>
<td>62%</td>
<td>62%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>33%</td>
<td>27%</td>
<td>27%</td>
<td>25%</td>
<td>24%</td>
<td>25%</td>
<td>27%</td>
<td>27%</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>LLVM</td>
<td>50%</td>
<td>59%</td>
<td>55%</td>
<td>53%</td>
<td>55%</td>
<td>56%</td>
<td>59%</td>
<td>58%</td>
<td>58%</td>
<td>55%</td>
<td>56%</td>
</tr>
<tr>
<td>xapian</td>
<td>60%</td>
<td>64%</td>
<td>63%</td>
<td>64%</td>
<td>66%</td>
<td>65%</td>
<td>59%</td>
<td>57%</td>
<td>56%</td>
<td>56%</td>
<td></td>
</tr>
</tbody>
</table>
Table 26. Function calls in for-loops for systems studied in Chapter 4

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>8%</td>
<td>8%</td>
<td>37%</td>
<td>37%</td>
<td>34%</td>
<td>33%</td>
<td>46%</td>
<td>31%</td>
<td>29%</td>
<td>28%</td>
<td>28%</td>
</tr>
<tr>
<td>Ruby</td>
<td>4%</td>
<td>4%</td>
<td>8%</td>
<td>11%</td>
<td>44%</td>
<td>47%</td>
<td>45%</td>
<td>46%</td>
<td>46%</td>
<td>46%</td>
<td>47%</td>
</tr>
<tr>
<td>Httpd</td>
<td>18%</td>
<td>19%</td>
<td>19%</td>
<td>19%</td>
<td>20%</td>
<td>19%</td>
<td>20%</td>
<td>19%</td>
<td>19%</td>
<td>19%</td>
<td>22%</td>
</tr>
<tr>
<td>OpenMpi</td>
<td>31%</td>
<td>38%</td>
<td>32%</td>
<td>38%</td>
<td>39%</td>
<td>40%</td>
<td>42%</td>
<td>43%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSG</td>
<td>25%</td>
<td>19%</td>
<td>24%</td>
<td>24%</td>
<td>24%</td>
<td>26%</td>
<td>23%</td>
<td>25%</td>
<td>26%</td>
<td>24%</td>
<td>23%</td>
</tr>
<tr>
<td>KOffice</td>
<td>22%</td>
<td>28%</td>
<td>28%</td>
<td>29%</td>
<td>27%</td>
<td>28%</td>
<td>28%</td>
<td>32%</td>
<td>28%</td>
<td>28%</td>
<td>28%</td>
</tr>
<tr>
<td>Python</td>
<td>46%</td>
<td>45%</td>
<td>46%</td>
<td>47%</td>
<td>46%</td>
<td>42%</td>
<td>42%</td>
<td>43%</td>
<td>42%</td>
<td>43%</td>
<td></td>
</tr>
<tr>
<td>Quantlib</td>
<td>11%</td>
<td>11%</td>
<td>13%</td>
<td>14%</td>
<td>14%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>15%</td>
<td>16%</td>
</tr>
<tr>
<td>KDELibs</td>
<td>18%</td>
<td>20%</td>
<td>21%</td>
<td>22%</td>
<td>22%</td>
<td>25%</td>
<td>23%</td>
<td>26%</td>
<td>26%</td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td>Subversion</td>
<td>54%</td>
<td>65%</td>
<td>65%</td>
<td>67%</td>
<td>68%</td>
<td>67%</td>
<td>66%</td>
<td>66%</td>
<td>68%</td>
<td>69%</td>
<td>68%</td>
</tr>
<tr>
<td>LLVM</td>
<td>39%</td>
<td>33%</td>
<td>37%</td>
<td>36%</td>
<td>35%</td>
<td>34%</td>
<td>32%</td>
<td>32%</td>
<td>33%</td>
<td>36%</td>
<td>35%</td>
</tr>
<tr>
<td>xapian</td>
<td>26%</td>
<td>23%</td>
<td>23%</td>
<td>25%</td>
<td>20%</td>
<td>20%</td>
<td>30%</td>
<td>31%</td>
<td>31%</td>
<td>31%</td>
<td></td>
</tr>
</tbody>
</table>
Table 27. The percentage of different inhibitors of for-loops for releases during the time period of 2001 – 2003 for gcc. Note that while for-loops with function-call inhibitors increased greatly, other inhibitors do not show any increase for gcc.

<table>
<thead>
<tr>
<th>Project</th>
<th>GCC 3.2.1</th>
<th>GCC3.2.2</th>
<th>GCC3.2.3</th>
<th>GCC3.3</th>
<th>GCC3.3.1</th>
<th>GCC3.3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Calls</td>
<td>7%</td>
<td>10%</td>
<td>10%</td>
<td>10%</td>
<td>34%</td>
<td>37%</td>
</tr>
<tr>
<td>Data Dependency</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
</tr>
<tr>
<td>Jumps</td>
<td>20%</td>
<td>20%</td>
<td>20%</td>
<td>19%</td>
<td>20%</td>
<td>20%</td>
</tr>
</tbody>
</table>
Figure 41. Inhibitors distribution for all studied systems in Chapter 4
<table>
<thead>
<tr>
<th>Project</th>
<th>QuanLib</th>
<th>GCC</th>
<th>Chrome</th>
<th>Python</th>
<th>Ruby</th>
<th>Subversion</th>
<th>Hippd</th>
<th>Koffice</th>
<th>KDELibs</th>
<th>OpenScene</th>
<th>LLVM</th>
<th>Xpian</th>
<th>OpenMpi</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Calls</td>
<td>45%</td>
<td>64%</td>
<td>78%</td>
<td>72%</td>
<td>77%</td>
<td>90%</td>
<td>51%</td>
<td>74%</td>
<td>68%</td>
<td>69%</td>
<td>80%</td>
<td>71%</td>
<td>72%</td>
<td>70%</td>
</tr>
<tr>
<td>Jumps</td>
<td>12%</td>
<td>30%</td>
<td>24%</td>
<td>53%</td>
<td>34%</td>
<td>20%</td>
<td>12%</td>
<td>56%</td>
<td>51%</td>
<td>26%</td>
<td>35%</td>
<td>17%</td>
<td>26%</td>
<td>31%</td>
</tr>
<tr>
<td>Data Dependency</td>
<td>54%</td>
<td>23%</td>
<td>12%</td>
<td>11%</td>
<td>3%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>12%</td>
<td>3%</td>
<td>12%</td>
<td>18%</td>
<td>16%</td>
<td>16%</td>
</tr>
</tbody>
</table>
Figure 42. How a for-loop is parallelized with openMP directives
REFERENCES


Toolkit", in Proceedings of the 2011 IEEE 11th International Working
Conference on Source Code Analysis and Manipulation: IEEE Computer Society,

[Collard, Decker, Maletic 2011b] Collard, M. L., Decker, M. J., and Maletic, J. I.,
(2011b), "Lightweight Transformation and Fact Extraction with the srcML
Toolkit", in SCAM'11. Williamsburg, VA, USA, pp. 10.

"An XML-based lightweight C++ fact extractor", in Proceedings of Program
Comprehension, 2003. 11th IEEE International Workshop on, 10-11 May 2003,
pp. 134-143.

"Supporting document and data views of source code", in Proceedings of the 2002
34-41.

Document and Data Views of Source Code", in Proceedings of ACM Symposium
on Document Engineering, pp. 8.

Transformational Approach to Support Large Scale Adaptive Changes",
Proceedings of 26th IEEE International Conference on Software Maintenance
(ICSM'10), pp. 10.
"Optimization of Object-Oriented Programs Using Static Class Hierarchy
Analysis", in Proceedings of the 9th European Conference on Object-Oriented
Programming: Springer-Verlag, pp. 77-101.

"Relooper: refactoring for loop parallelism in Java", in Proceedings of the 24th
ACM SIGPLAN conference companion on Object oriented programming systems

[Dijkstra 1979] Dijkstra, E., (1979), "Go to statement considered harmful", in Classics in
software engineering, Yourdon Press, pp. 27-33.

"Using method stereotype distribution as a signature descriptor for software

"Using method stereotype distribution as a signature descriptor for software
systems", in Proceedings of in the Proceedings of the IEEE International

"Automatic identification of class stereotypes", in Proceedings of the 2010 IEEE
1-10.


programming, systems, languages, and applications. Atlanta, Georgia, USA: ACM, pp. 108-124.


Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801.


