An Empirical Examination of the Prevalence of Inhibitors to the Parallelizability of Open Source Software Systems

Saleh M. Alnaeli,

Jonathan I. Maletic, Ph.D.

Michael L. Collard, Ph.D.

Published online: …... ……..…….
© Springer Science …….……
Editors: …… and ……

Abstract An empirical study is presented 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 paper 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.

Keywords Parallelization inhibitors, data dependency, function calls, function pointers, virtual functions, empirical study

---

S.M. Alnaeli
Kent State University, Kent, OH 44240, USA
e-mail: salnaeli@kent.edu
J.I. Maletic, Ph.D
Kent State University, Kent, OH 44240, USA
e-mail: jmaletic@kent.edu
M.L. Collard, Ph.D.
The University of Akron, Akron, OH, USA
e-mail: collard@uakron.edu

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.

The process of parallelizing a software system 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++ 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.

Here we have undertaken an empirical examination of a variety of general-purpose open source software systems to better understand the roadblocks for automated and/or semi-automated parallelization tools. We are particularly interested in 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.

The study consists of examining thirteen C/C++ systems across various application domains that total over 14 MLOC. For each system, the number of loops is counted (for- and while-loops) and each for-loop is statically analyzed to detect if any parallelization inhibitors are present. A count of each of the inhibitor types is tabulated and this data is then presented to compare the different systems and uncover trends, and make other general observations. Furthermore, the history of each system is examined and the number of inhibitors is calculated for each release.

We are specifically interested in addressing the following questions. How many loops in these systems do not have any inhibitors? Which types of inhibitors are most frequent? There are often multiple inhibitor types in a single for-loop. Understanding which inhibitors occur alone in loops is also relevant. Since data dependency is a major topic in the literature, we want to understand how often this inhibitor occurs in general purpose software applications. Since calls using function pointers or to virtual method pose a serious problem for conducting static analysis, we examine the prevalence of these types of calls as potential inhibitors. Lastly, we want to see if the numbers or distributions of inhibitors change over the history of a software system? We formulate these as research questions in a later section.

This work extends our previous research (Alnaeli, Alali and Maletic 2012) on this topic in a number of ways. Most importantly, we have greatly increased the accuracy of the analysis to detect if a function call has actual side effects. This improvement has a very large impact on the results. We also extended the analysis of calls via function pointers and virtual methods, as this is often a great concern while conducting static analysis. Lastly, the study has been expanded to additional systems.

This work contributes in several ways. First, it is one of the only large studies on the potential to parallelize general-purpose software applications. Our findings show that function calls represent the vast majority of

inhibitors occurring in these systems. Our study also shows that function calls via function pointers and virtual methods have only a minor impact on the parallelization process and a conservative approximation solution can be considered. Moreover, the findings show that, for the most part, few systems have increased their potential to take advantage of the new hardware over their history. This knowledge will assist researchers in formulating and directing their work to address this problem.

The remainder of this paper is organized as follows. Section 2 presents a definition of the problem and related work on the topic of automatic parallelization and its challenges. Section 3 describes the methodology we used in the study along with the different inhibitors to parallelization along with how we performed the analysis to identify each inhibitor. Section 4 presents the findings of our study of 13 open source software systems. There is a discussion of results in section 5 followed by threats to validity and conclusions.

2 BACKGROUND AND RELATED WORK

In this study, implicit parallelism with the shared memory parallel model (Bik and Gannon 1997; Barney 2012) is considered and we intend to support the parallelization of 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 compilers support it (Nadgir 2001; Nikolopoulos, Polychronopoulos, Ayguad´e, Labarta and Papatheodorou 2001). 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 contain 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 that while-loops are not parallelized.

Current parallelization's APIs have common for-loops 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 and Kyriakopoulos 1999) even though, as we will show, there are many other inhibitors that appear to be more frequent (Intel 2010). 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 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 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 2010) and gcc (Feng 2009), 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.

Tools separate from production compilers have been developed to assist in these situations where the parallelizing compiler cannot assist. Kim et al. (Kim 2010) introduced Prospector, a profile-based parallelism identification tool using 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, Tarce, Radoi, Minea and Johnson 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 (Calder and Grunwald 1994; Shah Anand 1995; Bacon and Sweeney 1996; Grove, DeFouw, Dean and Chambers 1997; Muth and Debray 1997; Sundaresan, Hendren, Razafimahefa, Vall, #233, e-Rai, Lam, Gagnon and Godin 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 and Hwu 2000) 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 1995) 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. [14] 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 [16](Dean, Grove and 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 interprocedural analysis, and show how these systems evolve over time in terms of analysis difficulty based on the usage of virtual functions and function pointers.

3 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 and Padua 1993; Petersen and Padua 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 and 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 2002; Collard, Kagdi and Maletic 2003; Collard, Decker and Maletic 2011) 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 analyze 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 each of the different inhibitors in detail. We also describe the how the ParaStat tool finds and counts each inhibitor along with any limitations of the approach.

```
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];
      }
```

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

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 (Psarris and Kyriakopoulos 1999; Orso, Sinha and Harrold 2004).

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 and Psarris 1991; Pugh 1991; Petersen and 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 and Padua 1993; Petersen and Padua 1996).

The main purpose of data-dependence 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 (Psarris and Kyriakopoulos 1999; Jacobson 2003).

```
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];
}
-----------------------------------------------

LeftArrayNames          LeftArrayIndices (subscripts)
---------------          -----------------
M                             i*2
Data2                         i
Data2                         i-1
Data3                         Data4[t]
Data4                         t
constRef                      7

RightArrayNames         RightArrayIndices (subscripts)
----------------         ------------------
Data1                         i-1
M                             2*i-4
Data3                         i+1
Data1                         Data2[t-1]
Data2                         t-1
Data4                         i
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
```

Fig. 2 Example with potential data dependencies detected by ParaStat

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 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.

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; Kulkarni 1993; Kennedy 2002; Jacobson 2003). **Fig 2** shows examples of data dependency types that can be detected by ParaStat.

Fig 1 presents examples of the three different types. 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* (**Fig 1** 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* (**Fig 1** 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* (**Fig 1** 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.

Here 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 **Fig 3**.

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 **Fig 3**) 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 **Fig 3**. 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. Also, there are always situations where determining if an actual data dependency exists is computationally impractical.

```
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	LeftArrayIndices (subscripts)
array1	i
array2	k
array3	j*z
constRef	7

RightArrayNames	RightArrayIndices (subscripts)
array1	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

Fig. 3 Example of the limitations of ParaStat 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.

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 and Jazayeri 1982; Spuler and 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 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 and 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

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 (int *ptr; 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 ad(Mock, Atkinson, Chambers and Eggers 2005) dress 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 and Eggers 2005). As such, our approach to detection of side effects is not completely accurate in the presences of pointer aliasing. However, this type of limited static pointer analysis has shown (Alomari, Collard, Maletic, Alhindawi and Meqdadi 2014) to produce very good results on large open source systems

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. This reduced call graph is then used to propagate any side effect detected among all callers of the function.

By conducting this static analysis of function calls within for-loops, we produced more accurate result than our previous study (Alnaeli, Alali and Maletic 2012). 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 to 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.

3.2.2 Dealing with Function Pointers & Virtual Methods

It is very challenging to statically analyze programs that make function calls using function pointers (Shah Anand 1995; Cheng and Hwu 2000) and virtual methods (Bacon and Sweeney 1996). 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 (Zhang and Ryder 1994; Shah Anand 1995; Cheng and Hwu 2000).

```
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
```

Fig 4 Examples of function pointers detected by ParaStat.

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 and Hendren 1994; Zhang and Ryder 1994). For safe analysis and parallelism, the set of possible targets of a function pointer call must be determined. If one item in this set has a side effect, parallelization may not be safe resulting in the conservative decision to inhibit parallelization (Bacon and Sweeney 1996).

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 **Error! Reference source not found.** and the data is summarized in TABLE **5**. 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 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 these 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 has 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 1995). Our tool, *ParaStat*, detects all of these types of function pointers whenever they are present in a for-loop. **Fig 4** 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.

```
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 << obj1.action();

    for(int i = 0; i < 10; ++i)
        cout << obj2.action();
}
```

Fig 5 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

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 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 **Fig 5**, 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.

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 for safe 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 study 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.

3.5 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.

TABLE 1. THE 13 OPEN SOURCE SYSTEMS USED IN THE STUDY.

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

4 FINDINGS OF THE STUDY

We now study the parallelizability of thirteen open-source software projects. 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 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 1 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.

4.1 Design of the Study

Our study focuses on four aspects regarding 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 the 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 or 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 2. 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%

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

4.2 Percentage of Free-Loops

TABLE 2 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 Inhibitor Distribution

We now address R3 and R3a and present the details of our findings on the distribution of inhibitors. TABLE 2 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 much 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).

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 with Side Effects	Jumps	Data Dependency
gcc	22%	9%	8%
KDELIBS	21%	8%	3%
KOffice	25%	7%	3%
Subversion	58%	7%	1%
Open MPI	27%	10%	5%
LLVM	29%	7%	2%
Python	23%	12%	4%
Ruby	35%	9%	4%
OSG	20%	4%	6%
QuantLib	12%	3%	16%
httpd	15%	17%	4%
Chrome src14	39%	8%	4%
Xapain	26%	6%	6%
Average	**27%**	**8%**	**5%**

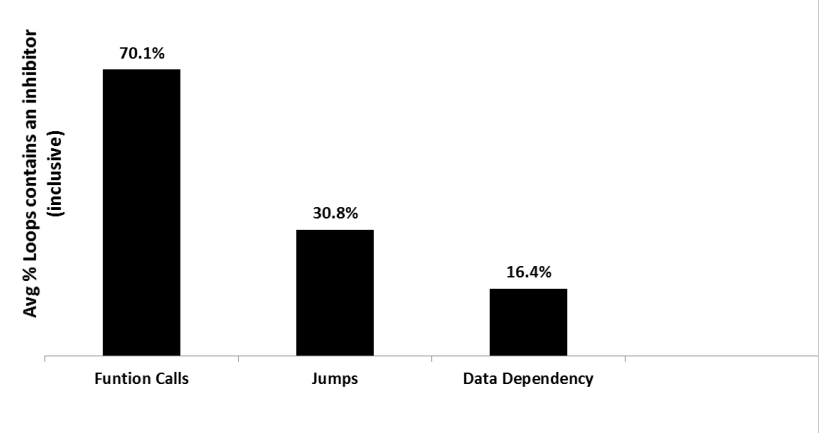

Fig 6. The average percentage of non-free-for-loops that contain at least one inhibitor of the given type, over all 13 systems. Loops may contain more than one inhibitor type.

Error! Reference source not found.**6** 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 jumpsfollowed next by data dependency.

Additionally, in TABLE **4** we compare the results of our previous work (Alnaeli, Alali and Maletic 2012) 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. COMPARISON ASSUMING ALL FUNCTION CALLS HAVE A POTENTIAL SIDE EFFECT AS IN (ALNAELI, ALALI AND MALETIC 2012) AND DOING MORE COMPLETE ANALYSIS TO DETERMINE ACTUAL SIDE EFFECT. THE RESULTS ON THE PERCENTAGE OF FREE-LOOPS IS ALSO GIVEN FOR EACH APPROACH.

System	Functions Call with Side Effect		Free Loops	
	Actual	Potential	Actual	Potential
gcc	28%	48%	48%	31%
KDELIBS	26%	90%	62%	8%
KOffice	38%	76%	62%	5%
Subversion	68%	77%	25%	8%
Open MPI	43%	49%	41%	20%
Python	43%	46%	41%	19%
Ruby	47%	52%	39%	22%
OSG	23%	79%	67%	9%
QuantLib	16%	63%	65%	12%
httpd	22%	50%	57%	23%
Chrome src14	46%	54%	41%	15%
Average	**36%**	**62%**	**50%**	**16%**

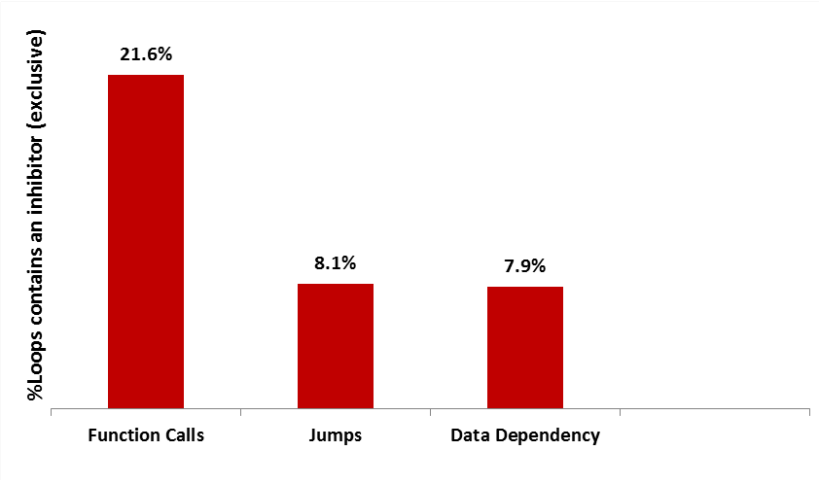

Fig. 7 Percentage of for-loops in gcc that contain only a single type of inhibitor. The remaining for-loops are either free or have multiple types of inhibitors.

Now let us examine one system, *gcc*, in a bit more detail. **Fig 7** 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* only contain data dependency inhibitors. As can be seen, the occurrence of function-call inhibitor is by far the most prevalent inhibitor (21.6 %). **Fig 8** presents the findings in 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.

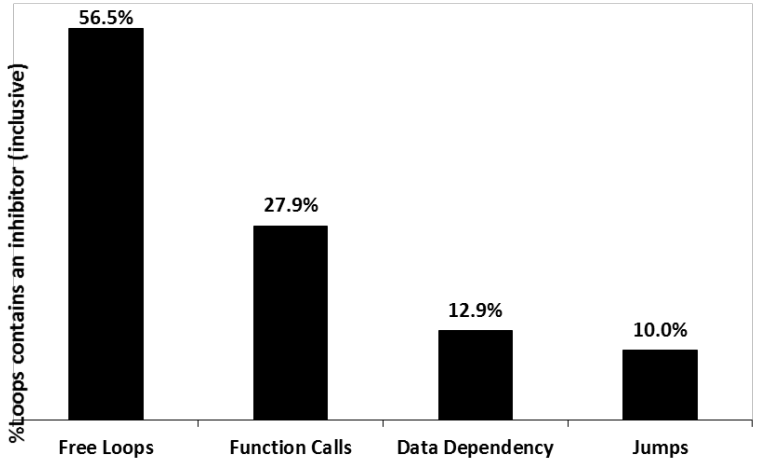

Fig. 8 Percentage of for-loops in *gcc* that contain at least one inhibitor. If a for-loop has two inhibitors it counts in both categories.

4.4 Calls to Function Pointers/Virtual Methods

To address R3b, all thirteen systems were examined with respect to function pointer and virtual method usage. TABLE **5** 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 1995). 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 **5** 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.

TABLE 5 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.

System	Number of Function Pointers	For-loops with Function Pointer Inhibitor	
gcc	2042	511	2%
KDELIBS	255	1	<1%
KOffice	13	0	0%
Subversion	479	22	2%
Open MPI	1587	163	3%
LLVM	135	15	<1%
Python	533	10	<1%
Ruby	266	28	2%
OSG	59	14	<1%
QuantLib	190	23	<1%
httpd	191	11	1%
Chrome src14	5330	736	4%
Xapain	96	6	1%
Average			**1%**

TABLE 6 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.

System	Number of Virtual Func-tions	For-loops with Virtual Function Inhibitor	
gcc	32978	2	<1%
KDELIBS	1337	116	2%
KOffice	1141	128	3%
Open MPI	0	0	0%
LLVM	1374	37	<1%
OSG	1420	123	2%
QuantLib	166	10	<1%
Chrome src14	1168	11	<1%
Xapain	342	1	<1%
Average			**1%**

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 **6** 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 over time of for-loop inhibitor usage.

4.5 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 relatively a 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.

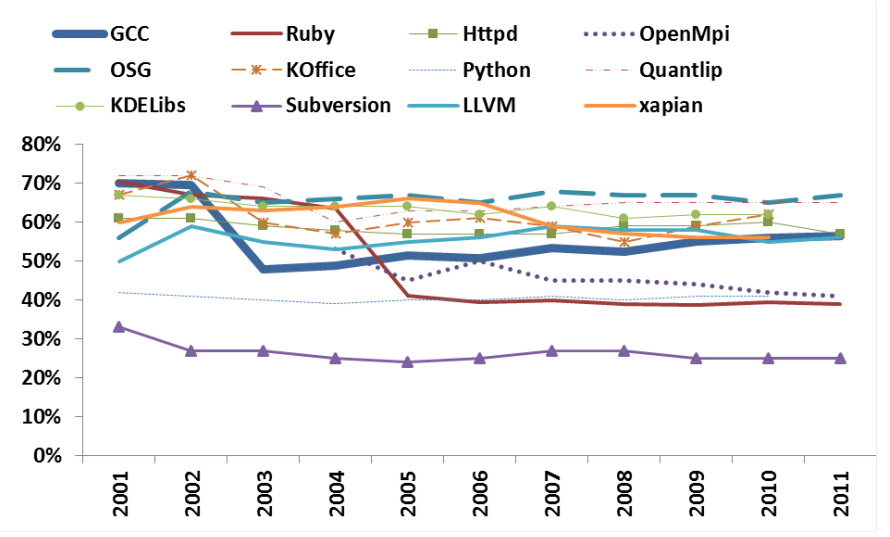

Fig 9 The evolution of the percentage of free-loops over a ten-year period for the ten systems.

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.

Fig 9 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 during the duration. Two systems, *gcc* and *Ruby*, have a steep decline early on

and then are relatively flat in proceeding years. **Fig 10** presents the percentage of for-loops that contain a function-call inhibitor. It is approximately a mirror of **Fig 9**. That is, the systems that increased in the number of free-loops decreased in the number of function call-inhibitors (e.g., gcc, Ruby).

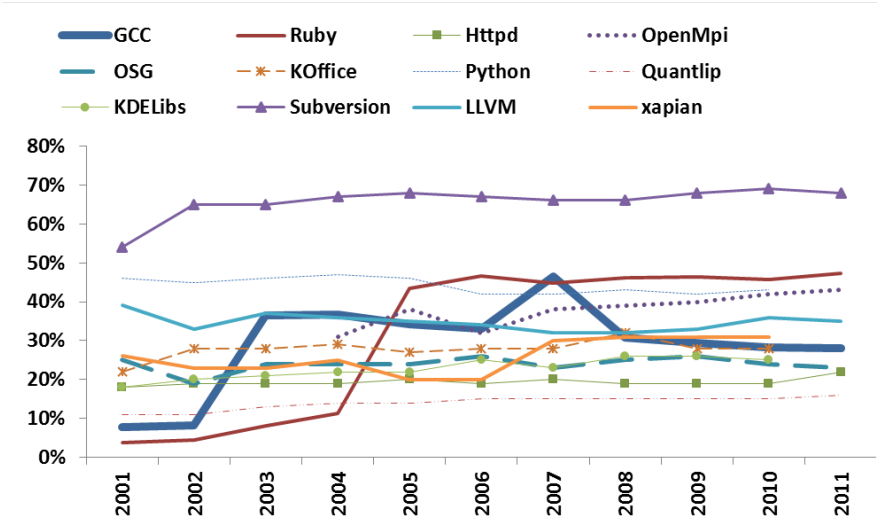

Fig 10. The percentage of function-call inhibitors of for-loops over a ten-year period for the ten systems.

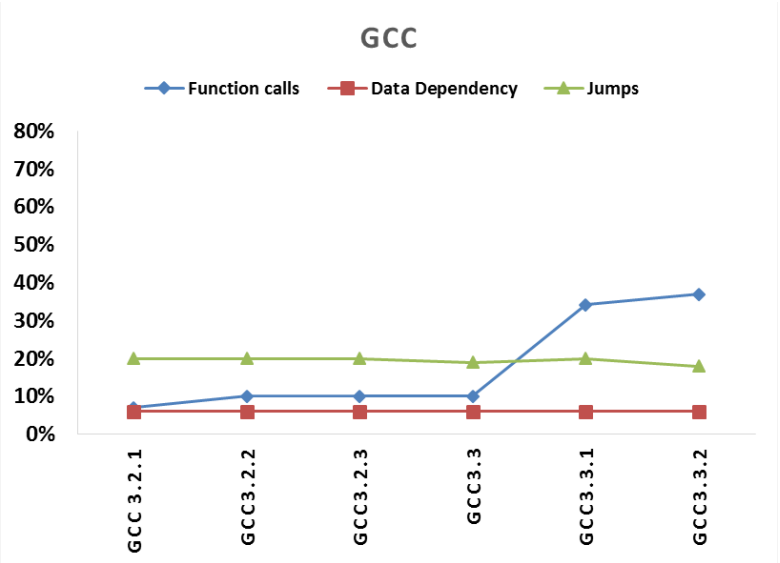

Fig 11. The percentage of different inhibitors of for-loops for releases during the time period of 2001 – 2003 for *gcc*. Note that while for-loops with function-call inhibitors increased greatly, other inhibitors do not show any increase for *gcc*.

As can be observed in **Fig 10**, *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. **Fig 11** 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).

5 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. 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 and Lam 1991; Petersen 1991; Petersen and Padua 1993), the empirical findings show that function calls within loops are the greatest roadblock to parallelization of general purpose applications. In fact, we see that (**Fig 6**) 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. But 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 then 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 7. None of the systems' history demonstrates any systematic decrease in the number of inhibitors over time. 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 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.

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 7. 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.

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

6 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 and Maletic 2009; Dragan, Collard and 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.

7 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).

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

8 ACKNOWLEDGEMENTS

This work was supported in part by a grant from the US National Science Foundation CNS 13-05292/05217.

9 REFERENCES

Alnaeli, S. M., A. Alali and J. I. Maletic (2012). Empirically Examining the Parallelizability of Open Source Software System. Proceedings of the 2012 19th Working Conference on Reverse Engineering.

Alomari, H. W., M. L. Collard, J. I. Maletic, N. Alhindawi and O. Meqdadi (2014). "srcSlice: very efficient and scalable forward static slicing." Journal of Software: Evolution and Process: n/a-n/a.

Bacon, D. F. and P. F. Sweeney (1996). "Fast static analysis of C++ virtual function calls." SIGPLAN Not. **31**(10): 324-341.

Bacon, D. F. and P. F. Sweeney (1996). Fast static analysis of C++ virtual function calls. Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. San Jose, California, USA, ACM**:** 324-341.

Banerjee, U. K. (1988). "Dependence Analysis for Supercomputing." Kluwer Academic Publishers.

Banning, J. P. (1979). An efficient way to find the side effects of procedure calls and the aliases of variables. Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. San Antonio, Texas, ACM**:** 29-41.

Barney, B. (2012). "Introduction to Parallel Computing." from https://computing.llnl.gov/tutorials/parallel_comp/#Models

Bik, A. J. C. and D. Gannon (1997). "Automatically Exploiting Implicit Parallelism in Java." Concurrency - Practice and Experience: 579-619.

Bliss, N. (2007). "Addressing the Multicore Trend with Automatic Parallelization." Lincoln Laboratory Journal **17**: 12.

Calder, B. and D. Grunwald (1994). Reducing indirect function call overhead in C++ programs. Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Portland, Oregon, USA, ACM**:** 397-408.

Cheng, B.-C. and W. Hwu (2000). An Empirical Study of Function Pointers Using SPEC Benchmarks. Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing, Springer-Verlag**:** 490-493.

Collard, M. L., M. J. Decker and J. I. Maletic (2011). Lightweight Transformation and Fact Extraction with the srcML Toolkit. Source Code Analysis and Manipulation (SCAM). Williamsburg, VA, USA**:** 10.

Collard, M. L., H. H. Kagdi and J. I. Maletic (2003). An XML-Based Lightweight C++ Fact Extractor. Proceedings of the 11th IEEE International Workshop on Program Comprehension, IEEE Computer Society**:** 134.

Collard, M. L., Maletic, J. I., and Marcus, A (2002). "Supporting Document and Data Views of Source Code." in Proceedings of ACM Symposium on Document Engineering: 8.

Dean, J., D. Grove and C. Chambers (1995). Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Proceedings of the 9th European Conference on Object-Oriented Programming, Springer-Verlag**:** 77-101.

Dig, D., M. Tarce, C. Radoi, M. Minea and R. Johnson (2009). Relooper: refactoring for loop parallelism in Java. Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. Orlando, Florida, USA, ACM**:** 793-794.

Dijkstra, E. (1979). Go to statement considered harmful. Classics in software engineering, Yourdon Press**:** 27-33.

Dragan, N., M. L. Collard and J. I. Maletic (2009). Using method stereotype distribution as a signature descriptor for software systems. in the Proceedings of the IEEE International Conference on Software Maintenance (ICSM'09).

Dragan, N., M. L. Collard and J. I. Maletic (2010). Automatic identification of class stereotypes. Proceedings of the 2010 IEEE International Conference on Software Maintenance, IEEE Computer Society**:** 1-10.

Emami, M., R. Ghiya and L. J. Hendren (1994). "Context-sensitive interprocedural points-to analysis in the presence of function pointers." SIGPLAN Not. **29**(6): 242-256.

Feng, L. (2009). "Automatic parallelization in Graphite." from http://gcc.gnu.org/wiki/Graphite/Parallelization.

Ghezzi, C. and M. Jazayeri (1982). Programming language concepts, Wiley.

Goff, G., K. Kennedy and C.-W. Tseng (1991). "Practical dependence testing." SIGPLAN Not. **26**(6): 15-29.

Group, T. P. (2012). "Parallel Fortran, C and C++ Compilers ", from http://www.pgroup.com/products/pgicdk.htm.

Grove, D., G. DeFouw, J. Dean and C. Chambers (1997). Call graph construction in object-oriented languages.

Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Atlanta, Georgia, USA, ACM**:** 108-124.

Intel. (2010). "Automatic Parallelization with Intel Compilers." from http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/.

Intel. (2010). "Automatic Parallelization with Intel® Compilers." from http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/.

J.L. Hennessy , D. A. P. (2006). "Computer Architecture: A Quantitative Approach." Morgan Kaufman Publishers, San Francisco.

Jacobson, T., Stubbendieck, Gregg (2003). "DEPENDENCY ANALYSIS OF FOR-LOOP STRUCTURES FOR AUTOMATIC PARALLELIZATION OF C CODE."

Kennedy, K., Allen, John R. (2002). "Optimizing compilers for modern architectures: a dependence-based approach." Morgan Kaufmann Publishers Inc.

Kim, M., Kim, H. , Luk, C.-K (2010). "Prospector: A dynamic data-dependence profiler to help parallel programming." In 2nd USENIX Workshop on Hot Topics in Parallelism , HotPar '10.

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

Kulkarni, D., Stumm, Michael, A, Ms , Kulkarni, D (1993). "Loop and Data Transformations: A Tutorial." University of Toronto.

Maydan, D. E., J. L. Hennessy and M. S. Lam (1991). Efficient and exact data dependence analysis. Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. Toronto, Ontario, Canada, ACM**:** 1-14.

Mock, M., D. C. Atkinson, C. Chambers and S. J. Eggers (2005). "Program Slicing with Dynamic Points-To Sets." IEEE Trans. Softw. Eng. **31**(8): 657-678.

Muth, R. and S. K. Debray (1997). On the Complexity of Function Pointer May-Alias Analysis. Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development, Springer-Verlag**:** 381-392.

Nadgir, N. (2001). "Using OpenMP to Parallelize a Program." from http://developers.sun.com/solaris/articles/openmp.html.

Nikolopoulos, D. S., C. D. Polychronopoulos, E. Ayguad´e, J. u. Labarta and T. S. Papatheodorou (2001). The Trade-off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms. Proceedings of the 15th international conference on Supercomputing (ICS '01). Sorrento, Italy, ACM**:** 15.

Novillo, D. (2006). OpenMP and automatic parallelization in GCC. Proceedings of the GCC Developers' Summit conference. Ottawa, Canada.

Oracle. (2010). "Subprogram Call in a Loop." from http://docs.oracle.com/cd/E19205-01/819-5262/aeuje/index.html.

Orso, A., S. Sinha and M. J. Harrold (2004). "Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging." ACM Trans. Softw. Eng. Methodol. **13**(2): 199-239.

Petersen, P. M. and D. A. Padua (1993). Static and dynamic evaluation of data dependence analysis. Proceedings of the 7th international conference on Supercomputing. Tokyo, Japan, ACM**:** 107-116.

Petersen, P. M. and D. A. Padua (1996). "Static and Dynamic Evaluation of Data Dependence Analysis Techniques." IEEE Trans. Parallel Distrib. Syst. **7**(11): 1121-1132.

Petersen, P. M., Padua, David A. (1991). "Experimental Evaluation of Some Data Dependence Tests " Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, Illinois, 61801.

Psarris, K. and K. Kyriakopoulos (1999). Data Dependence Testing in Practice. Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society**:** 264.

Pugh, W. (1991). The Omega test: a fast and practical integer programming algorithm for dependence analysis. Proceedings of the 1991 ACM/IEEE conference on Supercomputing. Albuquerque, New Mexico, United States, ACM**:** 4-13.

Robert P. Wilson , R. S. F., 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 (1994).

"SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers." ACM SIGPLAN Notices **29**: 31--37.

Shah Anand, R. B. G. (1995). Function Pointers in C - An Empirical Study. Technical Report LCSR-TR-244**:** 11.

Spuler, D. A. and A. S. M. Sajeev (1994). "Compiler Detection of Function Call Side Effects." Technical Report 94/01.

Sundaresan, V., L. Hendren, C. Razafimahefa, R. Vall, #233, e-Rai, P. Lam, E. Gagnon and C. Godin (2000). Practical virtual method call resolution for Java. Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Minneapolis, Minnesota, USA, ACM**:** 264-280.

Zhang, S. and B. G. Ryder (1994). Complexity of Single Level Function Pointer Aliasing Analysis, Rutgers University, Department of Computer Science, Laboratory for Computer Science Research.