Parallel Path-Based Static Analysis

Mitali Parthasarathy
Department of Computer Science
University of Virginia
Charlottesville, VA, 22904
mp2at@virginia.edu

Wei Le
Department of Computer Science
University of Virginia
Charlottesville, VA, 22904
weile@virginia.edu

Mary Lou Soffa
Department of Computer Science
University of Virginia
Charlottesville, VA, 22904
soffa@cs.virginia.edu

ABSTRACT

Multicore architectures are widely used in research and industry today. The innovations in parallel computer architectures create an opportunity for the development of software solutions that exploit the available parallelism. Static program analysis is a well-explored area of research which has many applications including detecting software vulnerabilities, test case generation, debugging and program parallelization. For many types of static analysis, scalability is an important concern. In prior research, the performance of static analysis has been improved with techniques such as exploring a limited portion of the search space, flow-insensitivity, context-insensitivity, abstraction and the use of heuristics. Many of these techniques can be viewed as trade-offs that sacrifice precision in favor of scalability and practical applicability.

Previous work has demonstrated that demand-driven query propagation is an effective and efficient static analysis technique that is more scalable than current techniques. In this research, we present a version of the demand-driven technique that is multithreaded to further increase scalability. The key idea that underlying our approach is that query-based demand-driven analysis exhibits significant opportunities for parallelization because queries typically have few dependencies. We present a parallel algorithm for statically detecting buffer overflows and evaluate its feasibility in a path-based static analysis framework called Marple. We implemented the scheme using Microsoft Phoenix [20] and Marple [15], a tool produced at University of Virginia.

Keywords

Static analysis, buffer overflow, demand-driven propagation, multi-core architecture

1. INTRODUCTION

Statically analyzing program source code to check for certain program properties is widely used in software development and maintenance activities such as detection of software vulnerabilities, test input generation, program parallelization and debugging [14, 25]. One major advantage of static tools compared to dynamic tools is its guarantee of finding all bugs in a program without executing it, while dynamic analysis is often incomplete. That is, through static analysis, we are able to obtain the behavior of relevant executions of a program without running it.

Although static techniques are useful, it is challenging to make a static tool effective in practice. A major hindrance to static analysis is the lack of scalability. There is a lot of research focused on increasing the precision of static analysis of programs by including path-sensitivity [6]. However, exhaustive path-based techniques have not been shown to be scalable to a million lines of code. Nowadays software is becoming larger and more complex thereby imposing the requirement for any analysis to efficiently handle a large state space. In the current state of the art, expensive static analysis is either terminated after exploring only a limited portion of the state space or sped up with solutions (such as intraprocedural, flow-insensitivity or context-insensitivity, abstraction or heuristics) that sacrifice precision for performance [11].

Reducing the state space of a program on which static analyses are performed is the key to scalability. One technique used to minimize the exploration and storage of state space associated with a program is the elimination of infeasible paths by *demand-driven propagation* [4]. Demand-driven program analysis initiates processing at statements of interest (depending on the information required to be extracted) and propagates a query backward along the interprocedural control flow graph (ICFG) of the program until it can be resolved [7].See Section 2.2 more detailed explanation of demand-driven analysis. This technique eliminates the propagation of information along infeasible and unnecessary paths thereby reducing the execution time of the analysis. Even so, demand-driven program analysis which is flow and context sensitive does not scale sufficiently to real-world programs, which often exceed hundreds of thousands of lines of code.

1.1 Motivation

In previous efforts to build scalable program analyses to detect software vulnerabilities in programs, a scalable path-based framework called Marple was developed [15]. Marple takes as input an application program and a specification

of program properties and automatically generates an analyzer to detect violations of these program properties in the application. The tool has been effective in detecting several classes of bugs such as buffer overflow, integer overflow and null pointer deferences. Unlike other tools that perform similar work, this framework presents the user with information pertaining to the program paths along which the potentially vulnerable statements are detected as well as the path segments which cause the violation and the statements which are directly responsible for it. This results in reducing developer time required to fix the bug by providing the context (i.e. trace) in which the vulnerability manifests itself. In addition, the framework is also able to detect interactions between different bugs and their impact on the correct functioning of the application. See Section 2.1 for more details on the implementation of Marple. It has been evaluated on several benchmarks with up to 1.6 million lines of code and has been shown to scale well with the size of the application. In this work, we are further increasing the scalability of path-based program analysis by exploiting architectural innovations.

Advances in computer architectures like multi-core CPUs have created vast interest in introducing parallelism to improve the performance of several applications such as ray tracing, abstractions for program parallelization, compiler for Brook streaming language and communication protocol design (eg. [22], [26], [18], [19]) to name a few. Program analysis is one such application that can gain significant improvement in performance by exploiting parallelism. Demand-driven program analysis is widely used due to the several benefits it offers over exhaustive analysis. In this technique, propagation of information (in the form of *queries*) is initiated at potentially vulnerable program points and propagated backwards along the ICFG of a program until there is sufficient information to classify it as a vulnerability. Even though the queries can communicate with each other by taking advantage of the relevant cached information at each node, it is not strictly necessary. As each query is independent of the others, enabling the propagation of queries that are exploring one/more program properties to execute on different cores simultaneously increases scalability. While the queries share the same call graph associated with a program, the successive passes which raise and resolve the queries are intrinsically independent providing opportunity for substantial parallelism.

According to Amdahl's law, the speedup of a parallel program is limited by the time needed for the sequential fraction of the program [2]. We have found the time taken to raise queries in a program to be small enough (less than 1s for all the benchmarks tested) to not limit the speedup in most cases. Thus the maximum speed-up of parallel demand-based query propagation is limited by the time spent in solving the most complex query in the analysis. In order to gain insight in to the amount of parallelism that can be obtained by this approach, we performed some preliminary experiments. We evaluated several benchmarks on the Marple framework and measured the time taken to solve each query. The highest execution time that is required to solve a query is an approximate estimation of the execution time of the benchmark. A previous version of Marple was used in these preliminary experiments and has since been updated. We use the updated version in our evaluation in Section 4. Table 1 contains 6 synthetic benchmarks and 8 real-wrold benchmarks. A hypothetical speedup of upto 77% for the synthetic programs and 86% for the real-world programs is seen. However, the estimate assumes that there are enough cores to start all threads at the same time (which is not unreasonable for the small benchmarks) and does not take into account the overhead of thread creation and deletion.

Benchmark	Size (kloc)	Total Time(sec)	Longest query time(sec)	Speedup (%)
StringOp	0.033	0.68	0.16	77
Interfoo	0.075	1.09	0.4	63
Insertion Sort	0.055	7.46	7.13	4
Substring	0.037	0.71	0.24	66
String Reversal	0.031	1.6	1.2	25
Palindrome	0.029	0.47	0.09	80
Poly-0.4.0	1.7	3.9	2.9	26.1
Sendmail-2	0.7	25.3	17	32.5
Wu-ftpd	0.4	52.7	21.8	58.6
Sendmail-6	0.4	205.8	202.9	1.43
Ffmpeg	39.8	31.8	18.7	41.1
Gzip	8.2	946.3	818.5	13.5
Apache-2.2.4	418.8	6823.3	931.4	86.3
Putty-0.56	112.4	4632.1	1599	65.48

Table 1: Hypothetical Speedup for 8 C/C++ benchmarks in the Marple framework

1.2 Our Solution

In this work, we develop a parallel algorithm to generate fast and precise program analysis to detect buffer overflow errors in C/C++ programs. Demand-driven analysis is naturally parallel in that resolving each query pertaining to a potentially vulnerable statement in a program is mostly independent. Thus, a straightforward way to parallelize the analysis is to execute each query on a separate thread and simultaneously on multiple CPU cores. The challenge here is to make optimal use of cores at all times. The threads need to be ordered in a way that prevents many cores from being idle at any point of time while paying attention to thread dependencies. We have adapted our algorithm to conduct a case study in evaluating the potential parallelism of the Marple framework. In reality, this algorithm should be no different from the general algorithm. However, due to reasons explicated in Section 1.3, there was a need to sacrifice some parallelism to support multhithreading. The two main contributions of this paper are:

1. A general parallel algorithm to resolve queries corresponding to desired program properties

2. An implementation and evaluation of our algorithm in the context of the Marple framework [15]

Experimental results in Section 4 evaluate the feasibility of this approach for a few sample programs as well as some real-world programs.

1.3 Implementation Challenges and Properties

Even though our parallel algorithm can be applied to any framework, we chose to evaluate it on the existing Marple

framework. In this section, we present various factors that contribute to the ease/difficulty of parallelizing Marple efficiently. A major challenge to the implementation of the algorithm was the properties of the underlying framework. Marple is built using the Microsoft Phoenix compiler framework which is an analysis framework for building compilers, tools for testing, program analysis and software optimizations [20]. While Phoenix has proved to be very useful in building tools for program analysis, the latest release does not contain adequate support for our work. More specifically, Phoenix does not support multithreading within a *function unit* (*function unit* refers to a function in the application program that is being analyzed). In other words, there is no synchronization within a function unit to protect the data structures that is maintained inside a function unit and shared by all threads accessing the function unit. This property combined with the limitation of selected API results in a race condition when the shared data is enumerated.

In our experience, introducing synchronization using primitives like locks or mutexes contribute to a huge overhead (around 2x slowdown) making multithreading ineffective. Further, implementing these data structures is a huge effort which could not be accomplished due to time constraints. Rather than parallelizing the queries, it is easier to parallelize across function units by suitably managing the dependence structure of the program without incurring the cost of employing locks. As a result, we are forced to raise the parallelism to the function unit level. In this paper, we first present the general algorithm for parallel static analysis followed by a functionally parallel algorithm for the Marple case study.

In spite of the complexity involved in parallelizing at the function unit level, certain properties of Phoenix make it a viable choice for us. Firstly, it provides an excellent framework for quickly generating static analyses of application programs. A plug-in model for the backend of Phoenix allows for easy addition of new phases/passes to perform any required analysis. Secondly, Phoenix utilizes the Microsoft C++ Compiler (C2.exe) to perform analysis and code generation for function units in parallel, which reduces the execution time considerably. However, the reduction in execution time does not make any difference to the speedup obtained as the sequential version also uses C2. Other advantages of the Phoenix framework include language independence, many levels of intermediate representation and compatibility with many architectures.

2. APPROACH

We have implemented our parallel static analysis algorithm in a path-based framework called Marple for identifying and reporting buffer overflows in an application program [15]. The Marple framework was created in previous efforts by the authors of [15]. In this paper, it is being extended to evaluate parallel static analysis. First, we present the framework in which the parallel static analyses are implemented. Demand-driven query propagation forms the crux of our static analysis and is presented next. Finally, an overview of parallel query propagation in a multithreaded environment completes a high-level description of our implementation.

2.1 Marple Framework

The Marple framework is used to statically generate analyses, based on specific program properties. The input to the framework consists of program properties, which we will call *specifications*, and an application program which is being analyzed for certain properties. The program paths along with the specific path segments which exhibit the required properties are output as a result of the analysis.

2.1.1 Specification

The path properties are defined by the user based on the specification language that is a part of the framework. It consists of points in a program where the property can be observed (known as α-*points* and α-*constraints*) and how it is effected along the different program paths (known as α-*impacts* and α-*transitions*) [15]. An example of the specification for the buffer safety property in a program taken from [15] is shown in Figure 1. *Code signatures* are used to describe the α-points and α-impacts while *attributes* describe the α-constraints and α-transitions. The code signature is a language based construct, which is in C for our purpose. The '$' symbol is used to specify code signatures. This can be altered to represent constructs in other languages as well. Attributes specify a characteristic of the program state. For example, in Figure 1 Size(a) refers to the size of a buffer or Len(b) refers to the length of the string stored in it. Using attributes, even advanced language constructs can be depicted effectively. There are three main sections in the specification of a property. **Vars** is used to define the variables that are used in the specification. The **ObservablePoints** section contains the α-points which are the statements in the program where the property may be observed or confirmed. These are paired with α-constraints that indicate the constraints that must be satisfied for the property to hold along the paths in which the statements are present. **DefiningPoints** is used to represent the α-impacts which are statements in a program which can lead to a resolution of the constraints imposed on it. Along with them, we also define α-transitions which are changes in program state caused by α-impacts.

2.1.2 Approach for Generating Path-based Analysis

The first step in generating demand-driven analysis is parsing the specification to generate code for the query constructor, evaluator and transformer [15]. The generator consists of three modules - language, attribute and path which make up the interface between the the specification and the framework and enables demand-driven propagation. Parsing a specification results in preprocessed code signatures and the required syntax trees. The preprocessing step produces an intermediate representation consisting of conditions and commands by replacing code signatures with constraints corresponding to the operator and operand in the statement [15]. The conditions and commands are then used to build pairs of syntax trees corresponding to α-points and α-constraints, and, α-impacts and α-transitions. Finally, attribute models are used to generate code based on the structure of the syntax tree.

The output from the first step is fed to the query propagator which performs the required analysis. The query propagator uses the code modules which are produced by the generator and guides the analysis in a demand-driven fashion. Every node in the ICFG is examined and a query is raised if α-point and α-constraints are determined. Path

Vars

Vbuffer a, b; Vint d; Vany e

ObservablePoints

α-Point $strcpy(a,b)$
α-Constraint Size(a). Len(b)
or
α-Point $memcpy(a,b,d)$
α-Constraint Size(a). min(Len(b), Value(d))
or
α-Point $a[d]=e$
α-Constraint Size(a). Value(d)

DefiningPoints

α-Impact $strcpy(a,b)$
α-Transition Len(a) := Len(b)
or
α-Impact $strcat(a,b)$
α-Transition Len(a) := Len(a)+Len(b)
or
α-Impact $a[d]=e$ && Value(e)='\0'
α-Transition Len(a) := Value(d)

Figure 1: Specification for the Buffer Safety property

modules define the backward propagation of queries starting at α-points, until it can be resolved as well as the forward propagation for isolating paths [15]. When the propagation reaches a fork, the query information is transmitted along both paths. Similarly, the propagation merges when they reach a join in the ICFG. Propagation to a different function is necessary when there is a function call. A linear scan is performed to determine the caller function and the query is propagated only to that function. Loops are a special case and need to be handled exclusively. Encountering a loop causes Marple to evaluate the impact of the loop. If the loop has no impact, propagation continues as usual. Otherwise, the loop is executed twice and an attempt to symbolically reason about the loop count and updated information from each iteration of the loop is made. If no such reasoning can be applied, the imprecision is recorded and propagation is continued.

2.2 Demand-driven Queries

We now briefly describe demand-driven analysis; for a full description, we refer the reader to previous work [15]. Demand-driven analysis is centered around the concept of a *query*. A query q is defined as a pair

$$q = \langle y, n \rangle$$

where y is a program fact of interest (e.g., a predicate in a formal logic), and n is the program point of interest (identified by a node in the ICFG). Demand-driven analysis maps a demand into a query and identifies desired information by computing the resolutions for the query. For example, to report whether the program contains a buffer overflow, we construct a query at the statement where a buffer overflow can occur. On a high-level, the demand-driven analysis is a worklist algorithm. Each query along with a program statement is stored in the worklist. If the query and the statement pair are an α-impact, they are evaluated. Queries can be updated at each node while it is being propagated. Query propagation rules are determined from previous work like [4, 7]. Intermediate results collected while resolving each query can be shared by caching them at each step.

2.3 Parallel Queries

The generation and resolution of queries along the ICFG of a program are implemented in separate passes. Each pass consists of a series of phases which perform different analysis tasks. The first pass in the analysis is the construction of an ICFG and raising of queries for the application program that is being analyzed. Once the ICFG is generated by the Phoenix framework, we visit every call node pertaining to functions in the program. When a potentially vulnerable statement is encountered (α-point), a query is raised and added to the worklist. The propagation and resolution of queries is carried out in the next pass. The call graph generated in the first pass is stored and read by the next pass. This portion of the analysis is the target of parallelization. In the original Marple framework, a query is chosen from the worklist and processed. The next query is chosen only after the previous query has completed. In our multithreaded version, each query is processed on an independent thread in parallel to the other threads. All threads are created around the same time but start execution depending on the availability of cores. Subsequently, each thread propagates the query backwards checking for query resolution at each α-impact.

While the above approach should be applicable in most frameworks or environments, the Marple framework requires working around the non-existence of synchronization within a function unit. As noted in Section 1.3, the Phoenix compiler framework used in Marple necessitates the use of function-level parallelism. In order to evaluate the application of parallelism, we designed another algorithm in which care is taken to ensure that not more than one thread is operating on the same function unit at the same time. Queries are added to a *query worklist* which is sorted based in the function unit. A thread is created and started for the top element in each function unit's worklist. The next thread is started when the previous thread in that function unit completes execution. Even though this workaround avoids a race condition, the amount of parallelism now depends on the number of function units which, in most cases, is considerably less than the number of queries as suggested by the general algorithm.

We have optimized the number of queries that are raised by eliminating the exploration of functions that are never called in the program. The basic idea for this optimization is similar to demand-driven analysis in that infeasible paths lead to false positives and hence we identify and exclude them. Therefore, we search the ICFG for the *main* function and only analyze functions which are reachable from it and ignore the other nodes in the call graph. We found a 5x reduction in the number of queries in the optimized version compared to the original version.

3. PARALLEL ALGORITHM

This section contains the general algorithm for parallel demand-driven query propagation as well as the modified algorithm for the case study used in this paper. The algo-

rithm for generating the path-based analyses is taken from [15] and hence is not discussed here. The first algorithm employs query-level parallelism and is potentially more beneficial (although we do not evaluate it in this paper). The second algorithm is based on function-level parallelism and was constructed to overcome the constraints in the framework which is being used for evaluation in this work.

```
   Input: Program p
   Output: paths of property X
 1 icfg = BuildICFG(p);
 2 set worklist L to {}; set queryList M to {} ;
 3 foreach s ∈ icfg do
 4 |   if isNode(s) then
 5 |   |   q = RaiseQuery(s);
 6 |   |   add (q, s) to M;
 7 |   end
 8 end
 9 foreach (q, s) in M do
10 |   Create new thread t; add (q,s) to t.L;
11 |   Start t to execute SolveQuery(q,s);
12 end
13 Join all threads;
14 SolveQuery(query q, statement s)
   while L ≠ 0 do
15 |   remove (q,s) from L;
16 |   if isNode(s) then
17 |   |   transQ(s,q);
18 |   end
19 |   if q.resolved then
20 |   |   add (q, s) to A[q];
21 |   end
22 |   else
23 |   |   foreach n in Pred(s) do
24 |   |   |   PropagateQuery(s,n,q);
25 |   |   end
26 |   end
27 end
28 IdentifyP(A[q]);
```

Algorithm 1: General demand-driven parallel algorithm

3.1 General Algorithm

Algorithm 1 can theoretically be generalized to any demand-driven path based static analysis framework which contains adequate underlying synchronization support. Even in the absence of synchronization, it is relatively easy and inexpensive to lock the required data structure if it is implemented by the programmer in contrast to the usage of existing data structures in the tools used. This algorithm is an extension of previous sequential demand-driven static analysis implemented in Marple to the multithreaded environment [15].

The algorithm takes a program p as input and outputs paths relevant to one/more program properties X as listed in the specification. The path-based analysis generated in the previous step is stored in A. In line 1, the ICFG of the program is constructed. **isNode** is a call to the code generated in the previous step of the analysis. Every node is visited and a query is raised by the **RaiseQuery** function if an α-point and α-constraint are determined in lines 2-8. Every query and statement pair is added to the *querylist* to be processed by the next step. In lines 9-13, threads are created for each element in the *querylist*, the query is added to each thread's worklist and solved simultaneously by executing the **SolveQuery** function in lines 14-27. In **SolveQuery**, each thread's worklist item is extracted one at a time and examined. If it is an α-impact, the corresponding *transition* is performed in line 17 by the **transQ** function. Subsequently, if the query is resolved, it is added to A in line 20. Otherwise, it is propagated to its predecessor in line 24 by the **PropagateQuery** function. Finally, the paths corresponding to the property are identified by a call to **IdentifyP** in line 28.

```
   Input: Program p
   Output: paths of property X
 1 set querylist M, worklist L, threadlist W, functionlist F
   to {};
 2 icfg = BuildICFG(p); add function unit of node to F;
 3 foreach s ∈ icfg do
 4 |   if isNode(s) then
 5 |   |   q = RaiseQuery(s);
 6 |   |   add (q, s) to M;
 7 |   end
 8 end
 9 solvers = InitiateSolvers(M,F);
10 foreach n ∈ solvers do
11 |   Start n with SolveQuery(q,s);
12 end
13 Join all threads;
14 InitiateSolvers (querylist M, functionlist F)
   foreach f ∈ F do
15 |   Create a thread for f and add to W;
16 end
17 foreach (q, s) ∈ M do
18 |   SetWorklist(W,bq);
19 end
20 return W;
21 SetWorklist (threadlist W, basicblock bq)
   foreach t ∈ W do
22 |   if t.functionUnitName = bq.functionUnitName
   |   then
23 |   |   add bq to t.L
24 |   end
25 end
```

Algorithm 2: Marple demand-driven parallel algorithm

3.2 Marple Case Study Algorithm

The Marple case study which is evaluated in this paper requires a few modifications on Algorithm 1 to support multithreading. A variation in constructing the worklist of each thread is implemented in this algorithm. **SolveQuery** remains the same as in Algorithm 1. Queries are raised from the ICFG similar to the general algorithm. At each node the function unit (function in the application program) at which the query is raised is recorded in a list. The queries are added to the *querylist* in line 9 from which the individual thread worklists are constructed. A call to **InitiateSolvers** in line 9 initializes and returns the set of threads to process queries. Threads are created and the worklist of each thread is determined in lines 14-20.

The key difference between this algorithm and the previous one is the way in which the worklist is determined

in the **SetWorklist** function. Every query in the *querylist* calls the **SetWorklist** function with the *threadlist* (made up of the thread for each function unit and the basic block of the query in line 18. Lines 22-26 sort the worklist according to the name of the function unit. Sorting is done to ensure that threads do not operate on the same function unit at the same time. A check to see if the thread's function unit name is the same as the basic block's function unit name is performed in line 23. Lines 10-13 start all the threads and subsequently join them. The result of using the technique of sorting threads by function unit results in decreasing the level of parallelism in most cases as the number of function units is less than the number of queries.

4. CASE STUDY EVALUATION

4.1 Setup

We evaluated the second algorithm in the Marple framework which consists of Microsoft's Phoenix compiler infrastructure and the Disolver constraint solver [20, 9]. The features provided by Phoenix include points-to analysis, ICFG construction, and part of the attribute module. Evaluation of inequalities for a range is conducted by the Disolver. The experiments were conducted on an Intel quad-core 2.3GHz machine with 16GB RAM running Microsoft Windows Server 2003. The specification for the framework tested for buffer overflows only. The first part of the evaluation tested synthetic benchmarks created by the authors. The second part discusses the real-world benchmarks that were tested and the extensions that were implemented for it. Each benchmark was run three times and the average of the three runs is presented here.

4.2 Synthetic programs

4.2.1 Benchmarks

We first evaluated the Algorithm 2 on a set of synthetic programs that we created and manually injected with simple faults. Most of the benchmarks such as *Insertion sort, Palindrome, Substring recognition* and *String reversal* are well-known problems with commonly accepted textbook solutions that manipulate arrays and strings. *StringOp* and *Interproc* are test programs with interprocedural string manipulations. *Polymorph-0.4.0* is actually a real-world benchmark which is presented again in Section 4.3 (after implementing extensions) for comparison. It is a Win32 to Unix file name converter. The program searches the current directory for 'mangled' names created while using Windows applications. Consequently, the mangled name is converted to a simpler file name that follows Unix naming conventions. The synthetic programs are 35 lines long on average while *Polymorph-0.4.0* is 1800 lines. Note that the *Polymorph-0.4.0* benchmark used here has been modified (function calls in loops are inlined) according to the description in Section 4.3.1 to ensure a fair comparison of this result with the extended algorithm described in that section.

4.2.2 Results

Table 2 shows the results for executing the sequential and parallel version of the analysis on 7 C/C++ programs and the speedup which is obtained. The results for the small programs were as expected. On average, we found that multithreading produces up to 63% speedup for these programs.

Benchmark	Size (loc)	Query Count	Bugs Found	Speedup (%)
StringOp	33	4	2	10
Interproc	75	6	4	14
Insertion Sort	55	2	0	13
Substring	37	5	1	63
String Reversal	31	5	3	7
Palindrome	29	4	2	2.4
Polymorph-0.4.0	1800	13	1	35

Table 2: Speedup for 7 synthetic C/C++ benchmarks in the Marple framework without extensions. Polymorh-0.4.0 is not a synthetic benchmark but is presented here for comparison with Table 3

More specifically, certain benchmarks (with more queries and function units) make optimal use of the available cores and perform better consequently. Other benchmarks are too small and raise most of their queries in the same function unit which does not allow for a high degree of parallelism. The *Substring* benchmarks shows the highest speedup of 63%. Even though all the queries are raised in the same function unit and worked on by a single thread, it avoids the multiple function calls to **SolveQuery** in comparison to the sequential version. These results are however not representative of performance gain in "real" benchmarks as they are small programs which are not complex and have manually chosen faults. In addition, there is lower contention for cores by the threads in the process compared to larger programs. Note that these numbers will not reflect the proposed speedup in Table 1 as it makes use of function-level parallelism as oppposed to query-level parallelism described there.

4.3 Real-World programs

Given that we saw a considerable speedup for small applications, we decided to try medium-sized, real-world applications. In order to do this, we had to implement a few extensions to our current algorithm for the following reasons:

1)For large programs, it is necessary to explicitly synchronize threads when there is a propagation request to the caller,

2)Similar synchronization is required if we want to resolve loops in parallel. Below, we describe the extensions that we implemented and the modifications that were required to evaluate real-world benchmarks.

4.3.1 Extensions

Interprocedural analysis is key to the precision of our analysis and it was very important to support it. As we have the restriction of ensuring that two threads do not enter the same function unit simultaneously, the act of propagating a query requires that we know in advance if there are any threads already operating on that function unit. Since the worklist is dynamically updated, there is no way to predetermine dependencies in the case of a propagation. Our solution to this problem creates and maintains a separate thread (R) which acts on a propagation list of all queries which are ready for propagation. Each propagation list element consists of a propagator function unit as well as target

function unit. All other threads periodically check to see if there is any query in the propagation list that needs to be processed. If a thread shares its function unit name with the function unit name of top element in the propagation list which is trying to propagate or is the target of propagation, it waits and allows the thread R to continue. R waits until both threads that are involved in the propagation enter the wait state. When R is ready, it performs the propagation and updates the worklist of the target thread with the query. Finally, R notifies the waiting threads to continue execution. This way, threads not involved in the propagation never have to wait. For this scheme to work all threads are in a busy waiting state. This is very expensive as we will see in Section 4.3.3 and we are looking into eliminating this cost.

With regard to loop resolution, we decided that the synchronization required for this was very heavy-weight. Therefore, loops are executed sequentially. This obviously reduces the parallelism and we have left parallel loop execution as future work. In addition, function calls within loops is a special case which requires further synchronization. For our evaluation, we inlined function calls within loops to overcome this. Inlining function calls for large benchmarks can be very difficult and we hope to avoid this with better synchronization techniques in the future. The extensions described in this section were not atraightforward to implement and more multithreading experience is required to produce a more efficient algorithm.

4.3.2 Benchmarks

We chose 4 C/C++ Windows benchmarks for this portion of the evaluation. *Polymorph-0.4.0* has been presented earlier and is evaluated again to provide an idea of the slowdown incurred due to the extensions. *Sendmail-6* belongs to the buffer overflow benchmark that was created by Zitser et al [27]. It is a command-line utility to send SMTP mail through the SMTP server. The other two benchmarks are *Sendmail-2* and *Wu-ftpd*. *Sendmail-2* is an earlier version of the same utility with a different set of vulnerabilities. *Wu-ftpd* is a FTP server software used in Unix-like operating systems. The specifications of the benchmark and the results from running the sequential and parallel algorithm is presented in Table 3.

4.3.3 Results

The execution time for the sequential version is displayed before the '/' while the multithreaded version is after it. Evidently, there is a huge slowdown when more synchronizations are added. *Polymorph-0.4.0* shows roughly 4x slowdown when compared to previous evaluation which used Algorithm 2 without the extensions. *Sendmail-2* is also very slow compared to the sequential version. *Sendmail-6* has the same execution time as the sequential version as there is only one query. In summary, more work is required to reduce the synchronization overhead in analyzing larger benchmarks for the Marple framework. Alternately, an evaluation of the general parallel algorithm where no synchronization is required should be done using another framework to obtain an idea of the performance gain that is possible.

5. THREATS TO VALIDITY

In this work, there are a few limitations which prevent us from hypothesizing that parallel demand-driven query propagation is always beneficial:

Benchmark	Size (loc)	Query Count	Bugs Found	Exec time (s/m)
Polymorph-0.4.0	1800	13	1	7.9/13.5
Sendmail-2	700	16	0	19/47
Sendmail-6	400	1	1	49.5/50

Table 3: Results for 4 C/C++ benchmarks in the Marple framework with extensions

1. We have not evaluated our general parallel algorithm to explore its feasibility due to constraints of the Phoenix Infrastructure. We think that this will be a better indicator of the speedup that can be obtained form parallel static analysis.

2. The benchmarks we use are not representative of large-scale, real-world benchmarks. We have shown that smaller benchmarks may benefit from this approach but this may be an artifact of the complexity of the benchmarks or the type of faults chosen. More experimentation is necessary to evaluate the usefulness of our approach.

3. Medium-sized, real-world benchmarks required programmer effort to inline functions which is not acceptable for larger benchmarks. We think that this problem can be overcome by more careful thinking regarding the dependencies in framework. This is however a problem of the framework and not the approach.

There are also certain limitations that are due to the Marple framework:

1. The framework can only handle path properties that can be represented by the specification that is fed to it. In other words, it can be used to perform a subset of static analyses that is allowed by the expressiveness of the specification.

2. The framework implements a limited modeling of function pointers and exception control flows like signal handling. We depend on Phoenix to perform some pointer analysis, the precision of which is unknown.

6. RELATED WORK

The widespread use of multiprocessors has led to the need for the development of software that exploits the parallelism they provide. There has been much research in developing parallel algorithms such as weighted and unweighted graphing algorithms [21, 5] including minimum spanning trees [3, 1, 12] and genetic computing [24] to name a few. Parallel algorithms to perform data-flow analyses are addressed in [16, 17, 13]. Their techniques do not target the same problems as we do.

Demand-driven pointer analysis has been shown to scale to a million lines of code [11]. However, their analysis is achieved in a flow-insensitive, context-insensitive manner. In addition, it targets a very specific problem of resolving pointers. Parallel test input generation on multiple cores by distributing the search space to be tested among several threads while sharing meta-information to avoid redundant exploration is the focus of Siddiqui and Khurshid's work [10].

The key idea in this work is to use a constraint-based test generation strategy that systematically searches for valid test inputs. Improving the cost-effectiveness of model checking a program state space by a parallelized random state-space search to detect errors in programs is the idea in [8]. Using multiple processors to explore the state space allows the execution to terminate when the first error is found. However, these techniques do not perform demand-driven analysis. As a demand-driven approach is known to be scalable, adding parallelism further increases its scalability. Parallel generation of reachability graphs for JAVA programs has produced an 86% decrease in execution time in [23].

Currently, the proposed system focuses on parallelizing query propagation and resolution after generating the ICFG, but it can also be extended to parallelize the generation of control flow graphs and raising queries. Since generation of ICFG and raising queires has been found to contribute to only a small portion of the execution time of program analyis, we do not think that it is worthwhile to parallelize it for our purposes.

7. CONCLUSIONS AND FUTURE WORK

In this work, we argue that static analysis of programs can be made more scalable by taking advantage of the parallelism of multicore computer architectures. We present a general parallel algorithm and as well as a modified version to extend the Marple framework developed by [15]. Our evaluation on synthetic programs has shown a reasonable performance gain. Currently, real-world benchmarks do not experience a performance gain in the Marple framework due to the heavy synchronizations that are necessary to overcome the limitations of the Phoenix compiler framework.

There are several extensions to this work that can be performed. Firstly, it is necessary to evaluate the general parallel algorithm on a different framework with adequate support for multithreading. We expect to see a significant performance gain in this scenario. Secondly, it will be interesting to explore the effects of turning on the software cache to store queries at intermediate nodes as opposed to turning it off to eliminate the need for synchronization. Lastly, we expect the benchmarks to scale with the number of queries as well as the number of cores. An evaluation of scalability in this respect will present a more complete idea on the advantages and disadvantages of this approach.

8. REFERENCES

[1] M. Adler, W. Dittrich, B. Juurlink, M. Kuty, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms (extended abstract). In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 27–36, New York, NY, USA, 1998. ACM.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[3] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *J. Parallel Distrib. Comput.*, 65(9):994–1006, 2005.

[4] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *ACM Transactions on Programming Languages and Systems*, 1997.

[5] A. Chan, F. Dehne, P. Bose, and M. Latzel. Coarse grained parallel algorithms for graph matching. *Parallel Comput.*, 34(1):47–62, 2008.

[6] W. R. B. J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000.

[7] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. . *ACM Transactions on Programming Languages and Systems*, 1997.

[8] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. *International Conference on Software Engineering*, 2007.

[9] Y. Hamadi. Disolver: A distributed constraint solver. *Technical Report MSR-TR-2003-91, Microsoft Research*, 2009.

[10] J. Haroon, Siddiqui, and S. Khurshid. Pkorat: Parallel generation of structurally complex test inputs. *International Conference on Software Testing Verification and Validation*, 2009.

[11] N. Heintze and O. Tardieu. Object recognition with gradient-based learning. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2001.

[12] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 363–372, New York, NY, USA, 1992. ACM.

[13] R. Kramer, R. Gupta, and M. L. Soffa. The combining dag: A technique for parallel data flow analysis. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):805–813, 1994.

[14] W. Le and M. L. Soffa. Refining buffer overflow detection via demand-driven path-sensitive analyis. *PASTE*, 2007.

[15] W. Le and M. L. Soffa. A framework for scalable path-based static analysis. *Techinical Report*, 2009.

[16] Y.-F. Lee and B. G. Ryder. A comprehensive approach to parallel data flow analysis. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 236–247, New York, NY, USA, 1992. ACM.

[17] Y.-F. Lee, B. G. Ryder, and M. E. Fiuczynski. Region analysis: A parallel elimination method for data flow analysis. *IEEE Trans. Softw. Eng.*, 21(11):913–926, 1995.

[18] S.-w. Liao, Z. Du, G. Wu, and G.-Y. Lueh. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 196–207, Washington, DC, USA, 2006. IEEE Computer Society.

[19] S. Passas, K. Magoutis, and A. Bilas. Towards 100 gbit/s ethernet: multicore-based parallel communication protocol design. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 214–224, New York, NY, USA, 2009. ACM.

[20] Phoenix connect.

https://connect.microsoft.com/Phoenix.

[21] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319–348, 1984.

[22] K. Ramani, C. P. Gribble, and A. Davis. Streamray: a stream filtering architecture for coherent ray tracing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 325–336, New York, NY, USA, 2009. ACM.

[23] R. Rangarajan, S. Iyer, and G. Sajith. A technique for parallel reachability analysis of java programs. *Proceedings International Conference on Information Technology*, 2000.

[24] W. Rivera. Scalable parallel genetic algorithms. *Artif. Intell. Rev.*, 16(2):153–168, 2001.

[25] Y. Xie, A. Chou, and D. Engler. Archer: Using symbolic, path-sensitive analysis to detect memory access errors. *Proceedings of 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2003.

[26] L. Yi, C. Moretti, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2009. ACM.

[27] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM.