Empirically Examining the Parallelizability of Open Source Software Systems

Saleh M. Alnaeli
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA
salnaeli@cs.kent.edu

Abdulkareem Alali
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA
aalali@cs.kent.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242, USA
jmaletic@kent.edu

***Abstract*—An empirical study is presented that examines the potential to automatically parallelize, using refactoring tools and/or compilers, 11 open source software. Static analysis methods are applied to each system to determine the number of for-loops and free-loops (i.e., loops that can be parallized). For each non-free loop the various inhibitors (to parallelization) are determined and counted. The results show that function calls within for-loops represent the vast majority of inhibitors and thus pose the greatest roadblock to adapt and re-engineer systems to better utilize parallelization. This is somewhat contradictory to the literature, which is focused primarily on the removal of data dependencies within loops. Additionally, the historical data of inhibitor counts for the set of systems is presented over a ten-year period. The data shows few of the systems examined are increasing the potential to parallelizable loops over time.**

Keywords- Parallelization inhibitors, reegnineering, emprical study, data dependency, function calls.

I. INTRODUCTION

Recently, multicore architectures have become very prevalent for general computing needs. Almost all of today's desktops and laptops are supported with multicore architectures. This fact has pushed the need for software engineers to think more about how existing software systems can be reengineered and adapted to better utilize the underlying hardware. Popular programming languages provide little direct support for parallelization and this optimization step, if undertaken at all, is mainly left to the compiler or other specific auto-parallelization tools. This is particularly the case with more general-purpose software applications versus applications focused on numerical or scientific problems.

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 to parallelize loops and take advantage of multiple processors (cores) and shared memory. Current C/C++ compliers can do a limited amount of auto-parallelization. That is, loops (with fixed iteration bounds) can in certain situations, be directly parallelized by the compiler. This auto-parallelization can also take place via a tool prior to compiling. These tools look for loops that do not contain any parallelization *inhibitors*. Inhibitors are code statements within the body of a loop that prevent it from easily being parallelized. Data dependencies between statements within a loop are a well studied inhibitor within the parallel computing field [1]. Much of the literature on removal and detection of inhibitors is focused on data dependency. Other inhibitors include such things as conditional jumps outside the loop and function calls with side effects.

In the past, software applications addressing numerical domains or complex scientific calculations (e.g., weather prediction) were the main focus for parallelization. These types of applications often have complex loops with large arrays and matrices computations. As such data dependency is an issue for the parallelization of such systems. However, since everyone now has a multicore processor on his or her desk, a renewed focus on the parallelization more general-purpose applications is at hand.

Here we empirically examine a variety of open source software systems to better understand the roadblocks for automated, and semi-automated, parallelization tools. We are interested in determining what are the most prevalent inhibitors that occur in a wide variety of software applications and determine if there are any general trends. While this work does not directly address the problem of automated parallelization, it serves as a foundation for understanding the problem requirements in the context of a broad set of applications. Further, this study should assist in focusing the research on inhibitor detection and removal to the most common situations actually occurring in software systems.

The study consists of examining eleven C/C++ systems in various application domains. For each system the number of loops are determined (for- and while-loops) and each for-loop is analyzed further to determine if any inhibitors are present. A count of all the inhibitors found, and of what type, is kept. This data is presented to compare the different systems and uncover trends and general observations. Furthermore, the history of each system was examined and the number of inhibitors determined for each release. The trend of increasing or decreasing numbers of inhibitors is then presented.

In short, our findings show that function calls represent the vast majority of inhibitors occurring in these systems. Moreover, for the most part few systems have increased their potential to take advantage of the new hardware during their history. The remainder of this paper is organized as follows. Section 2 presents a definition of the problem and some related work on the topic of automatic parallelization and its challenges. Section 4 defines the different inhibitors to automatic parallelization of for-loops, namely data dependency, function calls, goto, and break statements. The approach we used in this work to detect data dependencies is also described. Section 5 presents a study of 11 open source software systems. The counts of for-loops, free-loops, and each inhibitor are given. Section 6 presents the data of how this data changed over time during the evolution of each system. There is a discussion in section 7 followed by threats to validity and conclusions.

II. BACKGROUND & RELATED WORK

In this study, implicit parallelism is considered with the shared memory parallel model [2, 3] and we intend to support the parallelization of an existing sequential code.

While there are multiple API's 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 of the compilers support it [4, 5]. The API is a set of standards and interfaces for parallelizing programs in a shared memory environment. It provides a set of *pragmas* (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 have both serial and parallel code. *OpenMP* makes code transformation fairly simple; any transformation can be achieved by embedding the appropriate pragmas at the proper locations. *OpenMP* has a number of limitations, for example, pragmas can only be applied to for-loops with known iteration bounds. This implies while-loops are not parallelized.

Current parallelization's APIs have common for-loops parallelization inhibitors. Some are solvable by special *OpenMP* pragmas. For example, variables 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 these later 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 [1] even though there are many other inhibitors that appear to be more frequent as we are going to show [6]. There is a large body of work on parallelization. In the 1960s, parallel computers and research on parallel languages, compilers, first began. The focus was instruction-level parallelism [7] and mainly involved detecting instructions in a program that could be executed in concurrent to reduce the computation time. Since that 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 vectorizations. Parallelizing compilers divide loop iterations to be concurrently executes on separate processors or cores.

Parallelizing compilers, such as Intel's [8] and *gcc* [9], 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 [16]. For example, compilers still cannot determine the thread-safety of a loop containing external function calls because it does not know whether the function call has side effects that would introduce dependences.

Kim *et al.* [10] introduced *Prospector*, which is a profile-based parallelism identification tool using dynamic data dependence profiling. It advises on how to parallelize some identified sections of code. The authors demonstrated that *Prospector* is able to discover potentially parallelizable loops that are missed by the state-of-the-art production compilers. Dig et al [11] present *ReLooper*, an Eclipse-based refactoring tool, that performs two important tasks required for refactoring the regular arrays into parallel arrays in Java. The main tasks are analyzing whether the loop iterations are safe for parallel execution and replacing loops with the equivalent parallel operations automatically.

The *SUIF* parallelizer, presented by Wilson *et al.* [12], 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, uni-modular 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 [13] 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. Loops selected will be parallelized and developers are informed about them. In spite of the sophisticated analysis and transformations performed by the PGI compiler, limitations still exist, e.g., innermost loops and timing loops.

Here we do not propose specific solutions to automated parallelization. Rather empirically examine a number of systems to determine what roadblocks exist in the development and enhancement of such tools.

III. PARALLELIZATION INHIBITORS

We now describe the major inhibitors to the parallelization of for-loops. Data dependency is discussed first followed by function calls with side effects, and lastly break and goto statements. In this study, a for-loop is considered a *free-loop* if it does not contain any parallelization inhibitors. That is, a free-loop does not contain any of the following inhibitors: *potential data dependency*, *function calls with potential side effect*, *goto* statements, or *break* statements.

The terms potential data dependency [18, 19] and function call with potential side effects refer to the fact that it can be very difficult to determine if there actually exists a data dependency or side effect. As such techniques to determine data dependency and side effects are generally conservative and label situations as having a data dependency or side effect when in fact there may not be one.

The approaches we use for detecting data dependence and function calls with side effects are not completely accurate but are both conservative. That is, they claim a dependency or side effect when none exists but never miss a dependency or side effect.

A. Data dependency

One of the essential conditions that inhibit for-loop parallelization is data dependency. The order of statement execution within the body of the for-loop must be preserved in many situations to gain the same results as 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 [1, 14].

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 covered very well in literature [15-18]. 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 [18, 19].

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

B:      for (i=1; i<100; i++)
{
  S:       Data2[i] = 0.5 + M[2*i-4];
  T:       M[i*2]   = Data1[i-1]*0.25 ;
}
C:      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 1. Examples of data dependency types.

The main purpose of data dependence analysis techniques is to compare the references of the same array looking for any two identical and equivalent values of the references in different iterations. That is, they detect if the same memory position is used by more then one loop iteration. Indices of the for-loops that are used in iterating and controlling the for-loops are usually used in array references (subscripts). Each array element may use more than one subscript in order to be referenced, as in case of nested loops (e.g., `M[i][j][k]`, Figure 1). The subscripts are usually presented as functions that can be linear or nonlinear. Analysis of nonlinear functions is very complex. The focus of most dependency analysis tests is the references of the arrays where complexity is varied from one to other [1, 20].

We now briefly discuss the various types of data dependency. Also, we explain data dependence 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 auto 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. Figure 1 presents a number of examples of data dependencies that are inhibitors to parallelization. There are three types of dependency based on the way, and sequence of access, of a memory location. These are called *flow-dependence* (aka true dependence), *anti-dependence*, and *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 on the memory [16, 20-22].

Let us say that we have two assignment statements S and T in a for-loop body, and both `S` and `T` reference a memory location `M`, such that one of `S` or `T` is a write access. Also, let us assume that `S` is presented and executed before `T` in the sequential program that being analyzed. Flow-dependence, also known in the literature as True Dependence, occurs when `S` modifies `M` while `T` reads `M`. That is, one statement is dependent on the result of a previous statement. Anti-dependence occurs if `S` reads `M` while `T` modifies it. In other words, when a statement requires a value that is later updated. Output-dependence occurs when the ordering of the statements affect the final result [16, 20].

In Figure 1 for-loop A contains statements `S` and `T` that have a flow-dependency. 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]`. In for-loop B, 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`. In for-loop `C` of Figure 1, `S` is output-dependence on `T`. That happens when `M[i]` in T points to the same memory position that is later used and presented by `M[i-1]` in `S`.

B. *Function Calls with Side Effects*

Another condition that can inhibit parallelization is

```
using System.Xml;
// loading the srcML file
XmlDoc.Load("srcMLfile.xml");

// for-loops extraction from xml file
XmlNodeList oNodeList = XmlDoc.GetElementsByTagName("for");

// while loops extraction
XmlNodeListoNodeListWhiles = XmlDoc.GetElementsByTagName("while");

// extract assignment statements
XmlNodeList nodelistx = XmlDocSub.GetElementsByTagName("expr_stmt");

// get the function calls
XmlNodeList nodelistxMethodCalls = XmlDocSub.GetElementsByTagName("call");

// goto detection
XmlNodeList nodelistxgotos = XmlDocSub.GetElementsByTagName("goto");

// break statement detection
XmlNodeList nodelistxbreaks = XmlDocSub.GetElementsByTagName("break");

// Potential Dependency detection
PotentialDependencyDetector(oNodeList);
```

Figure 2. Collecting for-loops and inhibitors from srcML documents in ParaStat

calling functions or routines with 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[1]. A side effect can be produced by function call in multiple ways.

Basically, any modification of the non-local environment is referred as side effect [23, 24] (e.g., modification of a global variable or by reference passing arguments.) Moreover, a function or a subroutine call in a for-loop or in any function called from the called function may introduce data dependence that might be hidden [25]. The static analysis of the body of the function or the subroutine increases compilation time, hence this is to be avoided. As such, it is usually left to the programmer to ensure that no data dependence exists and the loop is parallelized by explicit markup using an API.

In general, a function or a routine has a side effect if it does any of the following:

- Modifies global variables
- Modifies static variables
- Performs I/O
- Modifies parameters passed by reference
- Calls a function or a routine that has side effects

All of these conditions are applicable to methods of objects in C++. So, a method may cause a side effect that prevents the loop from being parallelized. There are many algorithms proposed for side effect detection [23, 26], each varying in efficiency and complexity. Also, standard functions that are known to not have side effects can be easily identified. In general a conservative approach is normally taken and functions are labeled as having side effects if a complete analysis cannot be done or is too costly.

C. Conditional Exits, Breaks, Gotos

Another inhibitor of for-loop for parallelization is conditional exits, break, and goto statements. The loop must be a basic block, meaning no jumps from inside to outside the loop are permitted. As such the occurrence of one of these statements is a roadblock. However, it is, in general, quite simple to determine the occurrence of these statements and poses little overhead. Also, the same applies to exception handling. Exceptions throws must be caught within the loop body for safe parallelization.

D. Shared and Private Data

There are other inhibitors that can prevent the for-loop from parallelization. Shared data and private data are both inhibitors that must be addressed. In this study we do not consider them since we assume that *OpenMP* is used for loop parallelization. *OpenMP* offers particular directives that can solve these inhibitors. Variables that are shared among all threads can be problematic because if one thread is reading it, another thread may be writing to it. *OpenMP* can solve this problem using special directive for that,

```
#pragma omp parallel for private
        (sharedVariable);
```

Reduction variables can also be an inhibitor. *OpenMP* has a specific directive that solves the problem:

```
#pragma omp parallel for reduction
        (SharedVariable);
```

[1] www.hp.com, www.intel.com

This clause makes the reduction variable shared to generate the correct results, but private to avoid race conditions from parallel execution. Since those types of inhibitors are solvable by *OpenMP*, they are not considered in our study and we consider the loops contain these inhibitors to be free-loops.

TABLE 1. CHARACTERISTICS OF THE ELEVEN OPEN SOURCE SYSTEMS USED IN THE STUDY.

System	Version	for-loops	Files	KLOC
gcc	4.5.3	28,293	40,638	4,029
KOffice	2.3	3,567	4,927	1,185
Subversion	1.6.17	1,437	687	922
Open MPI	1.4.4	5,062	3,606	888
KDELibs	4:3.5.10	4,577	5,000	863
Python	2.5.6	3,738	1,538	695
Ruby	1.8.6-p399	1,258	389	565
OSG	3.0.1	5,722	1,992	503
QuantLib	1.1	4,499	3,398	449
httpd	2.2.17	935	370	391
Chrome	11.0.696	2,797	1,380	174

IV. CASE STUDY

We now study the parallelizability of eleven open source software projects. First we present the approach and our tool, *ParaStat*, for gathering data from the systems. Then we take a close look at one specific system, namely *gcc*. Following is a comparative examination of the parallelizability among the eleven systems.

Table 1 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. Each for-loop is checked to see if it contains any of the inhibitors. As mentioned previously, a for-loop is called a free-loop if it contains no inhibitors.

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 small set of general-purpose open source applications that are widely used. These are the types of systems that are not particularly aimed at parallel architectures but may benefit from that type of hardware. We feel that they represent a reasonable reflection on the types of systems that would undergo reengineering or migration to better take advantage of modern hardware.

A. Data Collection

We developed a tool, called *ParaStat*, which analyzes loops and determines if they contain any inhibitors as defined previously. First, we collect all files with C/C++ source code extensions (i.e., c, cc, cpp, h, hpp, and cxx). Then we use the *srcML toolkit* [27-29] to parse and analyze each file. We use srcML because it is very efficient and flexible to construct specialized static analysis tools. *ParaStat* iteratively analyzes each for-loop for inhibitors and keeps a count of each encountered within each loop. It also counts the number of free-loops found. The final output is a summary report of the percentages of free-loops and for-loops with one or more types of inhibitors. The *srcML toolkit* basically produces an XML representation of the parse tree.

ParaStat analyzes these XML files using System.Xml provided by the .NET Framework. The System.XML

framework provides multiple classes that facilitate the XML processing. Figure 2 presents some examples of XmlDocument class being used to identify every for-loop in the system. The body of each loop is then extracted and examined for each inhibitor via other methods. Assignment statements are then extracted from the for-loop body for deeper analysis in order to detect potential dependency between array items with in the loop body.

There are a number of ways to detect the inhibitors. More accurate methods require much more analysis. *ParaStat* uses the following rules to identify loop inhibitors. If no inhibitors exist in a for-loop it is counted as a free loop otherwise the existence of each inhibitor is tallied. Again, in the study conducted here a conservative approach is used to detect inhibitors.

Potential data dependency. We say there is potential data dependence in a for-loop, if two statements refer to the same array with one of them referencing the array on the left-hand side and the other referencing the array on the right-hand side, or both on the right hand side. Additionally, if the same array appears on both the right- and left-hand sides of a statement, we say that there is potential data dependence. Examples of these situations are given in Figure 3. The techniques presented in this work do not use any data dependence tests to break the detected potential dependence; it just detects and counts the potential data dependence, thus our approach is conservative. This requires very simple analysis and while it will over count this inhibitor it will not miss any.

Function call with potential side effects. Any loop that contains a function call that is not in the list of known "no side effects" standard function is considered to have side effects. While this over counts the inhibitor, most all functions have some type of side effect [34, 35] (with the exception being such things as predicate and get type methods/functions).

Exit, goto, and break statements. All exit, goto, and break statements are found that are in a body of a loop. Any loop containing one of these statements is considered not to be a free loop.

As we said previously, we do not consider shared and private data to be an inhibitor here since there are known solutions for this using *OpenMP*.

B. Detailed Examination of gcc

Before comparing the 11 systems let us take a closer examination of one system, *gcc* version 4.5.3. This system contains 40,683 files, with total of approximately 4.02 MLOC of C/C++.

First, the source code for the entire system was converted into *srcML* format. This took approximately 16 minutes (less time than compilation). Next, all the converted files were analyzed by the tool detecting a total of 28,293 for-loops. These were analyzed in order to detect inhibitors in their bodies. Each assignment statement was analyzed to discover any potential dependency between array elements. At the same time, for-loops were analyzed for any break, goto, or function calls. We excluded calls to standard functions that are known in advance to have no side effects. Analyzing the 28,293 for-loops took a little over 2 minutes. Statistics are produced by the tool and saved.

```
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];
}
```

LeftArraysNames	LeftArraysIndices
M	i*2
Data2	i
Data2	i-1
Data3	Data4[t]
Data4	t

RightArraysNames	RightArraysIndices
Data1	i-1
M	2*i-4
Data3	i+1
Data1	Data2[t-1]
Data2	t-1
Data4	i

Cases that can be detected:

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 3. Examples of potential data dependence detected by ParaStat tool

We focus on three aspects regarding for-loops. First, total percentage of for-loops containing one or more inhibitors. This gives a handle of how much of a system could be readily parallelized by a compiler or other automated tools. Second, we examine what percentages of for-loops have data dependency. We are interested in understanding if the amount of literature on this topic is reflected in actual prevalence of this inhibitor in source code. Third, we seek to understand what are the most prevalent inhibitors to understand how to avoid, resolve, or eliminate their usage.

Table 2 presents the data for all the systems and *gcc* is the first row. We notice that the number of for-loops (28,293) is approximately four times the number of while-loops (6,856). Around 25% of the loops were determined to be free-loops and good candidates for automated parallelization. The number of inhibitors found in for-loop is also presented.

Figure 4 presents a summary of the percentages of each type of inhibitor for *gcc*. The percentages in Figure 4 are the number of for-loops with only one type of inhibitor (over the total number of for-loops). 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.

TABLE 2. DATA FOR EACH SYSTEM INCLUDING THE NUMBER OF LOOPS AND INHIBITORS FOUND USING OUR TOOLS

System	for loop		while loop		free loop		function call	break	data dependency	goto
gcc	28,293	80%	6,856	20%	8,726	25%	17,815	3,207	2,575	508
KOffice	3,567	74%	1,230	26%	187	4%	3,316	261	471	3
SVN	1,437	75%	473	25%	118	6%	1,285	167	28	26
OpenMPI	5,062	74%	1,770	26%	1,013	15%	3,616	909	506	330
KDELibs	4,577	65%	2,454	35%	373	5%	4,131	569	162	41
Python	3,738	66%	1906	34%	726	13%	2796	810	240	500
Ruby	1,258	56%	995	44%	276	12%	907	226	78	70
OSG	5,722	82%	1,292	18%	505	7%	5,065	308	424	11
Quantlib	4,499	91%	472	9%	554	11%	3,658	177	945	15
httpd	935	53%	827	47%	217	12%	657	209	41	18
Chrome	2,797	78%	785	22%	406	11%	2,113	429	532	81

We found that only a small percentage of loops in *gcc* have only a data dependency inhibitor (3.3%). As can be seen, the occurrence of function calls is by far the most prevalent inhibitor in for-loops (48.4%). It is also interesting to note that conditional breaks and gotos are used in for-loops even though it is considered a bad programming practice [30].

Figure 5 presents the findings in a bit different perspective. This presents the percentage of for-loops that contain any inhibitor. So 63% of all for-loops contain a function call and 9% contain a data dependency. 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 clear that function calls present the most serious problems to parallelization. Moreover, while there is much literature devoted to addressing the problems of data dependencies it appears that resolving that problem will have a limited impact comparatively.

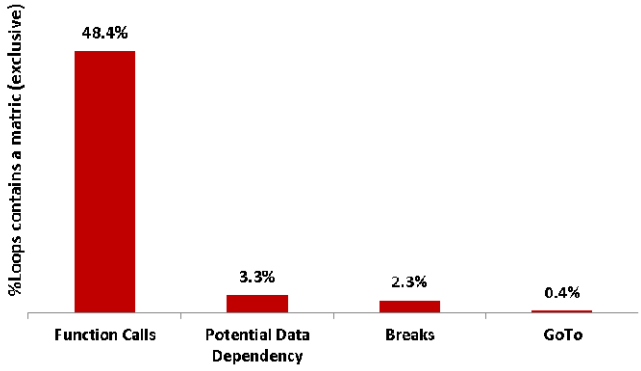

Figure 4. Percentage of for-loops in gcc that contain only one type of inhibitor. The remaining for-loops are either free or have more than one type of inhibitors

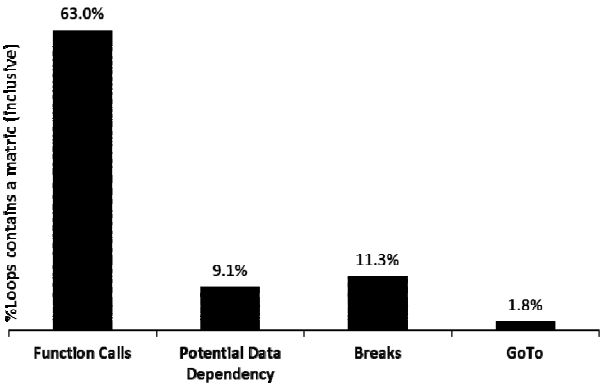

Figure 5. Percentage of for-loops in gcc that contain at least one inhibitor, if a for-loop has two inhibitors it counts in both categories

C. Comparison Over Eleven Systems

We now examine all eleven open source systems. A list of these projects was given previously in Table 1 and Table 2 presents the collected numbers and percentages of interest. The first two columns, shows the distribution of for-loops and while-loops for each system. With the exception of *Httpd* and *Ruby* all of the systems, show a much larger use of for-loops. The third column gives the count of free loops (those loops easily parallelizable by embedding *OpenMP* directives). Free loops account for around 10% of all loops (for and while) in these systems except in the case of *gcc* (25%) and *Open MPI* (15%). Since while loops do not have fixed iteration bounds, they are all un-parallelizable (baring some refactoring in certain cases).

The remainder of the table presents the counts of each inhibitor in for-loops. Function calls is by far the most prevalent across all the systems. This is then followed by data dependency or break. Goto statements are consistently the least prevalent.

TABLE 3. 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

System	function call	break	data dependency	goto	Free for-loops
gcc	48.4%	2.3%	3.3%	0.4%	30.8%
KOffice	75.7%	0.3%	1.5%	0%	5.2%
Subversion	76.9%	1.8%	0.5%	0.1%	8.2%
Open MPI	49.1%	5.1%	2.7%	0.9%	20.0%
KDELibs	90.3%	12.4%	3.5%	0.9%	8.1%
Python	45.8%	3.4%	3.5%	0.6%	19.4%
Ruby	52.2%	3.0%	2.1%	0. 6%	21.9%
OSG	78.7%	0.3%	2.2%	0.03%	8.8%
QuantLib	63.0%	0.7%	5.8%	0.1%	12.3%
httpd	50.1%	4.6%	1.3%	0.2%	23.2%
Chrome	53.8%	2.3%	7.0%	0.1%	14.5%
Average	**62.2%**	**3.3%**	**3%**	**0.36%**	**15.7%**

Table 3 presents the percentage of for-loops that contain only one type of inhibitor for each category. We see that *OSG* has the largest percentage of function calls, 78.7% followed by *Subversion* (76.9%). *Python* and *gcc* have the lowest, 44.6% and 48.4% respectively. The percentage of the for-loops that contain only a potential data dependency across all the systems is quite small by comparison. *Chrome*

has the large percentage of data dependencies while the smallest belongs to *Subversion*. The use of goto statements is not common for any of the systems.

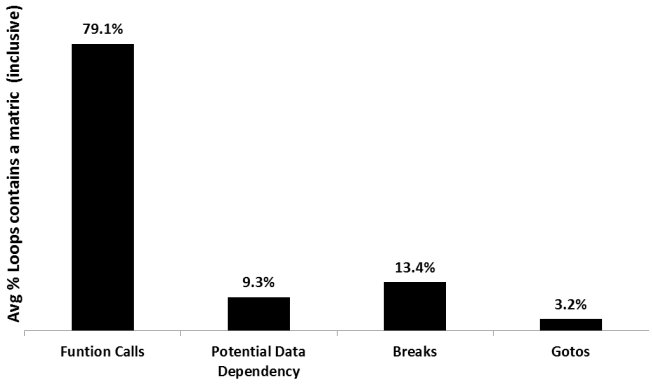

Figure 6. Average percentage of for-loops for all 11 systems that contain at least one inhibitor.

Figure 6 presents the average percentage, over all 11 systems, of for-loops that contain any one of the inhibitors. We see that the trend is likewise similar via this perspective. Function calls are by far most prevalent. However, on average we see the next most prevalent inhibitor is the break statement followed closely by data dependencies.

V. Using Parallelizability Over Time

We now examine how the presences of for-loop inhibitors changed over a 10-year period for a subset of these systems. The history of 10 of the 11 systems is presented. Chrome is relatively a new project and has a very small version history. As such it was exclude from this comparison. The change in the number of for-loops, free-loops, and presences of each inhibitor was computed for each version. These values were aggregated for each year so the systems could be compared on a yearly basis. We update each system to the last revision for each year. At which point we examined all files with source code extensions (i.e., *c*, *cc*, *cpp*, *h*, *hpp*, and *cxx*) and then extract all for-loops in those files. Our goal is to uncover how each system evolves in the context of parallelization. Ideally, this could lead to recommendations for utilizing and adapting to the current multicore processing trends.

Figure 7 presents the change in the percentage of free-loops for each of the 10 systems. During the 10-year period only one system, *gcc*, appears to have a steady and sizable increase in the number of free-loops. The other systems show a fairly flat trend during the duration. *KOffice* seemed to have improved but then ended the period with approximately the same percentage.

In the case of *gcc* we assume that the development team is quite aware of code optimization techniques. Parallelizability over processors is a major optimization factor for faster compilers under the *gcc* GNU tool-chain. This most likely explains, in part, the consistent increase in free-loops.

Figure 8 presents the percentage of for-loops that contain a function call inhibitor. It is approximately a mirror of Figure 7. That is, the systems that increased in the number of free-loops decreased in the number of function call inhibitors (e.g., *gcc*) or vice-versa (e.g., *OpenMpi*).

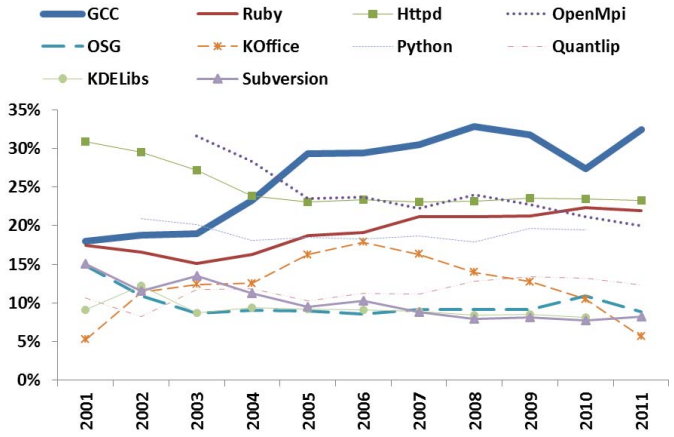

Figure 7. The percentage of Free-loops evolution over ten years for the ten systems

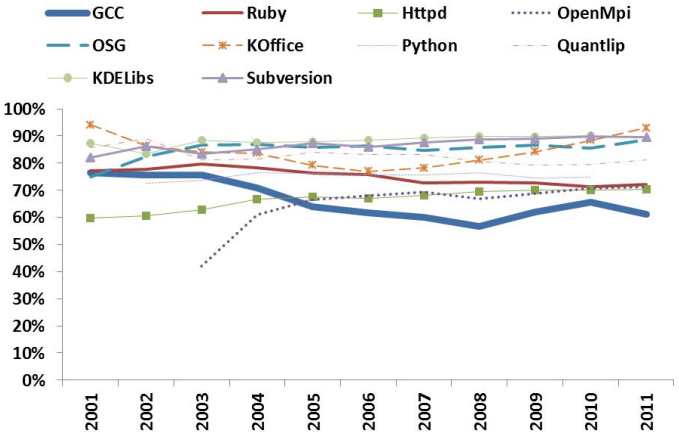

Figure 8. The percentage of function calls inhibitors occurring in for-loops over ten years for the ten systems

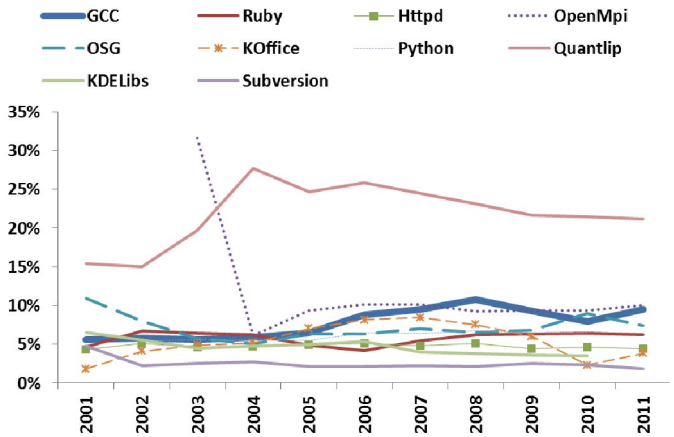

Figure 9. The percentage of potential data dependency inhibitors occurring in for-loops over ten years for the 10 systems

Figure 9 presents the results for data dependency inhibitors. We see a flat trend across all systems except for a large drop early on for *OpenMpi* (most likely the result of optimizations after an initial release). Figure 10 and Figure 11 presents the results for goto and break inhibitors, respectively.

The trends for the presences of goto and break statements are similarly flat over the 10-year duration and no interesting drastic changes were observed. In Figure 10 we have the plot for the use of goto statements. For the most part the usage of goto statements over the lifetime of the systems is fairly flat. Some exhibit a small variation of around 2% and *Python* has more occurrences that the others.

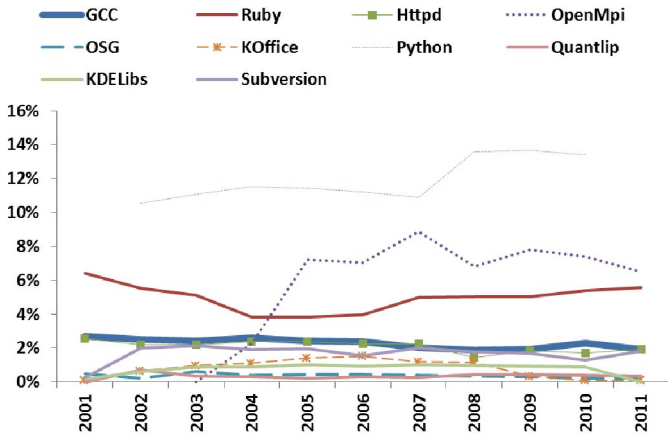

Figure 10. The percentage of goto inhibitors occurring in for-loops over ten years for the 10 systems

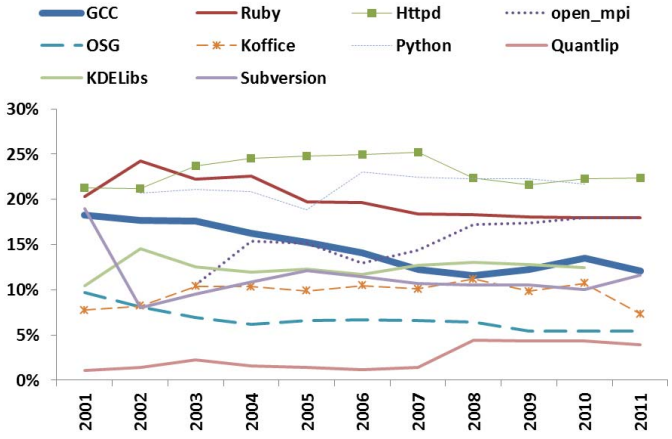

Figure 11. The percentage of break inhibitors occurring in for-loops over ten years for the 10 systems

VI. DISCUSSION

Our main observation is that while the literature has been heavily focused on solving and detecting data dependencies [19, 31, 32], our empirical data shows that function calls within loops are the greatest roadblock to parallelization. In fact, the vast majority (63%) of for-loops with inhibitors are those with a function call inhibitor.

Moreover, it appears that methods to remove breaks and gotos could potentially have a greater impact on improving parallelizability then data dependency (3% versus 3.7% on average).

If tools to support the automation of parallelization are to take advantage of current hardware, they must be highly focused on dealing with function call inhibitors. Additionally, developing methods to remove break and goto statements from for-loops is also a worthwhile investment.

Our more detailed examination of *gcc* resulted in a number of interesting observations. For example, *gcc* has the largest percentage of free-loops. It also appears (Figure 7 and Figure 8) the developers of *gcc* are well aware of parallelization inhibitors and are systematically refactoring the code to reduce their numbers. Additionally, it seems that *gcc* is the only system, of the ones we examined, that is increasing the number of free-loops and decreasing the number of inhibitors. This is not particularly surprising given that *gcc* is supporting features for automatically optimizing code for parallelization [33]. We also point out that both *Ruby* and *Python* (interpreters) have a relatively high number of free-loops, 22% and 19%, respectively. The developers appear to be practicing these idioms on their own code base.

Given this, *gcc* could be used as a model, or starting point, in the development of refactorings to assist in inhibitor removal. There appears to be a number of good practices towards the goal of writing code that can be parallelized. There are few pedagogical approaches that highlight these types of coding techniques.

Figure 8 is particularly telling. Few systems show any systematic decrease in the number of inhibitors over time. Clearly, this is not a priority for these development teams. Alternatively, there are few documented means to decrease inhibitors and as such not much work on the problem is taking place.

The study also shows that developers are still using goto statements even while it has been considered a harmful statement [30]. For example, about 10% of the for-loops in *Python* contain a goto statement. *Open MPI* and *Ruby* have 7% and 6%, respectively. While there are appropriate times that a careful use of goto can be practical, we lose the opportunity for parallelizing those for-loops.

There are a number of practical ways one can deal with inhibitor other than data dependency. A function call is only harmful if it has a side effect (as presented in section 3). Analysis methods can be used to document functions and methods that have any type of side effect. Approaches such as labeling methods with stereotypes [34] 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 need to avoid analyzing the functions called within the bodies of for-loops that are considered for automatic parallelization. Upfront function analysis could greatly increase compiling 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.

VII. 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 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 assumed function calls have a potential side effect. Again, this is a very conservative approach. Additional analysis is required to determine a more accurate picture of if a function call results in a side effect that would inhibit parallelizability. This is of concern to the findings and observations. If extremely large portions of function calls have no side effects, then their overall impact as an inhibitor could be reduced nearer to the level of data dependencies or breaks. However, we have seen in previous studies [34, 35] that the majority of functions appear to have side effects of one type or another. As such, while more careful analysis will most likely reduce the number of for-loops with function calls as inhibitors, the number will still greatly outweigh those of the other inhibitor types. We are working to extend our analysis and check this issue more carefully.

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 *OpenMPI*.

VIII. Conclusion and Future work

This study empirically examined that potential parallelizability of 11 open source software systems. We found that the greatest inhibitor to automated parallelization is the presence of function calls with potential side effects occurring in for-loops. 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. From what we have observed, in this relatively small study, is that more attention needs to be placed on dealing with function call inhibitors if a large amount of parallelization is to occur in the studied systems. While we cannot completely generalize this finding to all software systems (across all domain) there is some indication that this is a common trend. A more comprehensive study on open and close source systems is needed to fully support this hypothesis.

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 plan to investigate further the development process of *gcc* in an attempt to decipher any valuable idioms or lessons for parallelization.

References

[1] K. Psarris and K. Kyriakopoulos, "Data Dependence Testing in Practice," presented at the Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, 1999.

[2] A. J. C. Bik and D. Gannon, "Automatically Exploiting Implicit Parallelism in Java," *Concurrency - Practice and Experience,* pp. 579-619, 1997.

[3] B. Barney. (2012). *Introduction to Parallel Computing*. Available: https://computing.llnl.gov/tutorials/parallel_comp/#Models

[4] D. S. Nikolopoulos, C. D. Polychronopoulos, E. Ayguad´e, J. u. Labarta, and T. S. Papatheodorou, "The Trade-off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms," presented at the ICS '01, Sorrento, Italy, 2001.

[5] N. Nadgir. (2001). *Using OpenMP to Parallelize a Program*. Available: http://developers.sun.com/solaris/articles/openmp.html

[6] Intel. (2010). *Automatic Parallelization with Intel® Compilers*. Available: http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/

[7] D. A. P. J.L. Hennessy "Computer Architecture: A Quantitative Approach," *Morgan Kaufman Publishers, San Francisco,* 2006.

[8] Intel. (2010). *Automatic Parallelization with Intel Compilers*. Available: http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/

[9] L. Feng. (2009). *Automatic parallelization in Graphite*. Available: http://gcc.gnu.org/wiki/Graphite/Parallelization

[10] M. KIM, H. KIM, and C.-K. LUK, "Prospector: A dynamic data-dependence profiler to help parallel programming," *In 2nd USENIX Workshop on Hot Topics in Parallelism , HotPar '10,* 2010.

[11] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson, "Relooper: refactoring for loop parallelism in Java," presented at the Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, Orlando, Florida, USA, 2009.

[12] R. S. F. Robert P. Wilson , Christopher S. Wilson , Saman P. Amarasinghe , Jennifer M. Anderson , Steve W. K. Tjiang , Shih-wei Liao , Chau-wen Tseng , Mary W. Hall and Monica S. Lam , John L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," *ACM SIGPLAN Notices,* vol. 29, pp. 31--37, 1994.

[13] T. P. Group. (2012). *Parallel Fortran, C and C++ Compilers* Available: http://www.pgroup.com/products/pgicdk.htm

[14] A. Orso, S. Sinha, and M. J. Harrold, "Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging," *ACM Transactions on Software Engineering and Methodology,* vol. 13, pp. 199-239, 2004.

[15] X. Kong, D. Klappholz, and K. Psarris, "The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization," *IEEE Transactions on Parallel and Distributed Systems,* pp. 342 - 349 1991.

[16] U. K. Banerjee, "Dependence Analysis for Supercomputing," *Kluwer Academic Publishers,* 1988.

[17] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," presented at the Proceedings of the 1991 ACM/IEEE conference on Supercomputing, Albuquerque, New Mexico, United States, 1991.

[18] P. M. Petersen and D. A. Padua, "Static and Dynamic Evaluation of Data Dependence Analysis Techniques," *IEEE Trans. Parallel Distrib. Syst.,* vol. 7, pp. 1121-1132, 1996.

[19] P. M. Petersen and D. A. Padua, "Static and dynamic evaluation of data dependence analysis," presented at the Proceedings of the 7th international conference on Supercomputing, Tokyo, Japan, 1993.

[20] T. Jacobson and G. Stubbendieck, "Dependency Analysis Of For-Loop Structures For Automatic Parallelization Of C Code," 2003.

[21] D. Kulkarni and M. Stumm, "Loop and Data Transformations: A Tutorial," Univerity of Toronto, Technical Report1993.

[22] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, 2002.

[23] D. A. Spuler and A. S. M. Sajeev, "Compiler Detection of Function Call Side Effects," Townsville, Queensland, Australia, Technical Report1994.

[24] C. Ghezzi and M. Jazayeri, *Programming language concepts*: Wiley, 1982.

[25] Oracle. (2010). *Subprogram Call in a Loop*. Available: http://docs.oracle.com/cd/E19205-01/819-5262/aeuje/index.html

[26] J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," presented at the Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, 1979.

[27] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An XML-Based Lightweight C++ Fact Extractor," presented at the Proceedings of the 11th IEEE International Workshop on Program Comprehension, 2003.

[28] M. L. Collard, Maletic, J. I., and Marcus, A, "Supporting Document and Data Views of Source Code," *in Proceedings of ACM Symposium on Document Engineering,* p. 8, 2002.

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

[30] E. Dijkstra, "Go to statement considered harmful," in *Classics in software engineering*, ed: Yourdon Press, 1979, pp. 27-33.

[31] P. M. Petersen and D. A. Padua, "Experimental Evaluation of Some Data Dependence Tests " Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, Technical Report1991.

[32] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," presented at the Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, Toronto, Ontario, Canada, 1991.

[33] D. Novillo, "OpenMP and automatic parallelization in GCC," presented at the Proceedings of the GCC Developers, Ottawa, Canada, 2006.

[34] N. Dragan, M. L. Collard, and J. I. Maletic, "Using method stereotype distribution as a signature descriptor for software systems," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 567-570.

[35] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic identification of class stereotypes," presented at the Proceedings of the 2010 IEEE International Conference on Software Maintenance, 2010.