Assessing the State and Improving the Art of Parallel Testing for C

Oliver Schwahn
os@cs.tu-darmstadt.de
TU Darmstadt, Germany

Nicolas Coppik
nc@cs.tu-darmstadt.de
TU Darmstadt, Germany

Stefan Winter
sw@cs.tu-darmstadt.de
TU Darmstadt, Germany

Neeraj Suri
suri@cs.tu-darmstadt.de
TU Darmstadt, Germany

ABSTRACT

The execution latency of a test suite strongly depends on the degree of concurrency with which test cases are executed. However, if test cases are not designed for concurrent execution, they may interfere, causing result deviations compared to sequential execution. To prevent this, each test case can be provided with an isolated execution environment, but the resulting overheads diminish the merit of parallel testing. Our large-scale analysis of the Debian Buster package repository shows that existing test suites in C projects make limited use of parallelization. We present an approach to (a) analyze the potential of C test suites for *safe concurrent execution*, i.e., result invariance compared to sequential execution, and (b) execute tests concurrently with different parallelization strategies using processes or threads if it is found to be safe. Applying our approach to 9 C projects, we find that most of them cannot safely execute tests in parallel due to unsafe test code or unsafe usage of shared variables or files within the program code. Parallel test execution shows a significant acceleration over sequential execution for most projects. We find that multi-threading rarely outperforms multi-processing. Finally, we observe that the lack of a common test framework for C leaves make as the standard driver for running tests, which introduces unnecessary performance overheads for test execution.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Static Dependency Detection, Parallel Test Execution, Software Repository Analysis, Static Analysis

ACM Reference Format:
Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. 2019. Assessing the State and Improving the Art of Parallel Testing for C. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330573

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330573

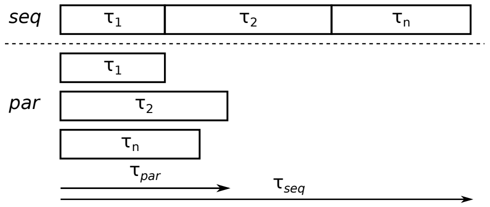


Figure 1: Illustration of the intended achievement. Execution time τ_{par} for the parallel case is defined by the longest executing test case, whereas for the sequential case τ_{seq} it is defined by the sum of all test case execution times.

1 INTRODUCTION

To obtain high test throughput and limit the influence of human error, dynamic software tests are themselves commonly implemented as software for test automation. As the amount of test code has exceeded that of the application logic by far for numerous projects [9], its execution time is critical for the performance of various steps in software development and maintenance. For maintainability and selective execution, the test code is organized as collections of test cases in *test suites*. With the increasing parallelism of modern processors, test execution times can only benefit from increasing computational power if test suites are designed for concurrent execution. The total execution time of a test suite consisting of test cases $t_1 \ldots t_n$ with execution times $\tau_1 \ldots \tau_n$ would be reduced from $\sum_{i=1}^{n} \tau_i$ in the sequential case to the execution of the longest running test $\max(\{\tau_1 \ldots \tau_n\})$, as illustrated by Figure 1, if two conditions hold: (a) sufficient parallel processing units are available, and (b) all tests in a test suite are *independent*.

Unfortunately, this assumption of test case independence within a test suite has proven problematic [22]. Even sequential executions of a test suite can lead to differing test case results across different permutations of their execution order. The major root causes behind test dependencies found in existing software projects have been identified as (a) shared heap memory and (b) shared files [22]. While the former has been identified as the most common reason for test dependencies (62.7 % of all dependencies analyzed in [22]), it is only problematic if test executions share the same memory address space. Isolating tests in individual processes would, therefore, solve a substantial portion of the problem, but reportedly induces significant overheads on test executions [1]. As shared files affect any tests operating on the same file system, file dependencies need to be identified irrespective of address space isolation.

In this paper, we explore several implementation alternatives (with different degrees of memory isolation) to achieve safe parallel executions of existing sequential test suites for projects written in C. By *safe* parallel execution, we mean that the results of test cases executed in parallel *cannot* differ from the results of their original sequential execution order. We focus our work on C, the predominant language in the Debian main package repository (as we will show in Section 3). C features the second highest test count across projects hosted on GitHub according to a study of Kochhar et al. [12]. To check if tests can interfere in parallel execution, we implement two static analyses based on the LLVM compiler infrastructure [14]. The decision to focus on existing test suites is motivated by the large amount of existing sequential test code that is shipping with widely used software.

Our paper makes the following contributions.

- We present an analysis of Debian Buster's main package repository showing that the majority of code contained in the packages is written in C, that no test framework dominates test implementations for C packages, and that few test suite implementations benefit from concurrent execution.
- We develop automated static analyses for C code to find test case dependencies on files and shared global data to identify which parts of a test suite can safely execute in parallel.
- We develop a test harness to use this information for safely executing tests in parallel and explore the trade-off between address space isolation and parallel test suite performance in different parallelization alternatives using processes and threads for nine Debian source packages.
- We present the results of an in-depth analysis of nine software projects from Debian Buster, for which we parallelize test execution using our dependency analyses and test harness. Our results show that test suites in C can benefit from even modest degrees of parallelism provided by virtually every desktop or server hardware, that threads do not perform significantly better than processes, and that our test harness (and likely any specialized test tool) outperforms generic automation tools like `make`.

2 RELATED WORK

The goal of our work is to assess if the concurrent execution of tests in C projects can achieve better latencies without compromising test outcomes. Articles related to our work fall in three categories: (1) same objective and mechanism (concurrent execution for latency improvement), (2) same objective, but different mechanisms (latency improvements by other means), (3) similar mechanisms (test interference detection), but different objectives.

In summary, only one existing approach, VmVm [1], does not require the execution of tests. Approaches based on dynamic analyses suffer from the need to execute the test suite at least once. After the test suite has been executed, the test results are known and there is no benefit in obtaining them again, no matter with which run time improvement. Hence, to detect test interferences for safe concurrent execution, we need to use static analysis. As we cannot reuse VmVm's analysis, since it runs on Java code, we develop static analyses of accesses to global variables and shared files for C programs.

Concurrent Test Execution for Latency Improvement. Early approaches for concurrent test executions [5, 15, 17, 19] *assume* test cases to be independent and do not analyze if their parallel execution can alter test results. As test dependencies were found to affect permutations of test sequences [13, 22], newer approaches address the possibility of test dependencies.

ElectricTest [2] identifies dependencies in Java tests to determine which tests need to execute in sequence to prevent spurious results. Dependencies are derived from traces of shared resource accesses gathered during test execution. Lam et al. [13] assess the impact of dynamically detected test dependencies in Java projects on test parallelizability, achieving speedups between 1.02× and 7.14×.

CUT [8] executes unit tests in parallel and isolates them in separate virtual machines or Docker containers to ensure that concurrently executing tests *cannot* interfere. CUT relies on external input in the form of a directed acyclic dependency graph, which can be provided by analyses like those presented in this paper.

O!Snap [7] uses VM snapshots to speed up test execution. To avoid missing libraries or setup steps for running the tests, O!Snap analyzes dependencies on the software package level. Our approach is orthogonal, as it targets concurrency of tests within a package, as opposed to concurrency across packages.

Candido et al. [3] investigate how common concurrent test executions are in open source projects. Their results show that only 13 out of 110 investigated Java projects execute tests concurrently. The authors experimentally assess the speedup (up to 75.9×) and the rate of spurious test failures (up to 96.3 %) of naive parallelization that ignores dependencies, emphasizing the need for dependency analyses for test parallelization. Our complementary study for C projects in the Debian Buster repository confirms the finding that few projects can benefit from parallel testing out of the box.

Improving Test Latencies without Concurrency. We found only one project that, similar to parallelization, achieves latency improvements without omission of tests. VmVm [1] reduces the execution latency of sequential test suites by replacing costly per-test initialization and termination routines with lightweight reset routines that are sufficient to provide non-interference across consecutive tests. To identify which part of the software under test's (SUT) state needs to be reset, VmVm uses a static analysis to identify heap memory that is possibly accessed by multiple tests.

Test Interference Detection. Another reason for analyzing test dependencies is the identification of bugs in test code. If individual tests are supposed to be independent from each other, any dependency indicates a bug.

Muşlu et al. [16] propose to execute tests in isolation to reveal dependencies on other tests and report an actual bug in Apache Commons CLI using this technique.

DTDetector [22] permutes the execution order of Java test suites to identify test dependencies via static fields. To limit execution overhead, DTDetector samples permutations using different algorithms, one of which uses test independence information to filter permutations that cannot reveal test dependencies. To gather dependency information, DTDetector executes each test once in isolation.

PraDet [6] detects manifest test dependencies with a similar approach as DTDetector, but reduces false positives by using an enhanced version of ElectricTest's [2] dependency detection.

PolDet [10] detects *state pollutions* of shared state across Java tests by identifying shared heap memory regions at run time and tracking accesses to them. PolDet also tracks modifications to files, but relies on user input for identifying which files are relevant.

3 EMPIRICAL STUDY: C SOFTWARE IN DEBIAN BUSTER

In our literature review (Section 2), we made the observation that all existing work on test dependencies is focused on Java projects. While we do not speculate about the reason, we needed to confirm that it is *not* because testing of C code is an irrelevant problem. For this purpose, we analyzed the entire main package repository of the upcoming "Buster" release (version 10) of the widely used Debian Linux Distribution [21] with three major objectives: (1) To assess the amount of C code compared to other languages to confirm the relevance of our work, (2) to assess which test frameworks are most widely used to test C code, (3) to assess to which degree parallel execution is able to improve test performance.

3.1 Programming Languages in Debian

We downloaded and unpacked all 25 684 source packages available in Debian Buster [20]. To determine both the programming languages and the amount of source lines of code (SLOC) per package, we used `cloc` [4] and excluded markup languages such as XML.

Figure 2a shows the total number of packages by their predominant language, i.e., the languages that contribute most SLOC to the respective packages, and the relative contribution of each language to the entire repository. With almost 25 %, C is the most prominent language across all packages. To affirm that this finding is not biased by differing amounts of code in the packages, we also accumulated the SLOC number per language across all packages as shown in Figure 2b. With around 250 million SLOC, more than 28 % of the total code in the repository is written in C. This number excludes code in header files, as `cloc` cannot distinguish whether they belong to C or C++ code, and is, thus, a conservative estimate.

C is the dominant language in the Debian ecosystem.

3.2 Test Frameworks

To analyze the use of test frameworks, we scanned the downloaded sources for JUnit usage in the case of Java and for the presence of typical files and directives of 34 different freely available test frameworks[1] in the case of C. Figure 2c summarizes our findings. We found that, with less than 5.5 %, few C projects make use of any of the test frameworks. This is in strong contrast to the situation for Java, where over 65 % of projects use the de facto standard JUnit.

No test framework is commonly adopted for C software in the Debian repositories.

3.3 Test Parallelization

To detect if packages can benefit from parallel test execution, we identify all packages that show indications for the presence of any tests by scanning for file and folder names that include "test" as substring. By invoking typical build and test execution targets of GNU `make`[2], we then build each of these packages, execute their test suites, and measure the tests' execution times for varying degrees of execution parallelism specified via `make`'s `-j` flag. We repeat our time measurements three times per configuration to account for possible variations due to factors that are not under our control.

Out of 6419 C packages in the repository, we identified 1617 to show indications for the presence of tests. Out of these, 627 completed our measurement without failure for all three runs. Most packages that failed did so consistently in all three runs (99.2 %). A remaining 8 packages exhibited flaky build or test behavior. Half of them had test failures in the parallel case, despite successful sequential test executions. We also observed such behavior among 10 packages that failed consistently in each of the three repetitions.

From the 627 non-failing packages, only 177 (28 %) had shorter test execution times for the parallel case in all runs. 261 packages (41.3 %) had equal or longer test execution times compared to sequential execution. The remaining packages did not yield clear results, with parallel test performance sometimes exceeding the sequential case and sometimes vice versa.

The achieved test execution time speedup factors for the 177 benefiting packages are shown in the bubble plot in Figure 3. From the plotted data we observe that the degree by which projects benefit from parallel test execution varies greatly. While it is not surprising that longer sequential execution times (on the x axis) tend to coincide with bigger time savings (bubble size), it is remarkable that even projects with short test suite execution times between 250 ms and 1 s can achieve speedups well above the median of 1.37.

If the degree of parallelism for test execution is increased from 4 to 8, we observe only modest additional speedup, as indicated by the bubble colors, for the majority of the projects. Almost 60 % fall in the lowest category and 30 % in the second lowest. More than half of the projects in the lowest category have a speedup of 1 or less, i.e., they do not benefit from increased parallelization.

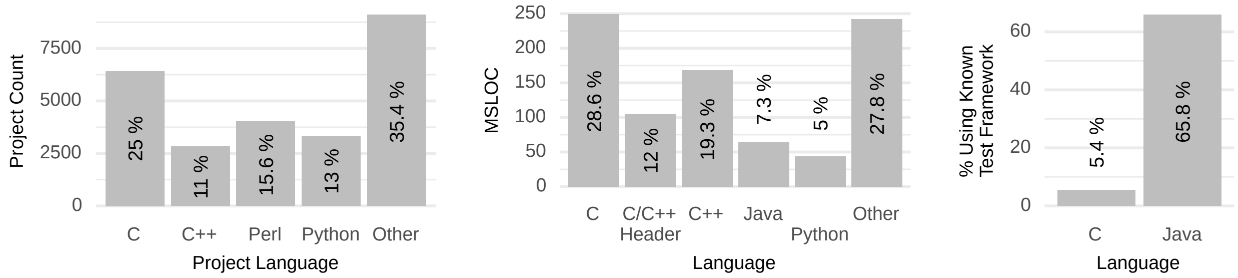
Test parallelization via `make -j` works for less than 39 % of C packages using `make` for test execution. Most of these packages do not deterministically benefit from 4-fold parallel test execution. Few of those that do benefit from further increased parallelism.

3.4 Threats to Validity

We do not claim that the results from our study apply for other ecosystems. With Debian, we target a large ecosystem that forms the basis of many production software stacks [21]. While this choice guarantees practical relevance, there are potential inaccuracies in our analyses resulting from the need to scale them to an ecosystem of significant size and projects with limited support for automated analyses. Our analysis of dominant languages relies on `cloc`'s accuracy, which is widely used for SLOC counting. Our analysis of test frameworks depends on the list of frameworks we searched for in the projects and the accuracy of our search heuristics. Similarly, the conclusions from our test run time analysis may depend on our build and test automation. Our conclusions are drawn from three repetitions of the run time analysis. We have used the coefficient of variance as a rough measure to detect massive instabilities, which we only found in one case of averaged time differences for 8-fold parallelism and which we excluded from the analysis. The exit codes observed were stable across the three conducted runs in more than 99 % of cases, which adds to our confidence in the absence of massive deviations from the reported results.

[1] The full list is included in our accompanying artifact.

[2] We also invoke typically found configuration steps such as `autoconf` or `configure` and try different `make` targets for executing tests such as `check` or `test`.



(a) Number of packages by dominant language.

(b) Millions of source lines of code (MSLOC) by language.

(c) Usage of known test frameworks in C- vs. Java-dominated packages.

Figure 2: Results of Debian "Buster" package repository analysis. SLOCs were counted with `cloc` [4].

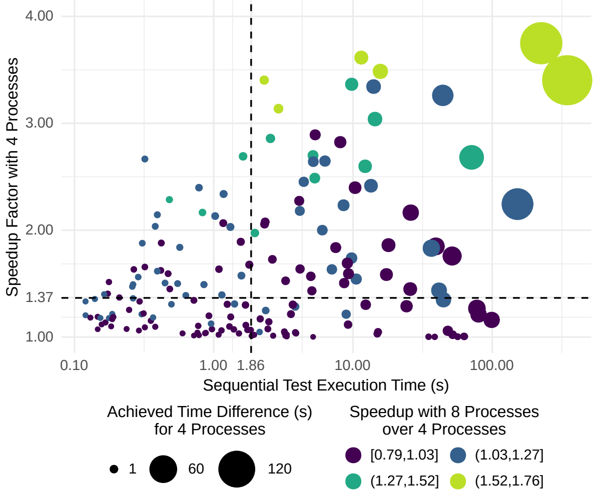


Figure 3: Achievable test speedup for C packages in Debian Buster. The dashed lines indicate the median sequential test execution time and the median speedup achieved with 4 processes. The bubble sizes indicate time differences between sequential and 4-fold parallel execution. The color coding shows additional speedup achievable by 8-fold parallelism.

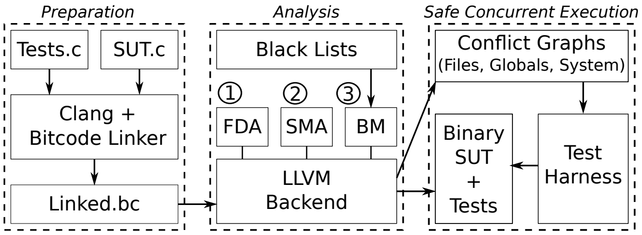


Figure 4: For our analyses, the tests and the SUT have to be compiled to bitcode and linked. After running our analyses ① – ③, which are implemented as LLVM compiler passes, we obtain information on potential test conflicts. Our test harness leverages these to derive safe parallel test schedules.

4 SAFE CONCURRENT TESTING FOR C

Our empirical study has shown that only a small fraction of those C projects in Debian that invoke tests via `make` benefit from parallel test execution. In the following we present an approach to (1) assist C developers with the implementation and maintenance of concurrent test suites and (2) enable safe concurrent test executions for legacy test suites that have been designed for sequential execution.

Figure 4 provides an overview of our approach. The three phases of preparation, analysis, and safe concurrent test execution are discussed in the following subsections.

4.1 Preparation

We implement our analyses as LLVM optimizer passes that perform a whole-program analysis on the tests and the SUT. For this purpose, we require the tests and the SUT to be compiled to LLVM bitcode (LLVM-IR) and linked together. Everything that is not linked in at the point at which our passes run is deemed external and any test inter-dependencies due to external resources must be addressed via a *blacklisting mechanism*, which is discussed in Section 4.2.3.

Our analyses assume test cases to be *self-contained*, i.e., not to rely on external inputs. External inputs are either generated by human testers or by external test automation tools written in other languages. If the test suite relies on human input, its potential performance gain from automated parallelization is limited. If input data is generated by tools written in other languages, those parts of the test harness would require an analysis engine for those languages. If any input generating code can be linked with the LLVM-IR of the tests, our approach can include it in the analysis.

4.2 Detecting Potential Test Interference

Concurrent test executions can interfere if two or more test cases access the same data, at least one such access is modifying that data (write access), and the test outcome of at least one other test depends on that data. Which data is shared among concurrent tests depends on their execution environment. Concurrently executing tests in separate *processes* (as in the case of `make` in Section 3) share the same operating system state (e.g., system wide configurations like the locale) and in particular the same file system, but not the same memory. Dependencies on shared memory only affect concurrent tests if they execute as threads within the same process context. We developed separate static analyses to detect potential dependencies (due to global variables or file system usage) in a given test suite, because of these different parallelization strategies they enable.

We chose a static approach for analyzing potential dependencies over a dynamic approach since static analyses have the advantage that the analyzed tests do not need to be executed. A dynamic analysis would already produce the desired test results, limiting the utility of the approach to cases for which a repeated execution of the same tests in the same configuration is desirable. Static analyses can be integrated into the software build process which ensures that the used dependency information always matches that of the produced test executables. This integration is especially useful if a software project has many build-time configuration options, which may influence test dependencies.

4.2.1 Analysis ①: File Dependency Analysis (FDA). To detect file dependencies, our analysis first checks whether certain known functions that are used to interact with the file system, such as `fopen`, are reachable from a test case by constructing the static call graph for the SUT and traversing it for each test case's SUT invocations. Then, for each call site of such a function that is reachable from at least one test case, we traverse use-definition chains to determine which (constant) file names may be passed to the function. A test case t may access a file f if a call site of a file processing function is reachable from t and f is a reaching definition for a function argument at that call site. We use the same technique for mode arguments to distinguish read-only accesses from writing accesses. The resulting file read and write sets $F_r(t)$ and $F_w(t)$ for each test case t can be used to detect dependencies between any pair of test cases and we construct an undirected test case conflict graph $C_F = (V, E)$ as follows:

- For each test case, we add a corresponding vertex to V.
- For each pair of vertices $t_i, t_j \in V$, we add an edge to E iff $(F_r(t_i) \cap F_w(t_j)) \cup (F_r(t_j) \cap F_w(t_i)) \cup (F_w(t_i) \cap F_w(t_j)) \neq \emptyset$, i.e., when there is a possibility of accesses to the same file including at least one write operation.

Our current analysis does not identify potential file accesses for files with dynamically generated file names, but could be extended to track string operations that are used to construct such names.

4.2.2 Analysis ②: Shared Memory Analysis (SMA). Analogous to Section 4.2.1, we construct the static call graph of the SUT and the tests. We then follow the definition-use chains of all global variables of the SUT, as well as function arguments in cases where global variable addresses are passed as parameters, to identify which of them may be read or written in which test case. We consider it sufficient to focus on global variables, because (1) global variables are implicit heap allocations and shared among threads, (2) function-local variables are allocated on the stack and are, thus, thread-local and not shared among concurrent threads, (3) to share explicitly allocated heap data (e.g., via `malloc`), threads need to communicate its addresses, which is only possible via previously shared memory.

Our analysis does not identify shared memory accesses to hard-coded constant-value addresses. Such accesses constitute a severe risk to memory safety and must be considered bad practice for commodity systems. For embedded systems there may be cases of software containing hard coded addresses. For these scenarios, our analysis would need to be augmented with a (straight-forward) mechanism to analyze constant propagation.

The result of our analysis is a mapping that assigns to each function f in the module its read and write sets of global variables $G_r(f)$ and $G_w(f)$. A test case t may read or write a global variable g if any of the functions reachable from that test case according to the static call graph may read or write g. Thus, the set of global variables that may be read (or written) during execution of t can be computed as $G_r(t) = \bigcup G_r(f_i)$ and $G_w(t) = \bigcup G_w(f_i)$ of all functions f_i reachable from t. The resulting read and write sets for each test case can then be used to detect dependencies between any pair of test cases by constructing the conflict graph C_G for global variables analogous to C_F in Section 4.2.1.

4.2.3 Analysis ③: Blacklisting Mechanism (BM). As previously mentioned, we rely on a blacklisting mechanism to model test dependencies on shared system resources besides files and memory. This mechanism takes a list of functions as input that access such shared resources, along with additional information on whether the access is reading or writing the shared resource. We analyze the test cases and the SUT for call sites of these functions and create read and write sets of shared resources for each function in the module, analogous to how we handle global variables. We then reuse the static call graph constructed during the shared memory analysis to determine which of the identified call sites can be invoked during test execution. The resulting conflicts are added to C_G, thereby effectively modeling them as global variables.

4.3 Concurrent Test Execution

The orchestration of test executions is generally implemented in some *test harness*. As we found in our empirical study in Section 3, C projects frequently use the general purpose build automation tool `make` for this purpose. We implement a custom test harness in our work to achieve the concurrent execution of test cases. Our test harness supports different parallelization strategies that make use of the dependency information extracted by our static analyses to prevent test interferences.

In general, there are two options for concurrent test executions, which differ in their risk of interfering test executions and their run time overhead: (a) executing tests in parallel, isolated *processes* or (b) executing tests in parallel *threads* without memory isolation.

Option (a) provides isolated address spaces, which eliminates memory interferences for parallel tests. Option (b) does not provide this isolation, but potentially offers lower overhead compared to (a) since thread management operations are generally considered lightweight and do not have to create/switch address spaces for isolation. However, since option (b) lacks program state isolation for each test, all tests must be analyzed for their interference potential before they can be run in parallel. For either option we implement two execution strategies in our test harness.

4.3.1 Multi-Process Strategies. Our first strategy (P_M) merges all test cases into one program and forks a new process for every test case. The only difference of this harness compared to `make` is that our implementation does not schedule two tests for concurrent execution if they have file dependencies, as this may lead to deviating

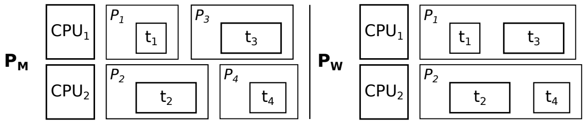


Figure 5: Multi-Process strategies P_M and P_W: For P_M a clean process is started for each test, whereas for P_W the repeated process initialization overhead is saved by reusing processes as long as tests do not have dependencies.

test results. The maximum number of processes in P_M is configurable to prevent resource contention from adversely affecting test suite execution times, e.g., when the number of processes is much larger than the number of CPUs in the system.

The other option for test parallelization with processes is a *worker model* (P_W) that forks a fixed number of processes, each of which executes several tests in sequence. This option avoids spawning new processes (similar to VmVm [1]) when sequential tests do not have dependencies and, thus, cannot interfere. Tests with file dependencies cannot execute in parallel, as previously explained in the discussion of P_M, and cannot execute sequentially within the same worker process if they have dependencies on common globals.

Figure 5 illustrates the difference between P_M and P_W in an example of four independent tests $t_1 \ldots t_4$ executing on two processing units CPU_1 and CPU_2. For P_W, two processes P_1 and P_2 are spawned, whereas a new process is created for each test in P_M.

4.3.2 *Multi-Thread Strategies.*

We employ two multi-thread strategies T_M and T_W analogous to P_M and P_W: T_M creates a new thread for each test case and T_W uses worker threads. In addition to the dependency restrictions described for processes, tests cannot be executed concurrently or within the same worker thread if they have dependencies on global variables. This restriction does not apply for processes, as they execute within separate address spaces and do not have access to other processes' global variables.

Multi-threaded strategies, therefore, require both dependency analyses, but are expected to outperform their multi-process counterparts in terms of test execution times, because of the lower overhead for thread creation and context switching.

4.4 Scheduling Concurrent Test Execution

We use C_G and C_F to schedule safe, concurrent test execution according to the four parallelization strategies discussed above. For P_M, scheduling relies only on C_F. P_W, T_M and T_W all require both C_F and C_G. We use C_F to partition the set of test cases as follows: We greedily pick and remove maximal independent subsets of test cases I_i from C_F until C_F is empty. For P_M, these sets are directly used for concurrent test execution: Test cases from the same set are executed concurrently in different processes at the chosen degree of parallelism. Different sets are handled sequentially, and test cases from different sets are never executed concurrently. In the other cases (P_W, T_M, T_W), we extract for each I_i the corresponding induced subgraph from C_G. The result is a set of conflict graphs C_G^i that encode potential memory and environment conflicts among tests that do not have file conflicts. These graphs are then used to identify sets of independent tests that can safely execute concurrently (respectively, sequentially within the same process), analogous to how C_F is used in the case of potential file conflicts.

5 EVALUATION

In our evaluation, we address the following questions.

RQ 1 What are the steps involved to transmute legacy tests suites for our approach and how much manual effort is required?

RQ 2 What kinds of dependencies do our analyses detect and where do they originate?

RQ 3 How high are the achieved speedups for parallel test suite execution and does execution with threads perform better than with processes?

RQ 4 How much overhead does the proposed dependency analysis impose and does the overhead amortize with the achieved execution speedups?

5.1 Experimental Setup

5.1.1 *Software Project Selection.*

We investigate our research questions by applying our approach to 9 real world software projects that are included in the Debian Buster software repository. We selected the projects to cover a large range of project sizes, test suite sizes, and sequential test execution times, as shown in Table 1.

5.1.2 *Experiment Execution.*

We ran our file and globals dependency analyses on each project and recorded the resulting dependency graphs. We executed the test suites at 6 different degrees of parallelism (1, 2, 4, 8, 16, 32) and in 5 different execution modes to assess how test duration changes. The test outcomes did not deviate between sequential `make` and our execution modes, i.e., we observed the same test results as for `make`. We repeated all our experiments 30 times and discuss mean values throughout this section.

5.1.3 *Execution Environment.*

We conducted our experiments on a machine with Debian Buster (Linux 4.17, x86_64), which is equipped with an AMD Ryzen 7 1700X CPU with 8 physical and 16 logical cores (3.40 GHz), 32 GiB of main memory, and a 1 TB SSD.

5.2 RQ 1: Transmutation of Legacy Tests

To answer RQ 1, we report how we prepared the tests of the 9 evaluated projects (cf. Section 4.1) and which manual and automated steps were involved.

First, we manually identify the test suite and its test cases. We exclude tests that rely on external tools or scripts written in languages other than C as these are not accessible to our analysis, as well as tests that deterministically fail in our execution environment or rely on external inputs (e.g., network). To allow a fair comparison between process-based and thread-based parallelization, we also remove tests that cannot be executed together within the same process, e.g., because they close standard file descriptors such as `stdout` or otherwise corrupt their environment (e.g., sending process termination signals). We document the original number (*Total*) and the number of test cases included in our study (*Analyzed*) in Table 1. Moreover, we verify that each test has its own unique entry point to avoid naming collisions when merging them for analysis. We integrated an automated, semantics preserving source code transformation with Coccinelle [11, 18] in our tool

Table 1: Evaluated Software Projects: Each project is listed with the amount of C code, the number of all/analyzed test cases, the longest test case run time (in s), and the sequential test suite execution time with make and P_M. Column *Diffstat* lists the amount of required manual code changes. Columns *Files* and *Globals* list the number of conflict inducing files and globals found in total and inside test code. The analysis time columns list the mean time (over 30 runs) required to find these conflicts.

Name	Size	Test Cases			Seq. Time (s)		Diffstat	Files		Globals			Analysis Time (ms)	
	(C SLOC)	Total	Analyzed	Longest	Make	P_M	+/-/!	Tests	Total	Tests	BL	Total	Files	Globals
gnulib	204486	1130	908	4.0	23.5	12.0	130/0/65	5	5	5	4	15	86.35	556.35
libbsd	7182	16	12	55.9	56.2	55.9	70/0/15	0	0	0	1	1	0.46	1.53
libesedb	211882	22	22	<1ms	1.0	0.01	6/1971/60	0	0	0	0	0	5.10	5.92
libgetdata	96532	1649	1637	9.5	52.5	34.1	6253/875/264	36	36	1	0	4	15911.71	1106.98
librabbitmq	9833	6	6	<1ms	0.1	0.01	4/0/0	0	0	0	0	0	0.43	1.20
libsodium	26123	65	65	1.1	5.4	3.9	80/0/4	0	0	0	0	9	4.74	80.64
litl	2403	16	10	4.0	7.3	7.0	90/1/8	1	1	0	0	0	0.94	1.69
openssl	244048	548	29	0.9	2.8	2.6	83/0/9	0	0	0	0	88	28.57	47.53
sngrep	10381	10	10	1.8	11.6	11.3	708/0/16	0	0	0	0	0	0.87	1.97

chain that handles the common case of each test having its own `main` function by creating unique function names. In cases where a `#include` directive is used to share code for the `main` function (`libgetdata`, `sngrep`), we physically resolve the include before applying Coccinelle. Further manual and semi-automated steps are sometimes required to allow Coccinelle to correctly parse and process the C code. For instance, we had to resolve some preprocessor macros, either manually or using the `unifdef` utility (`libesedb`, `libgetdata`, `openssl`).

Next, we adapt the project's build system to produce a single bitcode file (for analysis) and a single shared object file (for test execution), both containing the library and test code, for which we developed general purpose scripts. To enable the linking into one file, we manually change the declaration of some global symbols to `static` to prevent name collisions as C does not support namespaces (`gnulib`, `libgetdata`, `litl`, `sngrep`). We then apply our analyses to assess the parallelization potential of the test suite. We use the diagnostics output of our analyses, including a set of reachable external functions, to construct a blacklist (cf. Section 4.2.3).

To allow the execution with our test harness, the assertion logic used in the tests needs to be adapted to communicate test outcomes to our test harness. To that end, we manually changed assertion macro definitions and implemented C headers to replace functions like `exit` or `abort`, which both terminate process execution and are often found as part of assertion logic in test suites to check test outcomes, to support execution modes other than P_M.

Of the 9 projects, only 2 (`gnulib`, `libbsd`) required blacklisting for external functions. Manual and semi-automated code modifications are usually required before our approach can be fully applied. Table 1 lists the total amount of textual code modifications for each considered project as diff statistic (number of added, removed, and modified text lines) from the `diffstat` utility (*Diffstat* column). Apart from `libesedb`, `libgetdata`, and `sngrep`, fewer than 200 text lines were touched. The higher number of changes for the three projects is due to the manual resolution of includes and preprocessor macros as discussed above, which is a straightforward mechanical task. Overall, we were able to convert the test suites in a matter of few days for each project, with the exception of `gnulib` and `openssl`, which took longer as `openssl`'s test suite makes heavy use of Perl scripting and `gnulib` includes many tests that touch low level system functionality such as raw file descriptors and process management, which is the reason why we had to exclude a higher number of tests for those projects. We expect that developers with intimate knowledge of a project and its tests could perform the conversion task considerably faster.

> Porting legacy test suites to our approach is feasible with reasonable manual effort and minor code modifications to the original test suites.

5.3 RQ 2: Dependencies

To assess which kinds of dependencies exist between different test cases and where these dependencies originate, we examine the results of our dependency analyses. Table 1 lists the number of conflict inducing files and globals found for each project.

We find file dependencies for three projects. For `gnulib`, the detected dependencies correspond to files that are in fact accessed during test execution but these accesses are benign (e.g., accesses to `/dev/null`, or attempts to open a non-existent file). Our analysis could be enhanced with a whitelist to account for such benign paths. We find substantially more conflicts for `libgetdata` as there is a small set of common file names used in virtually all test cases. This prevents concurrent execution, for our approach as well as for the `make`-based execution supported by `libgetdata`. Attempting concurrent execution while ignoring these dependencies we observed flaky test results across repeated test executions. For `litl`, we detect one file-based conflict between two tests, in which both tests access the same file. Ignoring this dependency causes flaky behavior in parallel `make`-based test execution. Since all file dependencies we detect originate in test code, only the test suites would require modification to enable further parallelization.

Globals dependencies are more common than file dependencies, and we detect them in five projects. Unlike file dependencies, most of them originate in the project itself, not in test code. Such conflicts in the project itself result from the use of global variables that are used in project code reachable from more than one test.

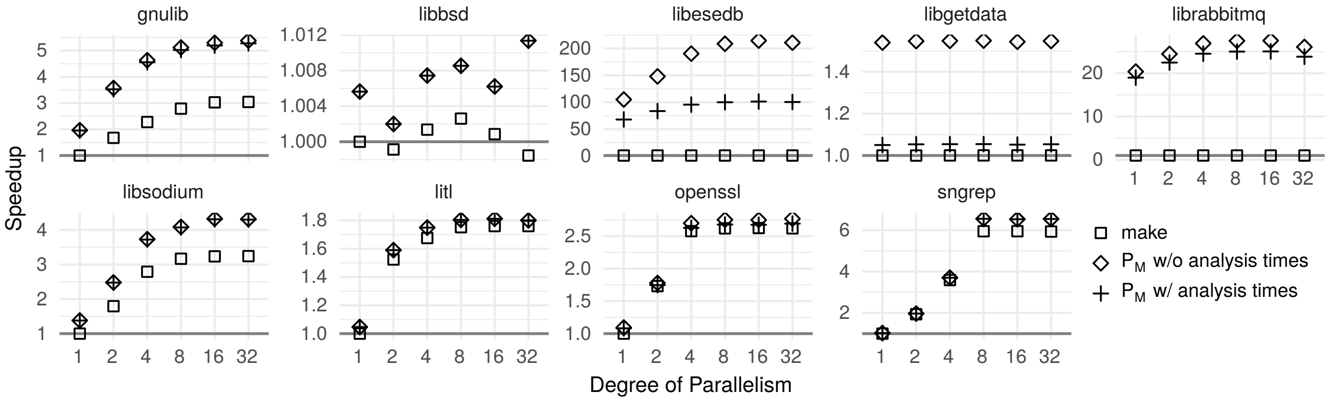


Figure 6: Parallel make and P_M speedups relative to sequential make-based execution. For P_M with analysis times, the file dependency analysis runtime was added to the test execution time.

We find globals dependencies in only two test suites: gnulib and libgetdata. In both cases, several tests declare their own versions of global variables using the same names, which induces potential conflicts when we link several tests together. We also observe conflicts in gnulib and libbsd resulting from our blacklisting mechanism. In particular, both gnulib and libbsd have tests that make assumptions about the absolute number of file descriptors, and gnulib has several tests that call functions which alter the execution environment in a manner that affects other threads in the same process (e.g., calling setrlimit or changing the working directory). Our globals dependency analysis and blacklisting mechanism allow us to parallelize these test suites despite such issues. Since most globals dependencies originate in the projects themselves, test suite modifications are insufficient to remove them.

File dependencies occur in few projects and exclusively originate in test code, leading to flaky behavior when not accounted for in parallel execution. Globals dependencies are more common and frequently originate in the project itself.

5.4 RQ 3: Achieved Speed-Ups

To assess the achievable speedups from concurrent test executions, we analyze how test suite execution times develop with increasing degrees of parallelism across the different execution modes.

As our Debian repository study shows that many projects benefit from parallel make-based execution, we first analyze execution times obtained with make as our baseline. Figure 6 shows the observed speedups (y-axis, different scales) compared to *sequential* make (cf. Table 1) for each project across increasing parallelism degrees (x-axis, □). We observe that 3 projects (libbsd, libgetdata, librabbitmq) show no meaningful speedups with increasing parallelism for make. The other 6 show speedups ranging from 1.02× to 5.95× (sngrep). libgetdata does not benefit from parallel make as sequential execution is hardcoded in its Makefile. Comparing speedups achieved with our P_M mode (◇ in Figure 6), being conceptually closest to make (but respecting file dependencies), to make speedups, shows that P_M consistently outperforms make with speedups over sequential make of 214× for the extreme case of libesedb, having extremely short tests (similarly to librabbitmq). Leaving out these extreme cases, we see speedups ranging from 1.01× to 6.55× (sngrep). The maximum relative speedup between parallel make and P_M was seen for gnulib with 2.13×. Remarkably, even sequential P_M execution is faster than sequential make execution (cf. Table 1), which shows that make imposes a non-negligible overhead, being over 11 s for gnulib and over 18 s for libgetdata.

Comparing P_M to T_M and T_W, we observe that only 3 projects consistently benefit from multi-threaded test execution, which is illustrated in Figure 7 where the achieved speedups over P_M at respective degrees of parallelism for T_M, T_W, and P_W are shown in the upper part (geometric mean and min/max). libesedb and librabbitmq achieve a best case speedup of 1.9× for T_M and 2.9× for T_W, corresponding to less than 7 ms, whereas libgetdata achieves a minor speedup of up to 1.03× for both T_M and T_W, corresponding to 950 ms. We attribute the better multi-threaded performance for libesedb and librabbitmq to their extremely short tests (<1ms) where process creation overhead outweigh actual test execution. Similarly for libgetdata, we see the reason for the better T_M performance in the high number of short tests where over 95 % of tests are shorter than 5 ms. openssl and libbsd, on the other hand, never benefit from T_M or T_W. All but the above 3 projects tend to perform worse in T_M/T_W than in P_M with a mean speedup of 1 or less with the extreme of libsodium with 0.4×.

To underpin our observation that multi-threading is not worthwhile compared to P_M, we perform a one-sided Wilcoxon signed-rank test with the null hypothesis that there is no execution time difference between P_M and T_M in the median and the alternative hypothesis that the median difference between P_M and T_M is positive. We perform the test for each project separately, pair the data points according to the parallelism degree, and use a significance level of $\alpha = 0.05$. For brevity, we omit the exact statistics and p-values; however, we were only able to reject the null hypothesis for the above mentioned 3 projects with p-values < 0.05 that showed geometric means speedups larger than 1. Hence, we cannot find statistically significant evidence that thread-based execution performs better than processes for the majority of studied projects.

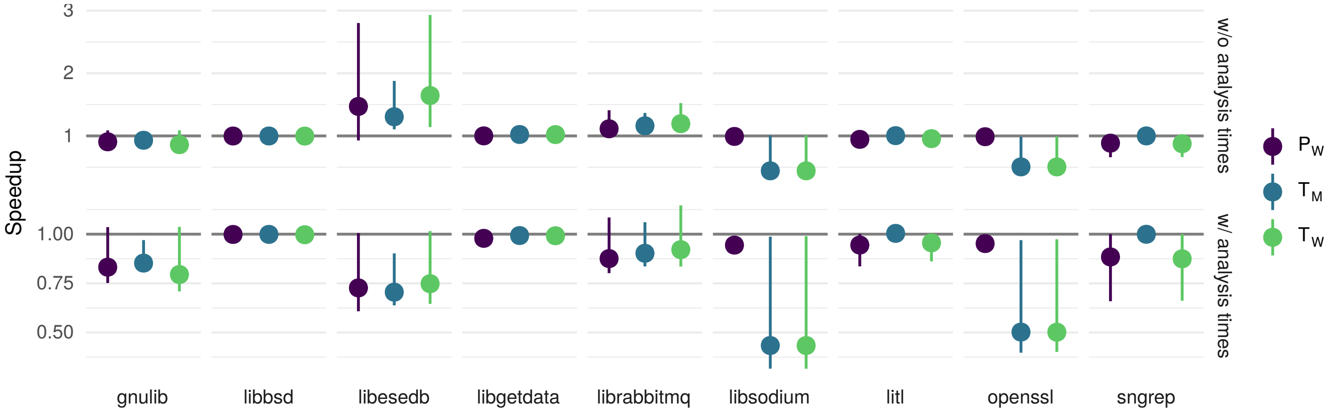


Figure 7: Geometric mean speedups relative to P_M at different degrees of parallelism. Lines indicate minimum/maximum speedups. Speedups without analysis times are based exclusively on execution times, while speedups with analysis times include the file dependency analysis for P_M and both file and globals dependency analysis for the other modes.

Worker-based execution in P_W performs similar to T_W with the exception of libsodium and openssl where P_W, with a geometric mean speedup close to 1, performs better than T_W. However, worker-based execution perform sometimes slightly worse compared to other modes as tests have to be assigned to workers for serial execution without prior knowledge of individual test case durations, which can lead to suboptimal performance if multiple long running tests are assigned to the same worker. This effect can be observed for gnulib, litl, and sngrep where worker-based modes show slightly lower geometric mean speedups.

Two of the studied projects, libbsd and libgetdata, have comparatively long test suite execution times (cf. Table 1) without a clear performance benefit of parallel execution. For libbsd a long running test case (arc4random) is the reason. For libgetdata file dependencies between virtually all test cases are the reason. To investigate the performance impact of such implementation decisions, we created variants where the long running test case of libbsd is restructured into 4 C functions that our analysis and test harness can recognize as test cases and the file dependencies in libgetdata have been removed by introducing unique filenames using a simple sed invocation. These very simple changes enable parallel execution in P_M mode with maximum speedups over make of up to 2.34× or 32.2 s for libbsd and 5.3× or 42.4 s for libgetdata.

Using P_M, we achieve parallel speedups of more than 2× over parallel and more than 6× over sequential make. Even sequentially, P_M consistently outperforms make, indicating that a dedicated tool is preferable over make. Multi-threaded parallel execution is advantageous in only few cases with limited benefits.

5.5 RQ 4: Analysis Runtime Overhead and Amortization

To assess the run time overhead of our analyses, we run them on each project and measure the execution times. In the following, we consider the mean values of 30 repeated measurements for each project, which we report in the *Analysis Time* columns of Table 1. Both our analyses finish in less than 1 s in all cases except for libgetdata, where our file analysis needs almost 16 s and our globals analysis 1.1 s to complete. This effect results from the high number of file dependencies and test cases in libgetdata (cf. Table 1). In general, our analysis requires pairwise comparison of read and write sets for all file-accessing test cases, which results in quadratic complexity. Reducing the number of file dependencies, as we did for the modified libgetdata variant discussed in Section 5.4, the file analysis time is reduced considerably by 16×.

To put the analysis run times into perspective, we relate them to the parallel test execution speedups that we achieve over make. We add required analysis times for each project to the test execution time for our approach. As shown in Figure 6, when adding the file dependency analysis time, P_M (x-axis, +) still outperforms make (□) for all projects across all degrees of parallelism. In the extreme case of libesedb the speedup is still up to 101× and for sngrep 6.55× over make. Looking at absolute time savings of P_M compared to make at respective parallelism degrees, we observed the best case for gnulib with 11.5 s saving. For our modified version of libgetdata, we saved up to 41.4 s. Overall, we observed savings between 15 ms and 1500 ms for 7 projects and savings above 2.5 s for the remaining 2 (excluding our two modified variants).

To assess the impact of the globals analysis time on the viability of the three modes that require it (T_M, T_W, P_W), we add file and globals analysis times to the test execution time for these modes and compute the resulting speedup relative to P_M with added file dependency analysis time. As shown in the lower part of Figure 7, this results in a best case speedup of just 1.15 over P_M (librabbitmq in T_M). No project exhibits a mean speedup over 1.0 in any of T_M, T_W or P_W. For libsodium and openssl, using either of the thread-based modes T_M and T_W effectively halves performance when taking the additional analysis time into account.

The observed analysis overheads are low enough to pay off for parallel test execution with processes in all cases. The performance advantages of multi-threaded parallel execution are not sufficient to justify the increased analysis overhead.

5.6 Threats To Validity

Our analyses and conclusions depend on the selection of software projects and may not generalize to other software. We performed all our experiments on one platform (hardware and software), which may bias our results towards that single platform. We use platform supplied means for our time measurements and depend on their precision and accuracy. We have done our best to limit the influence of defective implementations and include both the raw data we collected as well as our execution and analysis scripts in our artifact.

6 DISCUSSION & LESSONS LEARNED

As we observed in our experiments, relying on make for test suite execution requires longer sequential execution times and achieves lower parallel speedups compared to our test harness. libesedb is an extreme example for this effect where make requires 2 orders of magnitude more execution time than P_M. make's overhead can be saved by using tools that are tailored to test suite orchestration rather than a generic build automation tool like make. Hence, we recommend using specialized tools for test suite management. Such specialized tools should support the parallel execution of tests, as we observe parallel speedups with P_M in 7 out of 9 cases.

The observed performance of the multi-thread parallelization strategies was similar to the multi-process strategies. We expected to see both larger and more consistent differences in the execution times for P_M and T_M as both strategies spawn a new execution entity for each test, but thread creation is commonly considered a lighter operation than process forking. The 3 cases where we could observe a consistent performance advantage of multi-threading were those (1) that had very short test run times where the creation/cleanup of the execution entity dominates the overall execution time or (2) where a highly sequential execution was enforced in all modes (e.g., due to file conflicts) and the speedups achieved through parallelism could not compensate for the creation overhead of execution entities. As the analysis overhead required for multi-threaded execution eats up the small time savings these modes offer, we recommend P_M as the default choice for parallelization. The same considerations apply for the execution with a worker model (P_W, T_W) as we could not observe a clear performance benefit esp. when analysis overhead is taken into account.

For choosing a suitable parallel execution mode, the nature of the tests must be considered. Tests that persistently change their process environment without cleanup, e.g., changing working directories or changing environment variables, cannot safely execute in the same process, not even sequentially. As tests are often designed with the implicit assumption that they execute in their own process, cleanup code is commonly omitted. Such tests are inherently unsuited for multi-threaded or worker-based execution and they need to be removed for modes other than P_M or cleanup code needs to be added, which would increase the required manual effort. An extreme case, for which a cleanup is usually not possible, are tests that destroy their process, e.g., by explicitly aborting process execution, sending process signals, or causing segmentation faults. We opted to exclude such tests in our evaluation which is the reason for the reduction of test cases we report.

The achievable execution speedups depend on the parallelization potential of the test suite. The more test cases there are, the fewer dependencies they have, and the more similar the individual test case execution times are, the higher are the achievable speedups. Ideally, test suites would be designed with these goals in mind. However, our study of the Debian repository and our evaluation indicate that only a fraction of C projects ship with test suites that already benefit from parallel execution. Hence, a migration path to parallel test suites is desirable to tap into the full potential of modern hardware for testing. Our approach offers such a migration path as we demonstrated in our evaluation that existing test suites can be converted with an acceptable one-time effort. This way, projects can benefit from parallel test execution going forward. We furthermore demonstrated (for libgetdata) that by mechanically removing file dependencies identified by our analysis, the achievable speedups can be increased considerably. The locations of conflicting globals and files we found suggest that existing test suites have further parallelization potential as a non-negligible number of dependencies originate in test code (cf. Table 1), which could be easily removed by test suite restructuring.

The execution time savings we observed range from the order of tens of milliseconds to tens of seconds. These seem to be moderate savings in absolute numbers. However, we expect substantial savings in scenarios that require numerous test suite executions such as in mutation testing, when scaling to larger test suites, or when conducting analyses on the ecosystem scale. For instance, for conducting our experiments with gnulib in this paper, we executed its test suite 30 times for each of the 6 degrees of parallelism. The total execution time for these experiments was about 35 min when executed with make, and only about 19 min when executed with P_M, which is almost a reduction by half.

7 CONCLUSION

In our study of the Debian "Buster" software repository, we found that C is the predominant language (28.6 % of total SLOC) and that only a fraction of C projects benefit from trivial parallel test execution using make. We showed that our approach of static dependency analysis with multi-thread and multi-process execution strategies is applicable to real world software in a study of nine software projects. We identified file dependencies in three and globals dependencies in five projects. All file dependencies originated in test code but most globals dependencies originated in the project code itself, suggesting that file dependencies can be removed by test suite modifications whereas globals dependencies cannot. Moreover, we can efficiently execute tests in parallel, even in the presence of such dependencies using our static analyses and test harness. We achieved test execution speedups over make of up to 210× in extreme cases and 2.1× in other cases with our multi-process strategy P_M. P_M outperforms make even in the the sequential case, indicating that the use of a dedicated test orchestration tool is preferable over make. Multi-thread strategies did not show a consistent performance benefit for most projects we studied and offer no advantage when accounting for analysis time.

ACKNOWLEDGMENTS

Research supported in part by H2020-SU-ICT-2018-2 CONCORDIA GA #830927 and BMBF-Hessen TUD CRISP.

REFERENCES

[1] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 550–561. https://doi.org/10.1145/2568225.2568248

[2] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM Press, 770–781. https://doi.org/10.1145/2786805.2786823

[3] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test Suite Parallelization in Open-source Projects: A Study on Its Usage and Impact. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 838–848. http://dl.acm.org/citation.cfm?id=3155562.3155667

[4] Al Danial. 2018. cloc. https://github.com/AlDanial/cloc

[5] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. 2006. GridUnit: Software Testing on the Grid. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, 779–782. https://doi.org/10.1145/1134285.1134410

[6] A. Gambi, J. Bell, and A. Zeller. 2018. Practical Test Dependency Detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 1–11. https://doi.org/10.1109/ICST.2018.00011

[7] A. Gambi, A. Gorla, and A. Zeller. 2017. O!Snap: Cost-Efficient Testing in the Cloud. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 454–459. https://doi.org/10.1109/ICST.2017.51

[8] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. 2017. CUT: Automatic Unit Testing in the Cloud. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, 364–367. https://doi.org/10.1145/3092703.3098222

[9] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. 2017. An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, 495–498. https://doi.org/10.1109/MSR.2017.38

[10] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 223–233. https://doi.org/10.1145/2771783.2771793

[11] INRIA. 2018. Coccinelle Website. coccinelle.lip6.fr

[12] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. 2013. An Empirical Study of Adoption of Software Testing in Open Source Projects. In *2013 13th International Conference on Quality Software*. 103–112. https://doi.org/10.1109/QSIC.2013.57

[13] Wing Lam, Sai Zhang, and Michael D. Ernst. 2015. *When Tests Collide: Evaluating and Coping with the Impact of Test Dependence*. Technical Report. University of Washington Department of Computer Science and Engineering.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, 75–87. http://dl.acm.org/citation.cfm?id=977395.977673

[15] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel Test Generation and Execution with Korat. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, 135–144. https://doi.org/10.1145/1287624.1287645

[16] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding Bugs by Isolating Unit Tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 496–499. https://doi.org/10.1145/2025113.2025202

[17] M. Oriol and F. Ullah. 2010. YETI on the Cloud. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 434–437. https://doi.org/10.1109/ICSTW.2010.68

[18] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*. ACM, 247–260. https://doi.org/10.1145/1352592.1352618

[19] T. Parveen, S. Tilley, N. Daley, and P. Morales. 2009. Towards a Distributed Execution Framework for JUnit Test Cases. In *2009 IEEE International Conference on Software Maintenance*. 425–428. https://doi.org/10.1109/ICSM.2009.5306292

[20] Debian Software Repository. 2017. Debian Buster Source Package Index [retrieved 2017-07-18]. http://ftp.debian.org/debian/dists/buster/main/source/Sources.gz

[21] Debian Wiki Team. 2018. Debian Derivatives Census. https://wiki.debian.org/Derivatives/Census

[22] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM Press, 385–396. https://doi.org/10.1145/2610384.2610404