Scalable and Incremental Software Bug Detection

Scott McPeak
Coverity, Inc.
San Francisco, USA
smcpeak@coverity.com

Charles-Henri Gros
Coverity, Inc.
San Francisco, USA
chgros@coverity.com

Murali Krishna Ramanathan*
Indian Institute of Science
Bangalore, India
muralikrishna@csa.iisc.ernet.in

ABSTRACT

An important, but often neglected, goal of static analysis for detecting bugs is the ability to show defects to the programmer quickly. Unfortunately, existing static analysis tools scale very poorly, or are shallow and cannot find complex interprocedural defects. Previous attempts at reducing the analysis time by adding more resources (CPU, memory) or by splitting the analysis into multiple sub-analyses based on defect detection capabilities resulted in limited/negligible improvements.

We present a technique for parallel and incremental static analysis using top-down, bottom-up and global specification inference based around the concept of a *work unit*, a self-contained atom of analysis input, that deterministically maps to its output. A work unit contains both abstract and concrete syntax to analyze, a supporting fragment of the class hierarchy, summarized interprocedural behavior, and defect reporting information, factored to ensure a high level of reuse when analyzing successive versions incrementally. Work units are created and consumed by an *analysis master* process that coordinates the multiple analysis passes, the flow of information among them, and incrementalizes the computation. Meanwhile, multiple *analysis worker* processes use abstract interpretation to compute work unit results. Process management and interprocess communication is done by a general-purpose computation *distributor* layer.

We have implemented our approach and our experimental results show that using eight processor cores, we can perform complete analysis of code bases with millions of lines of code in a few hours, and even faster after incremental changes to that code. The analysis is thorough and accurate: it usually reports about one crash-causing defect per thousand lines of code, with a false positive rate of 10–20%.

*The author performed the work as an employee of Coverity Inc.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia

http://dx.doi.org/10.1145/2491411.2501854

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program Analysis*

General Terms

Reliability, Performance, Design

Keywords

Static Analysis, Bug Detection, Parallel, Incremental

1. INTRODUCTION

Static analysis for automated bug detection has been a well studied problem [16, 17, 15, 40, 19] and its usefulness well documented in [7]. A variety of bugs are detected by these techniques including null pointer dereferences, resource leaks, concurrency violations and buffer overflows. Detecting these bugs before deployment is important because it is quite expensive to fix a bug found in the field. There are also a number of dynamic analysis approaches for detecting bugs [41, 35, 20, 33, 21]. However, many of the dynamic analysis approaches are time consuming and are critically dependent on the quality of the tests generated. Motivated by the high cost of software bugs and the difficulties of thorough testing, we seek to build a static analysis that can accurately find bugs in large code bases and successfully integrate with real-world development processes.

Intra-procedural analysis techniques for bug detection are usually faster but cannot be used for detecting complex inter-procedural defects because of high false positive rate. On the other hand, accurately finding a compelling fraction of bugs requires interprocedural analysis: in particular, bottom-up [30, 2] propagation of behavior summaries, top-down computation [3] of calling context, and global specification inference [39, 26, 36]. Most industrial code bases are large, typically between one and ten million lines of non-blank, non-comment code (LOC). However, applying inter-procedural analysis on large codebases takes too long (sometimes running into weeks) [7].

Integrating static analysis with the development process requires at least two important properties. The first is it must be *fast*. The reason is that in order to be adopted into regular usage, static analysis must fit into one of the development cycles [4], and a natural fit is the nightly build cycle. A typical nightly build and test cycle takes 5–10 hours, so static analysis must complete within that time window; otherwise, organizations would have to create a new cycle just

for static analysis, and experience has shown that most are unlikely to do so.

Moreover, static analysis results quickly become stale. Static analysis identifies root causes, not bug symptoms, which is a big advantage—if results are delivered quickly. If not, developers are often reluctant to make changes to code that has "cooled" on the basis of defect reports sometimes perceived merely as theoretical [28]. New code usually goes through automated tests and/or a quality assurance cycle fairly soon after being checked in. Additionally, in many organizations, a significant component of the testing effort for a given piece of code happens implicitly as the product is exercised for other reasons. The longer a piece of code has been deployed, either internally or externally, the greater the sunk testing cost. That cost must be incurred again when the code is changed for any reason, even to address a static analysis defect. Delays also increase the cost of the developer learning or re-learning the relevant code. Therefore, timely delivery of defect reports is an important requirement for those reports to be useful.

The second important property is that defect results must be *deterministic* because this is critical to verifying a fix. Non-determinism means defects might disappear for reasons other than being fixed, even if the code is unmodified. Therefore, the results of analyzing a particular code base do not depend on how the analysis was run, for example, how many concurrent tasks were involved or how much incrementality was involved. We argue that determinism is necessary to build confidence in the tool: if users (developers) see results "flicker", they have a hard time learning what to expect from it, and may assume it is generally unreliable. Moreover, unpredictability makes it very difficult to put processes in place to *manage* the set of defects: both when dealing with the backlog when static analysis is first deployed, and handling the stream of new defects as development proceeds, it is critical that defects only disappear when they are truly fixed, and that they only appear when truly introduced. Otherwise it is impossible to answer the questions of how much work has been done and how much remains, which are central to the management task.

Additionally, the code being analyzed is mostly the same between any two consequent versions of a code base. So, it is natural to do *incremental analysis* by reusing results effectively. Small differences in the codebase should not trigger a full analysis and only relevant parts of the code need to be reanalyzed. More pointedly, for example, the addition of whitespace at the beginning of a file should not cause all of the functions and their interprocedural dependencies to be re-analyzed. Furthermore, the results of an incremental analysis must also be equivalent to the results from a full analysis – otherwise, user confidence quickly erodes.

We address the above discussed three objectives in this paper. We provide a design of a parallel analysis technique for interprocedural static analysis. We define a work unit as a serializable collection of self-contained analysis input data that can be analyzed independently (and quickly). The specific type of output from a work unit differs for each type of work unit, but is typically a set of defects or a function summary. The only inputs to the analysis of a work unit are the work unit itself and the analysis binary; i.e., the output is deterministic in those inputs, with the exception of possible diagnostic timing information included in the output. Furthermore, the output is platform-independent, in that analyzing the work unit on all platforms yields the same output (the analysis times can be different). The overall analysis is split between a single master process and a set of worker processes, with workers potentially on machines other than where the master is running. The master divides the analysis task up into work units that are sent to the workers, which then return results when each work unit is complete, for subsequent integration by the master. When the computation is finished, the master presents the finished results to the originating client and terminates the workers.

The design of a work unit has inherent challenges. As it is desirable for work units to be self-contained, there is design pressure to keep their size small, and avoid reliance on shared state, which adds complexity and compromises independence. Identifying the appropriate scope of a work unit so that the overhead of communicating and processing the work units (and results) back and forth between processes does not exceed the benefits accrued of distributing the work across multiple processes is important. Furthermore, the analysis of the work units must be deterministic under different resource conditions, or on different platforms.

Addressing these challenges yields a number of important benefits. The determinism of the work unit results enables the use of *incremental* static analysis. Our design of the work unit ensures that only a practically feasible minimum fraction of the entire code base is analyzed. Obviously, reducing the amount of code to be analyzed lessens the overall time taken for the analysis. The design of the work unit also yields a powerful advantage of failure isolation – when the analysis crashes or otherwise misbehaves, all that is required to reproduce the problem is the work unit and the appropriate analysis binary.

We have implemented our design in the Coverity Static Analysis [7] to analyze large codebases. The tool takes as input abstract syntax trees and performs a combination of bottom-up and top-down processing to detect various types of defects including null pointer dereferences, concurrency violations, resource leaks and buffer overflows. Applying our analysis on a number of large open source benchmarks (1-7 MLOC) shows a substantial reduction in the analysis time. Our results show that the analysis time is reduced by upto a factor of five to seven on an eight core machine on many C benchmarks, while still being able to detect critical defects with high precision (less than 20% false positives).

We make the following key technical contributions in this paper:

1. A design of a self contained analysis work unit that can be analyzed independently, quickly and deterministically.

2. A design and implementation for a scalable static analysis by leveraging the design of work units to find critical defects precisely.

3. An empirical evaluation of the proposed approach displaying its scalability on very large codebases with millions of lines of code.

The rest of the paper is organized as follows. We discuss the design challenges for building a scalable static analysis tool for bug detection and present the design and implementation of our approach in Section 2. In Section 3, we presents the results of our analysis on large codebases. The challenges posed in scaling the analysis for object-oriented

programs is discussed in Section 4. We compare our work with other approaches is Section 5 and conclude in Section 6.

2. DESIGN

2.1 Architecture

The overall architecture which includes the proposed design is given in Figure 1.

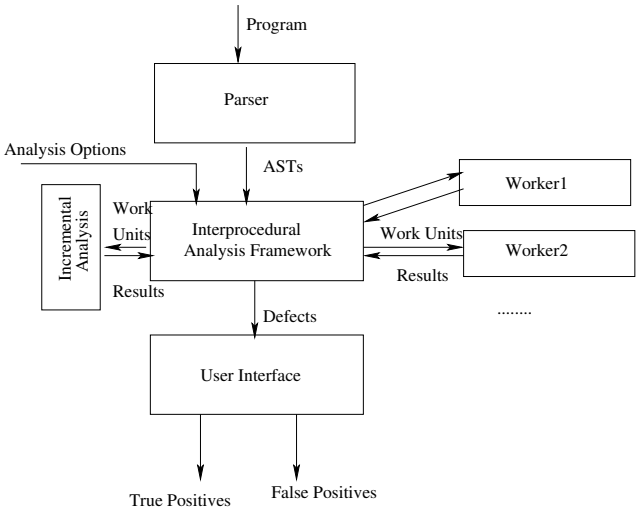

Figure 1: System Architecture.

First, we provide an overview of the interprocedural analysis computation to perform, independent of how it is spread across time and space. We informally use chronological metaphors, but that is just to convey data dependence.

To begin with, the program source code is parsed into an Abstract Syntax Tree (AST) forest, one tree per function. In this paper, we use the term "function" for functions, methods and initializer blocks.

Once the analysis phase begins, a "virtual linker" pass resolves named symbol references across translation units; heuristics are used to deal with dynamic linking. Then, indirect calls through function pointers and virtual function calls are approximately resolved using a whole program pointer analysis; this approximation is used to determine the interprocedural dependencies and hence available parallelism, and it is refined during the analysis proper by using the more precise information available at each individual call site. Finally, an acyclic call graph is constructed (treatment of recursion is omitted here).

With initialization complete, the analysis proper can begin. The core of the analysis is organized into "checkers" and "derivers". Checkers are responsible for reporting defects, while derivers are responsible for computing summaries for use interprocedurally. Each operates on a single function at a time, primarily using abstract interpretation [11].

The analysis as a whole is organized into five passes over the entire code base. First, in the statistical pass, derivers use intraprocedural analysis to summarize usage patterns for functions, which are then aggregated during the barrier to infer specifications [18] which augment the built-in rules. Because the statistical pass runs first and hence does not have access to interprocedural information, the inferred specifications are also (like indirect call resolution) somewhat imprecise initially, and must be combined with context-dependent information when used by later analysis passes.

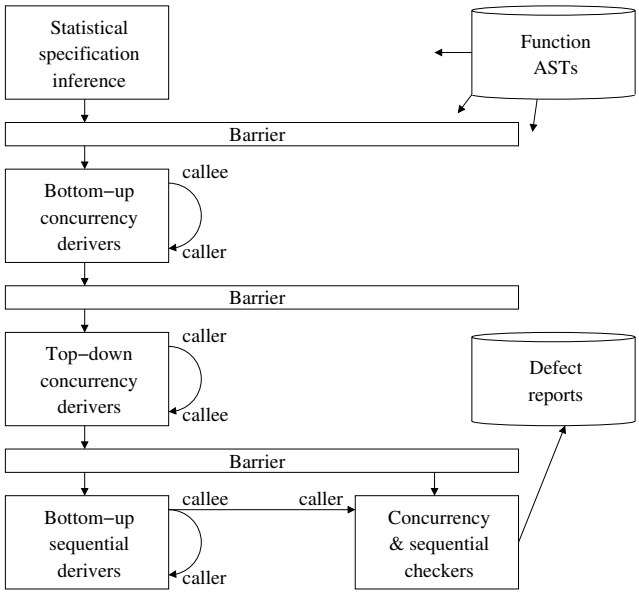

Figure 2: Interprocedural analysis computation dependencies.

Next, a bottom-up pass computes concurrency behavior summaries, including which mutexes are acquired, released or both. The derivers analyze each function, callees before caller, hence "bottom-up". Then, using results from the previous phase, a top-down pass summarizes additional concurrency behaviors, for example the detection of which mutexes are held on entry. Next we run another bottom-up pass, which summarizes sequential properties such as pointer dereferences and resource acquisition. Finally, the checkers run, and they report actual defects, both sequential and concurrent. After each deriver pass is a barrier: all functions in that pass must be analyzed before any function in the next is analyzed.

The current design places all the checkers to be run after the final barrier. However, sequential checkers need not be constrained by this barrier and could potentially be run in parallel after the statistical barrier. Relaxing this constraint will only expose more parallelism and reduce analysis time even further.

Figure 2 summarizes this structure. Each rectangle represents the set of computation nodes in a pass, one for every function in the call graph. Edges annotated with "caller" and "callee" on their endpoints connect nodes that have that relation in the call graph. Since there are five rectangles, if the program has m functions, then this computation dependency graph has $5m$ nodes, not counting the barriers.

As a brief illustration of the interaction in the sequential bottom-up phase, consider the program in Figure 3. When `g` is analyzed by the NULL pointer deriver, the result is a summary that says it unconditionally dereferences its first argument. When `f` is analyzed by the NULL pointer checker, it deduces that there is a path along which `p` is NULL when `g` is called. The checker consults the summary of `g`, sees that the pointer will be dereferenced, and reports a defect.

```
int f(int *p)
{
  if (p != NULL) {
    ...
  }
  return g(p);
}

int g(int *q)
{
  return *q;
}
```

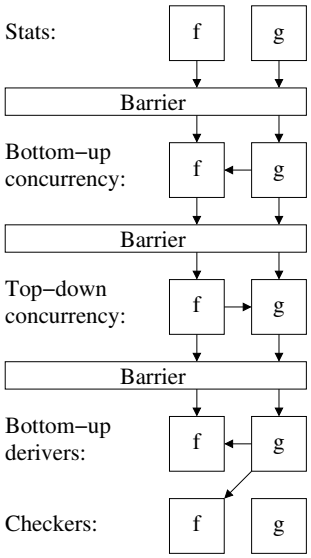

Figure 3: Example program with interprocedural defect (left), and its computation dependency graph (right).

```
int f(int *p)
{
  if (p != NULL) {

    ...
  }
  return g(p);
}
```

```
int g(int *q)
{
   h(*q);
   return *q;
}

void h(int q)
{
   print(q);
}
```

Figure 4: Incremental changes: Addition of method `h`.

The analysis is applicable for finding defects of various types and employs a number of derivers including NULL pointer deriver, resource deriver, lock deriver, etc. The details of what each checker and deriver do are beyond the scope of this paper. See [18, 16, 7] for discussion of some of them.

For any changes to the source, the interprocedural analysis need not run on the entire source. It suffices to analyze code that is affected by the modifications pertaining to the property. The design of the static analysis system into checkers and derivers for different properties is leveraged to reanalyze only the absolutely necessary parts of the program. We elaborate this further using an example.

In Figure 4, a method `h` is added and called from method `g`. A naive approach would be to reanalyze all the methods `f,g` and `h`. The design of our analysis ensures that not all methods need to be analyzed. Even though a potential null pointer dereference happens in the call to `h`, this does not change the overall function summary of `g`. In other words, the analysis of the changed program will only involve analysis of methods `g` and `h`. If there were other methods that are called from `f` (not shown in the Figure), then those methods need not be re-analyzed.

However, although `f` is not re-analyzed, the defect that will be reported to the user is different. That is because the defect contains interprocedural evidence; in particular, it cites the location and syntax of the dereference in `g`, which are different. This is made possible by deliberately omitting some defect reporting information from work units: instead, the work unit contains an abstract identifier for each piece of interprocedural evidence, and the work unit result refers to that identifier. When the analysis master receives the work unit result, it uses those identifiers to assemble the complete defect report for presentation to the user.

Most details of how the analysis of a single work unit operates are out of the scope of this paper, but the core is a general purpose abstract interpretation engine. For instance, loops are handled with widening and iteration to a fixpoint. Checkers vary in their choice of abstract domain, abstract operations and heuristic adjustments to the core engine behaivor. As static analysis techniques go, abstract interpretation is relatively expensive, but also can be very precise, which is important for user adoption. For the purpose of this paper, the key point is that analysis of a single function takes significant computational resources, and requires information from its dependencies according to the dependency graph.

2.2 Parallel Analysis

2.2.1 *Work Unit*

In typical programs, the structure in Figure 2 has a great deal of parallelism, and we exploit that directly.

Since at least 90% of a sequential analysis time is spent in abstract interpretation, which has a high ratio of computation to input and output, it forms a natural basis for task decomposition: we create one *work unit* for the abstract interpretation of each function. A work unit is an explicit realization of an atom of schedulable analysis work, with dependencies carefully crafted to both expose parallelism and minimize sensitivity to irrelevant changes. The result of analyzing a work unit is a combination of defect reports and behavior summaries.

A work unit contains the following elements:

- The AST of the function to analyze.
- Definitions of the types that the AST refers to.
- The source code of the function (but not the whole file).
- The callee behavior summaries and calling context summaries.
- Inferred specifications for interfaces used in the function.
- Analysis options.

This design of the work unit specifically ensures that each function is analyzed separately with only the information from the work unit. In practice this means the analysis only uses enough memory to load a single function, its models and some associated overhead. This relatively *flat* memory consumption enables us to use multiple cores in a predictable

and scalable fashion while using predictable amounts of memory. Knowing the approximate amount of memory ahead of time is critical. In contrast, whole program analysis can rarely estimate the required memory resulting in occasional failures.

2.2.2 Distributor

Work units are processed by *analysis workers*, which for software engineering and portability reasons are separate operating system processes. A single master process produces work units and consumes their results. Communication is done through a message-passing interface that can use various interprocess communication (IPC) primitives, such as sockets and explicitly shared memory. This design is based in part on Condor MW [22] and is depicted in Figure 5.

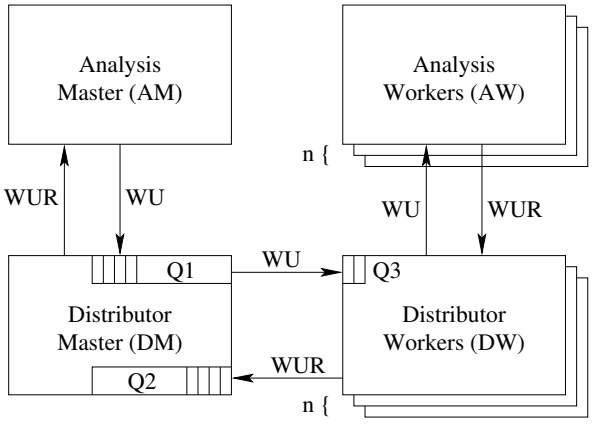

Figure 5: Process and communication architecture.

The user invokes the Analysis Master (AM). This process is responsible for reading from the program database, constructing work units (WU), consuming work unit results (WUR) and storing the results (defects and summaries) in the results database.

The AM spawns the Distributor Master (DM), a process primarily responsible for communication coordination and queueing. The DM in turn spawns n Distributor Worker (DW) processes, where n is chosen based on available hardware resources and influenced by user perferences. The DW is responsible for starting the Analysis Worker (AW), the process that computes the result of each work unit. The DW also monitors the AW, restarting the AW if it fails irrecoverably.

There are three types of communication queues: Q1 contains work units waiting for an available worker, Q2 contains work unit results ready for the AM to consume, and each DW contains an instance of Q3, a two-element queue intended to ensure that a worker does not become idle due to communication latency (which is mainly an issue when a worker is running on a different machine than the master). The AM tracks the number of outstanding work units and voluntarily pauses creating and enqueueing them once a few hundred are outstanding in order to conserve memory and give the workers a chance to catch up.

The DM and DW are collectively known as the Distributor Layer. This layer relieves the analysis processes of most fault tolerance concerns and simplifies their communication duties so that each analysis process only has to monitor one bidirectional IPC channel. The DW is a separate process so it survives if the AW crashes (in this context, a "crash" is usually an unrecoverable assertion failure). The DM is a separate process so that it has its own thread of control and therefore can service the IPC channels at the same time as the AM produces and consumes work units. The entire layer is independent of the global task to be performed; for example, internally, we use it to run our automated tests in parallel with each other.

The distributor layer takes care of measuring and aggregating various performance statistics such as worker idle time, memory usage and communication delays to enable later investigation into computation efficiently. The user can request more detailed logging, including logging every byte of every message. These diagnostic capabilities are invaluable when supporting a complex parallel computation running at remote sites to which one has limited access.

2.2.3 Scheduling

Generating an optimal schedule to parallelize the computations in a dependency graph is a variant of precedent-constraint scheduling and is NP-complete [27]. Therefore, we employ a greedy critical path scheduler. A critical path is a longest remaining path from a node to the next barrier. The critical path cost of a node is computed as the node's weight plus the maximum of the critical path costs of its successors. We then prioritize tasks based on the critical path cost.

In Section 2.1, we partitioned the analysis activities into derivers and checkers to maximize available parallelism. Whereas nothing in the analysis consumes the output of a checker, both derivers and checkers consume the output of derivers. That means that the derivers are on the *critical path*, but checkers are not. Therefore, to avoid resources from being idle, we must focus attention on that critical path. We delay running checker work units until we run out of parallelism in the deriver computation, since checker work units can be scheduled at any time once their callees have been analyzed by derivers.

Figure 6 illustrates the parallelism benefits of running derivers first. In schedule A, each work unit runs both checkers and derivers. Early on the analysis does reasonably well at utilizing workers because there is enough parallelism in the bottom half of the callgraph. But as we get nearer the top, callgraph parallelism drops off as the analysis becomes more serialized, and worker utilization drops. In schedule B, we have the exact same shape of available parallelism, but it only applies to the derivers. Since the derivers are run as soon as their inputs are available, taking precedence over checkers, they follow the same profile but it is scaled down horizontally because the derivers take less time to run. Where there is no deriver work to be done, checkers fill the gap. At the very beginning, the checkers for some leaf functions run in parallel with the derivers for the leaves, saturating the workers. The saturation stays just before the end when it drops off non-instantaneously simply because in the last batch of functions some will finish before others. Not only does the end-to-end computation finish faster in schedule B, but it has unexploited parallelism, and so could finish even faster if more workers were provided. In contrast, schedule A was only briefly able to exploit all of its available workers.

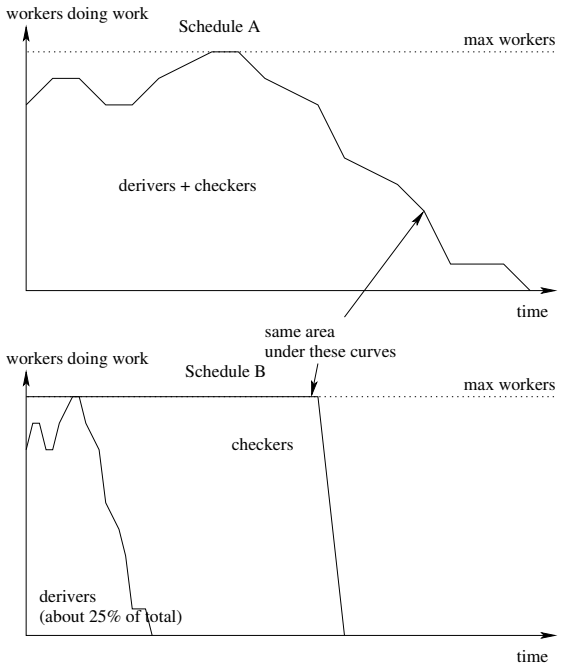

Figure 6: Derivers and checkers have the same priority in Schedule A. Derivers have higher priority over checkers in Schedule B.

Originally, we also surmised that it would be important to accurately estimate the time to analyze each node and weight the path cost accordingly. In particular, we tried using combinations of AST size and cyclomatic complexity to estimate analysis cost. However, experimentation showed that simply using unit cost per node performed about the same as any weighted cost, so we currently use simple path length as the cost of a path.

The critical path length of each node is also supplied to the distributor master when a work unit is enqueued, and Q1 is maintained as a priority queue. When Q1 is not empty, and some worker has fewer than two outstanding work units, the next highest-priority element of Q1 is sent to that worker. This way, a node with a high critical path cost but whose predecessor results were delayed for some reason can "jump ahead" of lower-cost nodes already in Q1.

2.3 Incremental Analysis

In the incremental analysis use case, an initial analysis is run, then the code to be analyzed modified and rebuilt, then a second analysis is run. The goal is to produce the same results on the second run as would be produced by a run from scratch, as quickly as possible.

The work unit design naturally supports incremental analysis: the map from work unit to work unit result is deterministic, so we simply maintain a cache across analysis runs. When a work unit is already in the cache, its result is available immediately, so the abstract interpretation is skipped; otherwise it is computed normally. The cache treats work units as opaque blobs when checking for equality.

Consequently, making this cache effective across code changes requires considerable care in work unit design, since otherwise it could be that a small change somewhere would cause slight differences in a large number of work units, and hence those units would have to be recomputed. To illustrate the problem, let us consider what happens if line numbers are included in work units. Line numbers are needed eventually in order to report meaningful defects, and defects are constructed from work units, so a naive implementation would include them. But then if the user adds (say) a blank line to the top of the file that contains the definition of a custom allocator, the summary of that allocator will have a different line number in its reporting structures. That will propagate to all functions that allocate memory, which is usually a large fraction of all of them. Consequently, incremental analysis would re-analyze that fraction of the program, all because of one blank line.

To solve this, defect reporting information is factored out of the packaged work unit and held on the master side while the work unit is processed: each source location is an offset relative to some stable anchor point, like a function definition or class member declaration. When the result comes back, the master computes the actual file names and line numbers based on the locations of the anchor points.

Similarly, when a defect or function summary makes use of interprocedural information to draw a conclusion, such as a lock being held or a function requiring that its return value be checked, it cites abstract evidence identifiers in the work unit result. When these conclusions are included in subsequent work units, they too are made abstract, carrying behavior semantics but little or no defect reporting information. Only the analysis master process, which has relatively little computation to perform, is exposed to the defect reporting details that are sensitive to "small" program changes; and the master process takes care of managing the abstract identifiers and correlating them across work units to assemble complete defect reports.

3. EXPERIMENTS

We implemented this design in the Coverity Static Analysis [7], a mature commercial static analysis tool. We did not change its output: its accuracy and thoroughness were unaffected by the use of the parallel and incremental design presented here. With the new design it merely runs faster, while (as before) it usually reports about one crash-causing defect per thousand lines of code, with a false positive rate of 10–20%.

Table 1 shows a selection of open source code bases with their sizes, analysis times with one worker process and eight worker processes, speedup factor and defect counts. All code bases were analyzed with default analysis options on an 8 core machine running Linux 2.6.18-128.el5 with two Xeon E5530 (4 core) 2.40GHz processors, 16GB RAM and a SAS 146GB 15k RPM disk. We ran the analysis with a subset of the total number of checkers (the set that is turned on by a default analysis run) and varied the number of workers that is available for each analysis run.

The size of the codebases varies from 1.13K to 6.7 million lines of code. The number of functions ranged from 13K to 510K and the analysis times with one worker varied from approximately 20 minutes for `postgresql` to half a day for analyzing `openoffice`. The analysis detected a large number of defects – 1190 for `postgresql` to 69206 for

Table 1: Benchmark Information.

code base	version	Language	LOC (in millions)	Number of functions	Analysis time (in secs)		Speedup with eight workers	Defect count
					one worker	eight workers		
postgresql	9.2.1	C	1.13	13550	1189	170	6.99	1190
wine	0.9.55	C	1.72	49359	3746	557	6.72	1471
xc	6.6	C	2.11	46953	6048	837	7.22	4228
linux	3.6.3	C	5.20	178344	8217	1675	4.90	4251
openoffice	2.4	C++	6.77	510064	44209	14981	2.95	69206

`openoffice`. Most reported defects from this analysis cause program crashes, hangs, data corruption or runtime exceptions. Furthermore, the reported defects were exactly the same across multiple runs of the analysis with different parallelization factors.

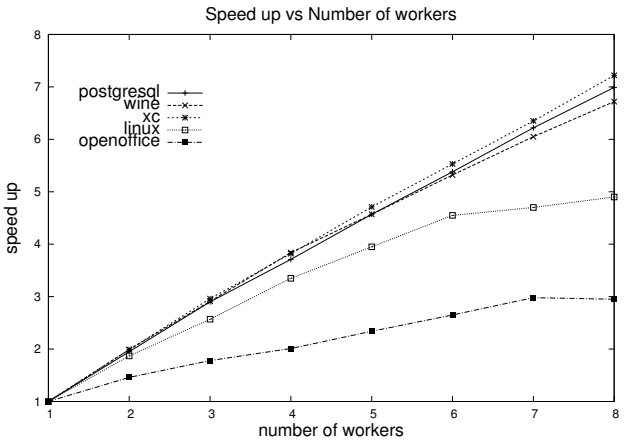

Figure 7: Analysis time with increase in number of workers.

Instead of relying on our own judgement to validate the precision of the defects found by our analysis, we used the evaluation of independent programmers familiar with the codebases to comment on the correctness of a defect. The website `scan.coverity.com` allows developers of open-source projects to review the analysis results for free and to rate the individual defects according to accuracy. For example, on an older version of `linux` [1], 5446 defects were inspected by the programmers out of which 300 were labelled as *false positives* and 323 as *intentional defects*. False positives are imprecise defects shown due to the imprecision of the analysis. While intentional defects are precise defects shown by the analysis, they are not necessarily fixed by the programmer because the *incorrectness* in the code was with a specific purpose (e.g., a crashing test). Since neither of these types of defects will result in a code change, we consider both cases as false positives. This results in a false positive rate of 11.4% for this code base, which is in line with the overall false positive rate of 9.7% for all of the Scan code bases [37]. We expect the accuracy of the defects found on the code bases used for the experiments in this paper to also lie within this range.

[1]All checkers except variants of coding style violations were enabled while analyzing `linux`.

The speedup achieved with parallel analysis is shown in Figure 7. In this figure, each analysis time is plotted with respect to the single worker analysis; that is, we plot the ratio of the single worker analysis time to the analysis time for the plotted configuration. For all `C` benchmarks, we observe a speedup of five to seven times over an analysis with just one worker. The speedup is also a function of the parallelism available in the call graph as observed by the differing improvements between `wine` and `xc` even though they have approximately the same number of functions. As the number of workers increase, on a larger benchmark like `linux`, the analysis stops getting faster once 6–8 workers are running due to bottlenecks in the master. On the other hand, `openoffice` with virtual call resolution enabled stops getting faster due to lack of available parallelism. Even though, the actual improvement in time of the analysis is significant, we still consider this as a limitation of our approach in scaling for C++ codebases with virtual call resolution and discuss this in further detail in Section 4.

The analysis has been deployed and applied on very large code bases. Practical experience shows that the analysis scales well with increase in the number of workers. When the analysis was applied on a ∼22MLOC (mostly C) industrial codebase with ∼27K files containing ∼204K functions, the analysis analyzed ∼171M paths to detect ∼32K defects. On a 24 core machine and 96GB RAM, the sequential analysis takes more than 200 hours. However, when parallel analysis was invoked with eight workers, the analysis time reduced to approximately 31 hours. As the number of workers increased to 16, the analysis time reduced to approximately 18 hours (at least 12 times speedup over the sequential analysis).

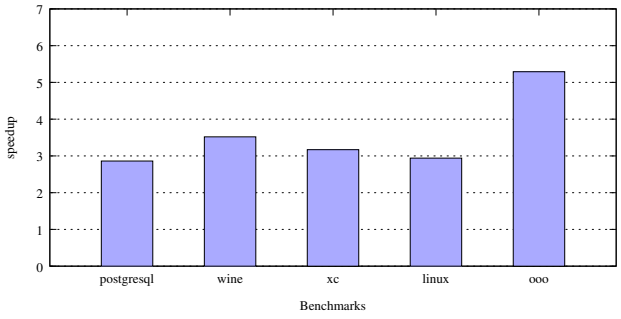

Figure 8: Comparison of the proposed design with older parallel-by-checker (PBC) implementation.

We also compared the design presented here with the older "parallel-by-checker" (PBC) feature of the product. The PBC implementation partitions the checkers, then invokes multiple sequential processes, each analyzing with one par-

tition of checkers. We analyzed all the benchmarks from Table 1 with the parallelization factor set to four (as the existing PBC implementation did not scale further). Figure 8 presents the ratio of analysis time with the PBC implementation to the time take for analysis using the approach presented in this paper using four workers in both cases. We observe a better scaling with the proposed approach and attribute the slowness in PBC to duplicate operations and the lack of scheduling of the analysis computations.

Table 2 shows statistics about work unit construction and processing time, and the sizes of work units and their results. Processing work units takes 2–3 orders of magnitude longer than construction (consumption of results is even faster), which helps maintain parallelism. Work units and results are small, usually under 10 kB, minimizing communication costs. Both times and sizes have a heavy tail, as mean and standard deviation are larger than median and interquartile range for all four measures.

Figure 9 shows a representative case of incremental analysis of a large code base. A historical version of the code, as recorded in the source control system, was selected at random. That version was built and analyzed sequentially, non-incrementally. Then, we incrementally built and analyzed the chronologically next 20 checked in versions. Over these 20 changes, an average of 4, median of 3 and maximum of 22 analyzed source code files were modified, and an average of 132, median of 20 and maximum of 1508 analyzed source code lines were added or modified. The changes were made by 12 different authors, checked in over a span of 3 business days of development work and address a wide range of feature and bug fix work. For comparison, we also show the time to do a non-incremental analysis of each version. The incremental analyses took on average just 15% of the time of a full analysis [2], demonstrating the effectiveness of the incremental analysis and work units' insensitivity to irrelevant changes.

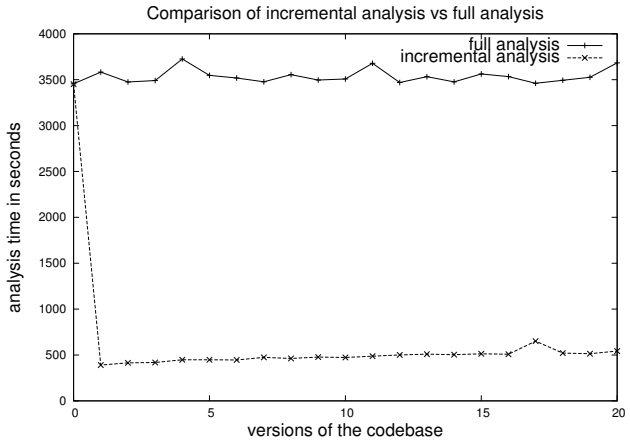

Figure 9: Performance of incremental analysis across 20 successive versions of a large codebase.

4. DISCUSSION

As the experimental results show, analysis of C++ code with virtual call resolution enabled shows reduced speedup in comparison to C code bases. One reason is that virtual function call resolution adds enough additional edges to the call graph. This creates more dependencies among the different nodes in the call graph and the available parallelism drops noticeably. Historically, we have focused on precise virtual call resolution only to reduce false positives. In particular, we could afford to wait to eliminate some of the resolutions until the abstract interpretation phase, when more information generally is available. But with parallel analysis, call resolution affects speed as well as precision, and to that end we are working on ways to do more of it in advance of call graph construction.

In addition to lack of available call graph parallelism, speedup is also limited by bottlenecks in the analysis master. This is particularly true for C++ code bases, which have more complex type hierarchies, which causes the master to spend more time loading types during work unit construction.

These effects can be seen in Figure 10, which shows a breakdown of the time spent in the analysis master during analysis of openoffice.org-2.4 with 8 workers. Sequential initialization includes the portion of virtual call resolution that is done before abstract interpretation. The master spends 16% of its time waiting for workers because of limited parallelism due to call graph density. 54% of time is spent constructing work units; the bulk of that is loading and processing interprocedural summaries, but a significant piece is loading class hierarchy fragments.

Task	Time fraction
Sequential initialization	7%
Wait for workers	16%
Construct WU	54%
Load function AST	1%
Load types	13%
Load summaries	32%
Serialize WU	2%
Other	6%
Consume WUR	22%
Store stats	2%
Store behavior summaries	5%
Insert into WUR cache	5%
Other	10%
Other	1%

Figure 10: Analysis master time breakdown for openoffice.org-2.4 with 8 workers. Indented lines are sub-tasks. All percentages are of the total time.

To improve speedup on C++ code, we are working on ways of further optimizing the analysis master and more precise virtual call resolution.

5. RELATED WORK

There are a number of static analysis techniques proposed for bug detection. Dillig *et al.* [15] propose a precise technique for a path-sensitive and context-sensitive program analysis. Das *et al* [12] present an approach for partial program verification in polynomial time. Engler *et al.* [16] propose an approach for writing system-specific compiler extensions that automatically checks the code for rule violations.

[2]The output of incremental and full analysis are identical.

Table 2: Work unit(WU) statistics analyzing linux-3.6.3.

	Min	Mean	Median	Max	Standard Deviation	Interquantile Range
WU construction time on master (ms)	0	4	3	723	6	4
WU processing time on worker (ms)	1	5,018	140	104,471	13,261	2,534
WU size (bytes)	200	6,159	4,233	314,898	7,705	4,816
WU result size (bytes)	10	748	660	12,737	509	537

This is in contrast to writing abstract specifications that are then verified by model checkers [24] or theorem provers [13]. In a follow up work, they also present their evidence of using the static analysis machinery on billions of lines of code [7] pointing out to the importance of reducing false positives and analysis times for static analysis tools to be adopted. There are also other approaches that infer common behavior of a program [18] and identify the deviants as bugs. Ganapathy *et al.* [19] provide a static analysis solution for detecting buffer overflows in C code by modeling string manipulations as a linear program. Our approach presented here is complementary to all these approaches. A primary goal for our work is to make interprocedural static analysis faster and on designing a system to apply the analysis continuously as part of the nightly build cycle on *any* codebase.

Recently, there has been some very interesting work on improving the scalability of program analysis [3, 29, 34]. Albarghouthi *et al.* [3] present an approach for parallelizing a top-down analysis and apply their approach on device derivers (approximately 25K LOC) to verify safety properties. Our approach combines top-down, bottom-up and global inference passes, and we focus on code bases that are several orders of magnitude larger. Lopes and Rybalchenko [29] present a distributed approach for predicate abstraction and refinement-based algorithm that is also deterministic. While we share common goals in terms of speed and determinism, the intended applications and the magnitude of its application vary. Prabhu *et al* [34] address the issue of slowness of higher-order control flow analysis and propose an algorithm for its acceleration with a GPU. Mendez-Lojo *et al* [31] propose a parallel analysis algorithm for inclusion-based pointer analysis and show a speed up of upto 3x on a 8 core machine on code bases with size varying from 53KLOC to 0.5MLOC. In comparison, we show better speedups on much larger codebases, albeit for the purpose of bug detection.

Counterexample-guided abstraction refinement [6, 8, 5] techniques use progressively better models of the program that is being analyzed to detect errors accurately. While these techniques may find potentially more accurate and even find deeper defects in a few cases, they are not necessarily scalable and very hard to determine in advance if they will work well for a given program. Moreover, the approaches are optimized to suit a specific codebase to verify a specific program property. So, a minor change to the program can potentially increase the analysis time making it hard to incorporate such approaches into a development flow like a nightly build. Furthermore, from our practical experience analyzing industrial codebases, the usual constraints of using static analysis are to find defects that have the highest impact on quality to fix within limited time and resources rather than to detect a specific defect type in a expensive manner. The advent of parallel and incremental analysis with the nightly build model satisfies these constraints for a wide spectrum of codebases. So, while some of these approaches have been shown to be able to scale to analyze specific programs, our approach *routinely* analyzes programs up to 20+MLOC that it has never seen before without scalability issues.

Parfait [10] is another static bug detection framework where simple analysis techniques are employed to detect easily detectable bugs and expensive approaches used, if necessary for other defects. They apply their analysis on a subset of the SAMATE benchmarks [38] pertaining to two defect types – buffer overflow and read outside the bounds of an array. The average time taken for analyzing each benchmark is 0.2068 seconds [10]. In contrast, we have applied our implementation on large (industrial and open source) codebases and detect different types of defects scalably.

PREfix is a static analysis tool that attempts to find complex interprocedural defects and PREfast is its faster variant where precision is traded for scalability [32]. Therefore, while the latter can potentially show defects quickly, further annotations are essential to do a deeper analysis. The strength of our approach is that it is not only scalable but is also automatic in finding a large number of real, reasonably deep defects with a low false positive rate.

MapReduce [14] is applicable to massively scaling relatively simple computations that have a high degree of parallelism. This analysis is different because there are intricate dependencies dictated by the structure of the callgraph and the different phases of analysis. The amount of parallelism available doesn't justify using a cluster, as commodity multicore hardware has enough power (and memory) to take advantage of the available parallelism.

Burckhardt *et al* [9] present an algorithm that can leverage a small set of primitives to exploit parallelism and can be used for incremental purposes as well. While their approach is a general paradigm for applications, in this paper we have presented an approach that is used for parallel and incremental static analysis for bug detection and empirically show the performance improvement across multiple codebases.

Impact analysis [1, 25] is the problem of identifying the impact of a change on the program. It is essentially used to identify regression tests that need to be run so as to minimize the amount of time required for testing. Incremental analysis shares common goals with impact analysis, albeit for identifying relevant code that needs to be reanalyzed rather than for testing. Guarnieri and Livshitz [23] address the problem of analyzing just the client code in the absence of the entire program source and apply it for JavaScript clients. While the constraints are different for both the problems, our approach can be leveraged so as to just store work units and analysis results of the server code and subsequently just analyze the clients.

6. CONCLUSION

This paper presents a design for parallel, incremental and deterministic interprocedural static analysis that addresses three practical concerns: (a) faster analysis to detect complex defects precisely, (b) detect defects deterministically and (c) analyze incremental changes to code efficiently.

The keys to the design are a carefully constructed work unit structure and a robust message-based concurrent task infrastructure. The resulting analysis finds about one critical bugs per thousand LOC in C code, with less than 20% false positives. Our experiments demonstrate scalability and determinism of our approach across a wide spectrum of codebases. It runs in a few hours on code bases with millions of LOC on modern commodity hardware and always produces the same results for the same input.

Consequently, this analysis can be deployed as part of a typical nightly build cycle, ensuring that defects detectable with static analysis are consistently found and *fixed* before they can cause harm.

7. REFERENCES

[1] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 746–755, New York, NY, USA, 2011. ACM.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 43–48, New York, NY, USA, 2007.

[3] A. Albarghouthi, R. Kumar, A. Nori, and S. Rajamani. Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 217–228. ACM, 2012.

[4] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[5] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54(7):68–76, July 2011.

[6] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software, LNCS 2057*, pages 103–122, May 2001.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[8] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[9] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 427–444, New York, NY, USA, 2011. ACM.

[10] C. Cifuentes and B. Scholz. Parfait: designing a scalable bug checker. In *SAW '08: Proceedings of the 2008 workshop on Static analysis*, pages 4–11, New York, NY, USA, 2008. ACM.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[12] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002.

[13] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[15] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000.

[18] D. Engler, D. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[19] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 345–354. ACM, 2003.

[20] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[21] P. Godefroid, N. Klarslund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Il, 2005.

[22] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, HPDC '00, Washington, DC, USA, 2000.

[23] S. Guarnieri and B. Livshits. Gulfstream: staged static analysis for streaming javascript applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, Berkeley, CA, USA, 2010. USENIX Association.

[24] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[25] M.-A. Jashki, R. Zafarani, and E. Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 84–90, New York, NY, USA, 2008. ACM.

[26] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Seventh USENIX Symposium on Operating*

Systems Design and Implementation (OSDI), 2006.

[27] J. Lenstra and A. Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978.

[28] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 133–146, New York, NY, USA, 2012. ACM.

[29] N. Lopes and A. Rybalchenko. Distributed and predictable software model checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 340–355. Springer, 2011.

[30] I. Matosevic and T. S. Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 252–263, New York, NY, USA, 2012.

[31] M. Mendez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010.

[32] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM.

[33] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 815–816, New York, NY, USA, 2007.

[34] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. Eigencfa: accelerating flow analysis with gpus. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM.

[35] M. Pradel and T. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 521–530. ACM, 2012.

[36] M. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Conference on Programming Language Design and Implementation: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 123–134, 2007.

[37] Whitepaper: Coverity scan: 2012 open source report. `http://scan.coverity.com`, May 2013.

[38] http://samate.nist.gov/.

[39] B. Sun, G. Shu, A. Podgurski, and B. Robinson. Extending static analysis by mining project-specific rules. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1054–1063, Piscataway, NJ, USA, 2012.

[40] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16, 2007.

[41] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 485–502, New York, NY, USA, 2012. ACM.