Panda: A Concurrent Scheduler for Compiler-Based Tools

Xutong Ma
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
Beijing, China

Jiwei Yan
Tech. Center of Software Eng., Ins. of Software, CAS
Beijing, China
yanjiwei@otcaix.iscas.ac.cn

Jun Yan*
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
University of Chinese Academy of Sciences
Beijing, China

Jian Zhang*
Key Lab. of System Software (CAS) and State Key Lab. of Computer Science, Ins. of Software, CAS
University of Chinese Academy of Sciences
Beijing, China

Abstract

The widely-used *Compiler-Based Tools* (CBT), such as static analyzers, process input source code using data structures inside a compiler. CBTs can be invoked together with compilers by injecting the compilation process. However, it is seldom the best practice for the inconvenience of running various CBTs, the unexpected failures due to interference with compilers, and the efficiency degradation under compilation dependencies. To fill this gap, we propose *Panda*, an efficient scheduler for C/C++ CBTs. It executes various CBTs in a compilation-independent manner to avoid mutual interference with the build system, and parallelizes the process based on an estimated makespan to improve the execution efficiency. The assessment indicates that *Panda* can reduce the total execution time by 19%–47% compared with compilation-coupled execution, with an average 39.03×–52.15× speedup with 64 parallel workers.

CCS Concepts

• **Software and its engineering → Development frameworks and environments**.

Keywords

Compiler Argument, Static Analysis, Generic Concurrent Scheduler

ACM Reference Format:
Xutong Ma, Jiwei Yan, Jun Yan, and Jian Zhang. 2024. Panda: A Concurrent Scheduler for Compiler-Based Tools. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages.
https://doi.org/10.1145/3650212.3685311

1 Introduction

Compilers collect rich information during the compilation process, which is also helpful to manufacturing code utilities, such as program static analyzers, code transformation tools, and so on. Some of them are *Integrated Tools* (IT), which are provided as a part of a full-volume compiler; while others are *Singleton Tools* (ST) that contain only necessary components to collect desired information. To make their dependent compiler components properly configured, the command line arguments launching a compiler are fundamental to executing *Compiler-Based Tools* (CBT).

To feed CBTs with compiler arguments, two approaches have been proposed for ITs and STs respectively. ITs are usually executed by directly overriding the compiler to be invoked, such as CpyChecker [8]; whereas STs will utilize their specialized drivers to capture and convert compiler arguments into tool arguments to launch them, such as the *Scan-Build* scheduler of Clang-SA [14]. For both approaches, the build system concurrently schedules the execution of CBTs during the project compilation. However, the approaches still face the three challenges below.

First, **customized the execution of different CBTs**. Tool users usually use several different ITs and STs together, also with customized configurations [15]. For ITs, multiple tools cannot be executed by overriding the compiler, as the build system can only schedule one exact kind of compiler. And customizing tool usages by adjusting the command line arguments can seldom be automatically carried out. For STs, their specialized drivers cannot run other tools. And advanced functionalities, such as unstable features under alpha tests, may be unavailable from their drivers. Hence, it is difficult for users to flexibly use CBTs.

Second, **interference with the build system**. Running a CBT during compilation may introduce undesired failures, and the mixed output of both CBT and compiler will make it difficult to collect. For instance, CpyChecker reports bugs through compiler warnings, which are mixed with compilation warnings. And when the compiler arguments contain `-Werror`, which converts a warning to an error, bug reports generated by the analyzer will be considered compilation failures and hence interrupt the build process. As different tools have various ways of generating output, it is difficult to avoid the interference when executing CBTs together with compilers.

Third, **efficiency degradation due to compilation dependencies**. In the build system, dependees should be built before the depender. However, in most cases when executing a CBT, files are always independent of each other. Running a CBT on one file will not depend on the outputs of the executions on other files. When executing CBTs together with the compiler during compilation, these unnecessary pauses scheduled by the build system based

*Corresponding authors. Emails: yanjun@ios.ac.cn and zj@ios.ac.cn

This work is licensed under a Creative Commons Attribution 4.0 International License.

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685311

on compilation dependencies will greatly degrade the efficiency of concurrent execution of CBTs.

To fill the gap, we design *Panda* as a generic scheduler for concurrently executing common C/C++ CBTs to respond to these challenges. (1) *Panda* is designed as a compilation-independent process to avoid mutual interference with the build system. This also makes it able to isolate the execution of various CBTs. (2) It can customize the execution of various CBTs according to a corresponding configuration by automatically modifying the compiler arguments. (3) It schedules CBT executions according to a lightweight online estimation of makespan for better performance.

2 The Panda Tool

Figure 1 presents the system structure and workflow of *Panda*, whose corresponding intermediate representations of processing the record in Figure 2 are presented in Figure 3 and 5.

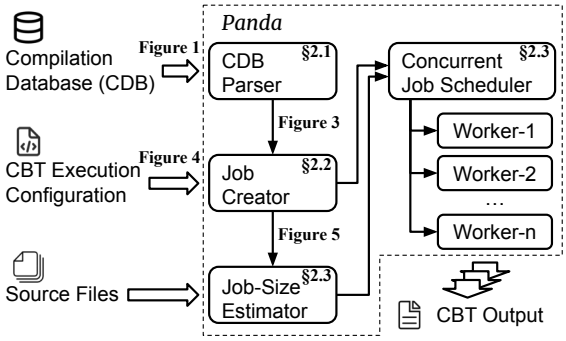


Figure 1: System Structure and workflow of *Panda*

Panda takes a pre-captured *Compilation Database* as input, extracts the information essential to execute a CBT independently to the build system (§2.1, *challenge 2*), constructs the command line arguments for CBT executions based on a *CBT Execution Configuration* to customize the execution of different kinds of CBTs (§2.2, *challenge 1*), and concurrently schedules the CBT executions in a dependency-free manner to gain further speedup compared with compilation-coupled execution (§2.3, *challenge 3*).

2.1 Parsing Compilation Database

To independently execute CBTs, it is essential to know how the compiler is originally invoked to replay the compilation process in a CBT. Such information is available in a *Compilation Database* (CDB) [12]. It is a JSON-formatted text file storing the command line arguments and the working directory of invoking a compiler on each *Translation Unit* (TU), *i.e.* one main file (`.cpp`) and all header files (`.h`) it includes. Figure 2 presents a record in a CDB, which is

```
1 [..., {
2   "command": "clang -x c++ -c temp.c -o temp.o -g -MD
  -MF temp.d -Werror -Wall -DVERSION=\"version 1.0\"
  -I/path/to/dependency",
3   "directory": "/path/to/project",
4   "file": "temp.c"
5 }, ...]
```

Figure 2: A compilation database with an example record.

- **arguments**: `[-x, c++, temp.c, -DVERSION="version 1.0", -I/path/to/dependency]`
- **directory**: /path/to/project
- **file**: /path/to/project/temp.c
- **language**: C++

Figure 3: A CDB record generated for the example in Figure 2

composed of a **`file`** field providing the path to the main file of the TU, a **`directory`** field recording in which directory the main file is compiled, and a **`command`** field storing the compiler arguments. The **CDB Parser** will read the JSON text of an input CDB to load the records in it and produce a CDB record as presented in Figure 3.

To unify the representation of the information in an input CDB generated by various kinds of producers, the content in the original JSON object will be adjusted. The path to the main file in field **`file`** will be converted to an absolute path according to field **`directory`**. And the compiler argument string in field **`command`** will be split into a list of strings storing each argument separately.

Then the string list of compiler arguments will be parsed to filter out unnecessary ones and update information in the output CDB record accordingly. The *language* type will be updated if it is specified explicitly with argument `-x`. And the original compiler as well as the arguments about compiler actions (such as `-c`, `-save-temps`, and so on), output (`-o`), debug (`-g` options), diagnostics (`-W` options), and dependency (`-M` options) will be pruned from the *arguments* list as they are unrelated to the compilation process.

2.2 Creating Description for Jobs

The **Job Creator** generates the *job descriptions* describing how to execute a CBT on a TU, which will be scheduled concurrently in later steps. To make the execution customizable to users, the *CBT execution configurations* are introduced to guide the process under the intent of users.

As mentioned in §1, we separate the common C/C++ CBTs into two categories: (1) *Integrated Tools*: they are invoked in the same way as compilers, and (2) *Singleton Tools*: they accept compiler arguments after their specific options. Hence, both *descriptions* and *configurations* have two corresponding formats (as shown in Figure 4).

In a configuration, *type* determines how to organize the command line arguments of invoking the CBT; *prompt* is the output message when executing the job; *tool* and *arguments* customize the CBT and its specific command line arguments to be executed, *source* indicates the output is generated by the CBT (file), or collected from standard streams (stdout or stderr); and *extension* denotes the extension name of the output file to be generated. And in a generated description (Figure 5), *arguments* and *directory* present how and where to invoke the CBT; *output* shows the path to the output file; and *source* and *prompt* are the same as in the input configuration.

When creating *descriptions* from a *configuration* for an IT (in Figure 3 $\xrightarrow{4a}$ 5*a*), the CBT to be executed is determined according to the language type. Then we add the arguments from the CDB, the input *configuration*, and for setting output paths. Similarly, for the *configuration* for an ST (in Figure 3 $\xrightarrow{4b}$ 5*b*), its *arguments* are composed of the *tool*, the main file of the TU, arguments from the *configuration*, a delimiter (`--`), and the compiler arguments.

• **type**:	Integrated Tool
• **prompt**:	`"Generating LLVM-IR code"`
• **tool**:	{C: `clang`, C++: `clang++`}
• **arguments**:	[`-c`, `-emit-llvm`, `-S`]
• **source**:	file
• **extension**:	`".ll"`

(a) Generating LLVM-IR code dump by invoking the Clang compiler

• **type**:	Singleton Tool
• **prompt**:	`"Matching goto statement"`
• **tool**:	clang-query
• **arguments**:	[`-c`, `"match gotoStmt()"`]
• **source**:	stdout
• **extension**:	`".clang-query"`

(b) Identifying all `goto` statements with code audit tool Clang-Query

Figure 4: Example *CBT Execution Configurations* for generating the *Job Descriptions* shown in Figure 5a and 5b

• **arguments**:	[`clang++`, `-x`, `c++`, `temp.c`, `-DVERSION="version 1.0"`, `-I/path/to/dependency`, `-c`, `-emit-llvm`, `-S`, `-w`, `-o`, `/path/to/output/path/to/project/temp.c.ll`]
• **directory**:	/path/to/project
• **output**:	/path/to/output/path/to/project/temp.c.ll
• **source**:	file
• **prompt**:	`"Generating LLVM-IR code"`

(a) Job description of Integrated Tools generated from Figure 4a

• **arguments**:	[`clang-query`, `temp.c`, `-c`, `"match gotoStmt()"`, `--`, `-x`, `c++`, `temp.c`, `-DVERSION="version 1.0"`, `-I/path/to/dependency`]
• **directory**:	/path/to/project
• **output**:	/path/to/output/path/to/project/temp.c.clang-query
• **source**:	stdout
• **prompt**:	`"Matching goto statement"`

(b) Job description of Singleton Tools generated from Figure 4b

Figure 5: Job descriptions generated from corresponding CBT execution configurations in Figure 4

Finally, each **Worker** in the **Concurrent Job Scheduler** will fork the process to launch the CBT. The output of the standard streams will be stored in the output file if specified in the *description*.

2.3 Concurrently Scheduling Job Execution

Since we assume that there are no precedence orders among TUs when invoking CBTs, the **Concurrent Job Scheduler** can be modeled with the *Identical-Machines Scheduling* problem whose target is minimizing the maximum completion time ($P||C_{max}$). The problem is NP-hard, and solving it is a significant overhead that an online scheduler cannot afford. A widely used approximate solution is a list-scheduling algorithm called *Longest Processing Time First*, which prioritizes the execution of jobs with longer makespans. When scheduling under a concurrency of m, the total makespan is $\frac{4}{3} - \frac{1}{3m}$ times of the optimal schedule in the worst case [4].

Hence, the key to a better execution sequence is to estimate the makespan of each job, *i.e.* to estimate how long a tool will execute with a given TU. Besides, as an online scheduler that estimates the makespan and schedules the jobs at the same time, we need to keep the overhead of the scheduler as small as possible.

To achieve this, the **Job Size Estimator** needs to set a value to each job description as the key to sorting the schedule list. Since a smaller overhead is preferred, to select a better key, we have measured the makespan of running the Clang-SA on every TU in project LLVM. And multiple approaches suggest that token semicolon (`;`) has higher importance and correlation to the makespan. Hence, we use the number of semicolons to sort the TUs.

The **Job Size Estimator** wraps every job description with the number of appearances of semicolons in the main file of the TU. And the **Concurrent Job Scheduler** dispatches the descriptions to a process pool of **Workers** with a priority queue.

2.4 Usage of *Panda*

Panda is provided as a command-line tool. In this subsection, we will introduce the functionalities of *Panda* with its options.

• ***Customizing CBT Executions.*** As mentioned in §2.2, users can customize the execution of CBTs. This can be achieved by defining an execution configuration, which overcomes the first challenge. For example, the execution configuration presented in Figure 4b can be defined with a plugin in JSON format in Figure 6. Besides, *Panda* also has built-in configurations for generating compiler intermediate representations.

```
1  { "type": "Singleton",
2    "action": { "prompt": "Match goto statement",
3                "tool": "clang-query",
4                "args": ["-c", "match gotoStmt()"],
5                "extension": ".clang-query",
6                "source": "stdout" } }
```

Figure 6: Plugin version for the configuration in Figure 4b

• ***Controlling Concurrent Execution.*** As a concurrent scheduler, users can also customize the number of workers executing CBTs in parallel, the strategy of the job scheduler (among the First-Come-First-Service (FCFS) and the Longest-Processing-Time-First (LPTF) strategies), and the feature of measuring job size (among semicolon and LoC). By default, LPTF and semicolon are used. According to our experimental results in Figure 8, we suggest setting the worker number to 10%–20% of the TU number and using the FCFS strategy with at least 2–4 parallel workers on small projects for smaller scheduler overhead. Figure 7 shows the duty (green) and idle (red) durations of four parallel workers when executing the configuration in Figure 4a on project *Bftpd*.

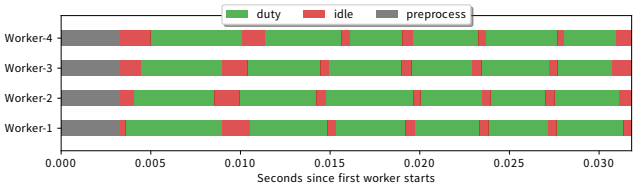


Figure 7: *Panda*'s schedule of parallel workers

• ***Controlling Input/Output.*** *Panda* allows users to customize the range of TUs to be processed. This makes it possible to flexibly do a partial or incremental analysis on an updated project. And

users can also determine the directory for storing the output files. This makes it easier to collect CBT outputs and hence can avoid interference with the source code and the build system.

3 Evaluation

We assess *Panda* to answer the following two research questions.

- **RQ 1**: How much time can be saved by executing CBTs independently with the build system?
- **RQ 2**: How efficient is our concurrent scheduler with the increment of number of parallel workers?

The first research question evaluates the efficiency improvements from compilation dependency and responds to the third challenge. And the second one assesses the effectiveness of the Job-Size Estimator and the efficiency of the Concurrent Job Scheduler.

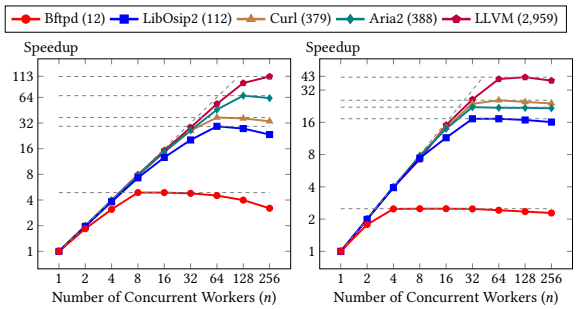
The benchmark is composed of five popular C/C++ projects of different sizes. The first two columns of Table 1 show their names and sizes (measured with TU numbers). The experiments are executed on a Linux server with Intel® Xeon® Platinum 8153 CPU. All data are measured with the mean value of five runs.

To answer the first research question, we measure the total time consumption of compilation and CBT execution against *Scan-Build*, the specialized command line argument converter of Clang-SA [10]. The compilation of all projects in the benchmark are scheduled with GNU Make. Table 1 shows the total time consumption of these two schedulers, where *Scan-Build* fails to execute the Clang-SA on project LLVM. The third column shows the number of parallel workers (#PW), which are determined based on the suggestions in §2.4. The fourth column ($T_{Coupled}$) denotes the total time consumption of compilation and CBT execution under compilation dependencies. And the last two columns represent the separated time consumption of compilation ($T_{Compile}$) and CBT execution (T_{Panda}, scheduled by *Panda*). As we can see from the table, due to the dependency-induced pause during compilation, the time consumption of *Scan-Build* is 19%–47% higher than *Panda*.

Table 1: Time consumption of *Scan-Build* and *Panda*

Project	#TU	#PW	$T_{Coupled}$	$T_{Compile}$ + T_{Panda}
Bftpd	12	4	10.19	0.61 + 7.64
LibOsip2	112	16	23.48	3.81 + 11.59
Curl	379	32	39.80	5.93 + 15.32
Aria2	388	32	72.98	25.28 + 34.03
LLVM	2,959	256	—	381.00 + 799.83

Besides, to answer the second research question, we measure the speedup against sequential execution with different concurrency, as shown in Figure 8. Each curve represents a benchmark instance, whose TU numbers are shown in the parenthesis. In this experiment, we use *Panda* to schedule the syntax checker of GCC and the Clang-SA, which respectively represent light-weight CBTs and heavy-weight CBTs that have more onerous computations or iterations. The average speedup among all projects with 64 parallel works is 52.15× for GCC and 39.03× for Clang-SA. The main performance bottleneck lies in the size of the project. It makes *Panda* reach the performance upper-bound when scheduling with more parallel workers than the number of TUs (*e.g.* Bftpd).



(a) GCC Syntax Check (b) Clang-SA

Figure 8: Speedup of different concurrency configurations

4 Related Work

Scheduling tool executions concurrently has been implemented with multiple approaches. As mentioned in §1, tools can be executed directly by overriding the compiler executed during compilation. Analyzer *CpyChecker* [8] and driver *blight* [9] achieve this by using environment variables. Whereas *Scan-Build* [14] directly injects the build system and replaces the compiler with its compiler wrapper. However, due to the interference with the build system, this approach usually leads to a failed build.

Besides, tool execution can also be scheduled with its specifically designed driver. *CodeChecker* [3] is an integrated system for executing the Clang-SA and presenting the bug reports. The `run-tool` script in Chromium is a driver designed for their internally-used tools [13]. In addition, the Clang-Tidy [11] and Infer Analyzer [2] also provide their drivers to execute the tool under a pre-extracted CDB. However, all of them are designed and optimized for specific analyzers with the First-Come-First-Service strategy only.

In the literature, *Panda* has already been used to schedule the execution of *PyRefcon* [7]. For other tools similar to *PyRefcon* [5, 6, 17], they can also be adapted to be scheduled with *Panda*. Besides, it can also be used to generate inputs (such as LLVM-IR and Preprocessed source code) for other static analyzers [1, 16].

5 Conclusion and Future Work

In this paper, we propose a compilation-independent concurrent scheduler for executing multiple Compiler-Based Tools according to the records in a Compilation Database, which can avoid interference with the build system and efficiency degradation due to compilation dependencies. In the future, we will continue adding more strategies for more accurate job size estimation and higher efficiency.

Tool Availability

For the archived version of *Panda* for the ISSTA 2024 conference, please visit the `demo` branch of its GitHub repository via https://github.com/Snape3058/panda/tree/demo. A demo video introducing its usage can be found at https://youtu.be/YQTg5LsId5k.

Acknowledgments

This research is supported by the National Natural Science Foundation of China (NSFC) under grant number 62132020 and Major Project of ISCAS (ISCAS-ZD-202302).

References

[1] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224. https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf

[2] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review* 35, 5 (2001), 57–72. https://doi.org/10.1145/502034.502041

[3] Ericsson. online. CodeChecker. https://github.com/Ericsson/CodeChecker

[4] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429. https://doi.org/10.1137/0117039

[5] Xutong Ma, Jiwei Yan, Yaqi Li, Jun Yan, and Jian Zhang. 2019. SPrinter: a static checker for finding smart pointer errors in C++ programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1122–1125. https://doi.org/10.1109/ASE.2019.00117

[6] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, and Zongyan Qiu. 2021. Detecting memory-related bugs by tracking heap memory management of C++ smart pointers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 880–891. https://doi.org/10.1109/ASE51524.2021.9678836

[7] Xutong Ma, Jiwei Yan, Hao Zhang, Jun Yan, and Jian Zhang. 2023. Detecting Memory Errors in Python Native Code by Tracking Object Lifecycle with Reference Count. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1429–1440. https://doi.org/10.1109/ASE56229.2023.00198

[8] David Malcolm. online. GCC Python Plugin. https://gcc-python-plugin.readthedocs.io/en/latest/index.html

[9] Trail of Bits. online. blight: A framework for instrumenting build tools. https://github.com/trailofbits/blight

[10] Clang Team. online. Clang Static Analyzer. https://clang-analyzer.llvm.org

[11] Clang Team. online. Clang-Tidy. http://clang.llvm.org/extra/clang-tidy/

[12] Clang Team. online. JSON Compilation Database Format Specification. https://clang.llvm.org/docs/JSONCompilationDatabase.html

[13] Chromium Team. online. run_tool.py - Chromium. https://chromium.googlesource.com/chromium/src/tools/clang/+/refs/heads/main/scripts/run_tool.py

[14] Clang Team. online. scan-build: running the analyzer from the command line. https://clang-analyzer.llvm.org/scan-build.html

[15] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25, 2 (2020), 1419–1457. https://doi.org/10.1007/S10664-019-09750-5

[16] Zhenbo Xu, Jian Zhang, Zhongxing Xu, and Jiteng Wang. 2014. Canalyze: a static bug-finding tool for C programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 425–428. https://doi.org/10.1145/2610384.2628050

[17] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. 2023. Detecting Exception Handling Bugs in C++ Programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1084–1095.

Received 2024-07-05; accepted 2024-07-26