Concurrency Analysis of Build Systems

Vasil Tenev, Bo Zhang, Martin Becker
Fraunhofer Institute for Experimental Software Engineering (IESE)
Kaiserslautern, Germany
Email: {vasil.tenev, bo.zhang, martin.becker}@iese.fraunhofer.de

***Abstract*—In order to derive executable software artefacts, a build system needs to be maintained properly along with the evolution of source code. However, in large software projects the build process often becomes effort-consuming, which is often caused by suboptimal concurrency either in the design of the build system or in the execution of the build process. To cope with these challenges, we present our concurrency analysis with practical experiences in this paper. In particular, we propose a new metric called Degree of Freedom for evaluating the concurrency potential of a build system based on dependencies among build jobs and artefacts. In fact, this metric is not limited to build analysis. It can be used for analyzing the concurrency potential of any executable process in general.**

***Index Terms*—concurrency, build system, control flow**

I. Introduction

While normal source code (also as known as production source code) implement the behavior of a software, its build system (including build tools and build code such as makefiles) derives the executable software from its production source code. In large industrial software systems, the complexity of the build system is often high (in terms of build jobs and build dependencies), and the build process is time-consuming (over one hour in large systems) even in a distributed environment using high-performance and multi-core computers. This is not acceptable in real continuous integration settings with frequent code revisions and builds per day.

In order to understand the build process and related issues, in our previous work [18] we have depicted the build process via the notion of Build Dependency Structure. Theoretically, the build dependency structure contains two dimensions as illustrated in Figure 1 on page 1. On the one hand, the root build command triggers a flow of build actions that can further run atomic build jobs (e. g., compiling and linking). These build actions and jobs are invoked and executed in a tree structure (vertical in Figure 1 on page 1). On the other hand, the build jobs with different build tools indicate dependencies between input build artefacts and output build artefacts, which further constitute a dependency graph (horizontal in Figure 1 on page 1).

In the build dependency structure, some build jobs need to be executed sequentially due to the dependencies among build jobs and artefacts. However, independent build jobs could be executed in parallel, which helps improving the build efficiency. Therefore, in practice it is important to analyze the build system and make sure that build jobs are executed with optimal concurrency. Although there was some endeavor in build dependency extraction and optimization [6], [12] a

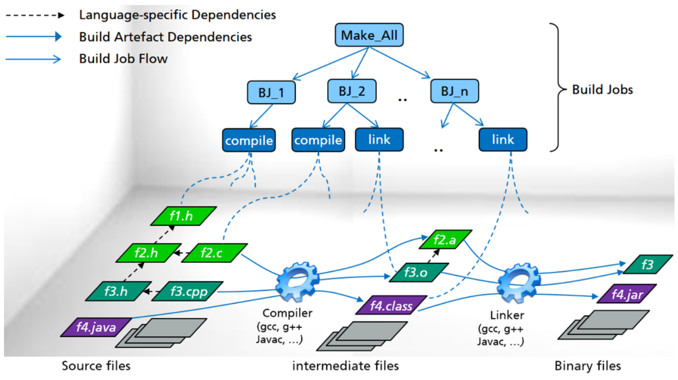


Figure 1: Notion of the Build Dependency Structure

comprehensive analysis focusing on build concurrency is still lacking.

In order to optimize the build concurrency and improve build efficiency, we would like to conduct automated analyses both on the build execution process and also on the build system architecture. At build execution time, we use existing commercial tools for monitoring the build process and analyzing the execution of build jobs in different threads and on different machines. This shows whether build jobs are actually executed in parallel. Moreover, we also conduct static analysis on build dependencies of the build system and measure concurrency potential of the build architecture. This helps identify root causes of concurrency problems in the build process.

In this paper, we provide the following contributions:

- Practices of dynamic and static concurrency analysis in an industrial study.
- An innovative metric called Degree of Freedom for static concurrency analysis.
- Lessons learned during the concurrency analysis and optimization.

This paper is presented in the following structure. Section II presents our practice of dynamic concurrency analysis. Section III introduces our static concurrency analysis approach. While Section Section IV discusses related work, Section V presents conclusions, summarizes lessons learned during our concurrency analysis study, and discusses future work at the end.

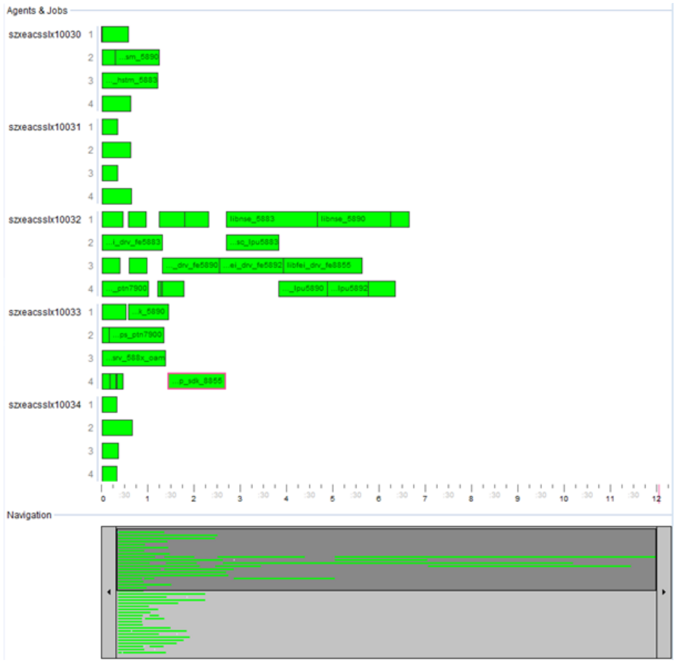


Figure 2: Dynamic Concurrency Analysis by ElectricInsight

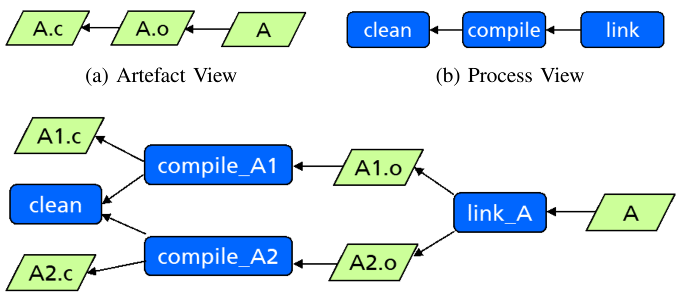
II. Dynamic Concurrency Analysis

The dynamic concurrency analysis shows the actual execution of different build jobs in a distributed environment. There exists commercial tools like ElectricInsight [2] for such purpose. Typically, these tools monitor the build process by instrumenting the GNU (GNU is a recursive acronym for "GNU's Not Unix!") make tool or even replacing it with their own make tool. As a result, a build execution graph is generated to visualize concurrent scheduling and execution of build jobs. For instance, in an industrial case study we used ElectricInsight and generated the build execution graph as shown in Figure 2 on page 2. While some build jobs are executed in parallel in different threads (by build agents) and CPU (Central processing unit) cores, other build jobs are executed sequentially in the same thread.

Besides monitoring build execution, ElectricInsight also claims to optimize the scheduling of concurrent build execution (to reduce the overall build duration). However, from our experience the build concurrency is often not optimal. As seen in Figure 2 on page 2, while a few build threads keep executing build jobs sequentially, many other threads finish early and remain idle until the end of the build process. It seems that some build jobs have to be executed sequentially due to build dependencies defined in the build system (e. g., makefiles). In order to investigate the root causes of this suboptimal build concurrency, it is necessary to further analyze build dependencies and identify the actual concurrency potential.

III. Static Concurrency Analysis

While dynamic concurrency analysis shows the actual behavior of build job execution during the build process, static concurrency analysis focuses on the concurrency potential of



(a) Artefact View (b) Process View

(c) Execution View

Figure 3: Different Views of Build Dependencies.

build jobs and artefacts based on their dependencies. In this section, we assume an arbitrary but fixed environment for the execution of the build process, i. e., number of CPU cores, parallel threads, RAM (Random-access memory) size, etc. Moreover, we assume that the build process has only one connected graph component.

A. Build Dependencies

Various dependencies exist between build jobs and artefacts. We consider three views to analyze the different aspects of the build dependencies structure:

The *Artefact View* contains source code artefacts, intermediate and final artefacts of a build process. In Figure 3a on page 2 an artefact "A.o" depends on artefact "A.c".

On the other hand, the *Process View* shows build jobs and the process dependencies in-between, that represent finish-to-finish relationships. For an example of a clean build see Figure 3b on page 2, where the job "link" can finish, only if job "compile" has finished. The job "link" may however start before, in parallel, or after the job "compile". In any case, "link" can't finish before "compile" is finished.

The *Execution View* describes three kind of dependencies (see Figure 3c on page 2):

- Execution Dependency: Job "compile_A2" executes job "clean";
- Input Dependency: Job "compile_A2" depends on input artefact "A2.c"; and
- Output Dependency: Artefact "A2.o" is result of job "compile_A2".

These views depict all build dependencies in the vertical and horizontal dimensions of the Build Dependencies Structure in [18]. Accounting the different views and the number of possibilities for scheduling build jobs make the basis of the static concurrency analysis.

In any of the views on a well-defined build dependency structure, we deal with an acyclic directed graph. Such structure is equivalent to a partial order over the set of artefacts and build jobs, respectively. This partially ordered set is considered by a build tool (like GNU Make [3], Ninja [5], etc.) to compute a schedule over all build jobs and execute them in the right order. The richer the scheduling possibilities with

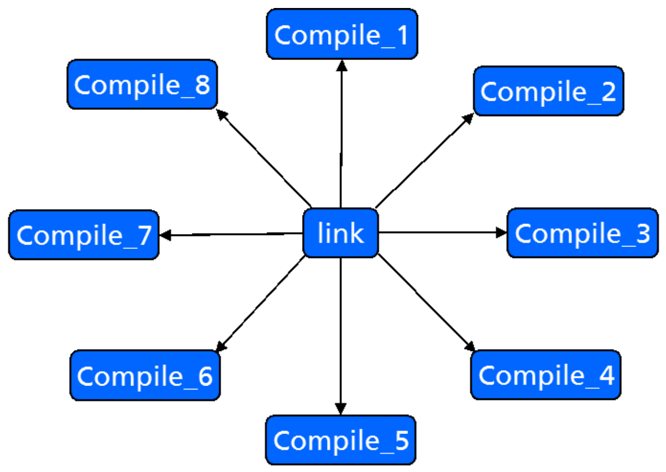


Figure 4: Process view of a best-case example for maximal parallelization

respect to the build dependencies structure, the bigger the optimization space, the greater the degree of freedom for concurrent execution.

B. Degree of Freedom

Here, we propose a new metric called Degree of Freedom that quantifies the parallelization property of a build process, i. e.,the degree to which extent a build process can be parallelized. This stands intuitively in direct correlation with the number of possible scheduling plans.

Examples: (a) Consider a build process where the dependency graph in the process view is a chain (e. g., Figure 3b on page 2). For such a process, there is only one possible plan for scheduling its execution and it is not parallelizable. (b) In comparison, for a star-shaped build process, like in Figure 4 on page 3, there are $8! = 40320$ possible scheduling plans to execute it in one processing thread and 8 of the 9 jobs can be parallelized.

We assume that a build process always has exactly one starting point (i. e., the main build job) and only one final resulting artefact (i. e., the product that is build). Thus, the dependency graph always has exactly one root in every view and therefore a star-shaped build process is the best possible case for maximal parallelization. Using this and the correlation from above, we define a metric that compares a given build process with a same-size, star-shaped build process by the number of possible scheduling plans.

However, the number of scheduling possibilities depends not only on the partial order, but also on the environment, i. e., mostly the number of parallel threads available for the execution. To get an environment independent metric, that is to multiply out the factor depending on the number of threads for parallel computing, we use the number of possible execution sequences for a single thread computation. Therefore, we define the *Degree of Freedom* by

$$\text{Freedom}\,(P) = \log_{S(P^*)} S\,(P)$$

where $S\,(P)$ is the number of possible execution sequences for build process P and $S\,(P^*)$ is the number of possible execution sequences for a star-shaped build process with the same number of build jobs/artefacts as in P. Thus, $S\,(P^*)$ equals$(|P| - 1)!$. Formally, $S\,(P)$ is the number of linear extensions over the partially ordered set (poset) P (see [11]).

Example: Let P be a build process with process view of 9 build jobs, such that P is a sequence. Then the corresponding star-shaped build process P^* is equivalent to this on Figure 4 on page 3. Therefore $\text{Freedom}\,(P) = \log_{S(P^*)} S\,(P) = \log_{8!} 1 = 0$ and $\text{Freedom}\,(P^*) = \log_{S(P^*)} S\,(P^*) = \log_{8!} 8! = 1$.

C. Implementation

At its heart, our metric is based on the number of possible execution sequences for a build process, which is a partial ordered set as discussed in III-A. The number of all sequences with respect to a poset is the number of all linear extensions of the poset [11]. In general, computing $S\,(P)$ is a #P-complete problem for arbitrary posets [8].

Although there are several algorithms that can find one linear extension in linear time [10], it is not clear if there exists even a polynomial time algorithm for computing the number of all linear extensions [16]. However, there are polynomial approximation schemes [9], which can be used to compute the number of all linear extensions by counting. In contrast to these works, we deal with special case of partial ordered sets. From the fact that they correspond to the dependency graph of a build process, we can expect that the posets are ‘almost’ trees. This means, that they contain a small number of edges that are in contradiction with the tree properties. Thus, we based our approach on the algorithm for computing the number of linear extensions in a tree-shaped poset proposed by Atkinson [7]. He proposes an $\mathcal{O}\left(n^2\right)$ algorithm for trees with n being the number of elements. Since we deal with ‘almost’ trees, we develop two algorithms for computing a upper and lower bounds for $S\,(P)$.

1) Upper Bound: We apply Atkinson’s algorithm in combination with Prim’s minimum spanning tree algorithm [15] to approximate a minimum tree-shaped poset that is a superset of P. Firstly, it will take all edges that are conform to the properties of a tree. All remaining edges are evaluated with respect to the minimal number of linear extensions resulting from taking one edge and recursively applying the same procedure until we end up with a tree on which we apply Atkinson’s algorithm. This minimal number defines the edge weight. Afterwards, a minimal spanning tree is constructed. The number of linear extensions of that tree gives a upper bound for the number of linear extensions of P, since the set of dependencies (edges) of the tree is a subset of the dependencies of P. This algorithm runs in $\mathcal{O}\left(k^2 n^2\right)$, where k is the number of edges that contradict to the tree properties from graph theory.

2) Lower Bound: For the approximation of the lower bound, we use a simple scheduling strategy that is more restrictive than an optimal strategy for the given poset. We traverse the poset graph bottom-up starting with leaf nodes — elements with no outgoing dependencies — and moving towards a root node — element without ingoing dependencies. Hereby, we detect independent sequences of nodes between branching locations in the graph. This gives us a sequence of layers in the graph. Each layer consists of independent subsequences. The scheduling strategy is to run all subsequences in parallel and synchronizing the build process where a branching takes place. We compute the number of linear extensions l_j for layer j using the multinomial coefficient. More precisely, for every layer j of m independent subsequences, we have

$$l_j = \binom{k_1 + k_2 + \cdots + k_m}{k_1, k_2, \ldots, k_m}$$

with k_i being the number of nodes in i-th subsequence for all $1 \leq i \leq m$. In total, we need the product of all l_j. Computing the lower bound requires only one traversal of the graph without repetitions, as described above, which implies linear time.

Example: In the following, let P be the build dependency graph from Figure 3c on page 2 to illustrate a computation for both lower and upper bounds. We compute the upper bound, as described earlier, based on Prim's minimum spanning tree algorithm. In this example, only two edges contradict to the tree properties from graph theory — these are "clean"←"compile_A1" and "clean"←"compile_A2". Choosing either edge results in equivalent trees with respect to the number of linear extensions due to symmetry. By applying Atkinson's algorithm we get $S(P) < 70$ linear extensions and we get $\mathrm{Freedom}(P) < 0.4$, since$|P| = 9$.

On the other hand, the lower bound is computed very easily. Following the introduced algorithm, we end with three layers (l_1, l_2, l_3). From left to right:

- l_1 consists of 3 independent subsequences each of a single node. These are "A1.c", "clean", and "A2.c" and imply $l_1 = \binom{1+1+1}{1,1,1} = 6$.
- l_2 consists of 2 independent subsequences each of length 2: "compile_A1"←"A1.o" and "compile_A2"←"A2.o". Here we get $l_2 = \binom{2+2}{2,2} = 6$.
- Finally, l_3 consists of one subsequence of 2 nodes "link_A"←"A", that means $l_3 = \binom{2}{2} = 1$.

Since $S(P) > l_1 \cdot l_2 \cdot l_3 = 36$, $\mathrm{Freedom}(P) > 0.337$ holds and in total $\mathrm{Freedom}(\mathrm{P}) = 0.3685 \pm 0.0315$.

D. Optimization and Visualization

The Degree of Freedom is a metric for the static concurrency analysis, that measures the parallelization potential of a build process as a whole. While it is an objective measurement, it does not provide direct recommendations for optimization. However, in cases where the build process can be separated in several independent sub-processes (see Figure 5 on page 4), a domain expert would be able to detect high-potential regions by applying our metric.

Here we recommend our »**GAME**-changing« method with:

- The **Goal** to increase concurrency
- by taking **Actions** to restructure dependencies.

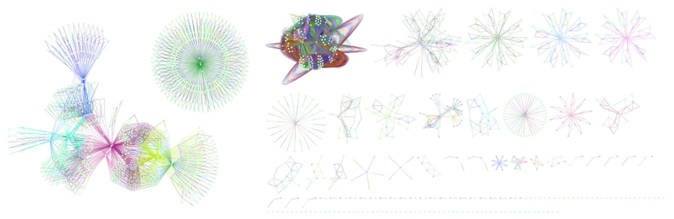


Figure 5: Visualization of Static Concurrency Analysis

- This is achieved by identifying the high potential using **Measurements** to detect large sub-process size with low Degree of Freedom and
- **Executing** the following steps:
  1) relationship analysis in high potential subgraphs by domain experts;
  2) alignment of process and artefact dependency graphs;
  3) reconnection of wrongly routed dependencies; and
  4) removal of unnecessary dependencies.

Additional visualization of the build process can also provide insights and support for further analysis. In Figure 5 on page 4, we provide an example visualization using Cytoscape [1]. Here we see the execution view of a (disconnected) build process, where each edge color represents different execution context, i. e., different execution scripts, different execution variable set, etc. Figure 5 on page 4.

IV. Related Work

There have been approaches and tools for both dynamic and static build analysis. In our previous work [18], we focused on the build process of Android and extracted the build jobs and dependencies by instrumenting the shell of GNU make. Gligoric et al. [12] intercepted file reads and writes in Windows by instrumenting Win32 functions using Detours [13]. In Linux there is the strace tool [14] for capturing such information like file read and write, process invocations, time measurement, etc. Moreover, there are tools for static makefile analysis, such as MAKAO [6], SYMake [17], and Makefile::Parser [4]. However, these tools are for general build extraction and analysis, and they do not support analyzing the concurrency of build execution.

V. Conclusion

This paper presents our approach for concurrency potential analysis we developed to support the maintenance of build system for constantly evolving software in big projects where the build process consumes an important amount of effort. We address these challenges providing our experience on dynamic and static analysis that we gained in our industry case. While conducting the concurrency analysis and optimization in this study, we have the following lessons learned:

1) Conduct dynamic concurrency analysis to monitor the concurrency performance of the actual build process.
2) Conduct static concurrency analysis to evaluate concurrency of the designed build process.

3) In order to optimize concurrency, deficits in the build system (e. g., obsolete/redundant build jobs) should be first fixed.
4) To prioritize concurrency optimization, consider to first focus on larger build subprocesses that have lower Degree of Freedom.

Moreover, we propose a new metric for measuring the concurrency potential of a build process. In fact, this metric is not limited to build analysis. It can be used for analyzing the concurrency potential of any executable process in general. Our implementation also provides a polynomial algorithm for computing lower and upper bounds for the number of linear exertions of a partial order set.

ACKNOWLEDGMENT

Thanks to our colleges from Fraunhofer IESE: Markus Damm for pointing to the right terminology by the problem formulation; and Sören Schneickert for many fruitful discussions and his constructive criticism.

REFERENCES

[1] Cytoscape. http://www.cytoscape.org, August 2018.
[2] ElectricInsight. http://electric-cloud.com/, August 2018.
[3] GNU Make. https://www.gnu.org/software/make, August 2018.
[4] Makefile::Parser. https://github.com/agentzh/makefile-parser-pm, August 2018.
[5] Ninja. https://ninja-build.org, August 2018.
[6] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. Design recovery and maintenance of build systems. In *Proc. IEEE Int. Conf. Software Maintenance*, pages 114–123, October 2007.
[7] M. D. Atkinson. On computing the number of linear extensions of a tree. *Order*, 7(1):23–25, Mar 1990.
[8] Graham Brightwell and Peter Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.
[9] Russ Bubley and Martin Dyer. Faster random generation of linear extensions. *Discrete Mathematics*, 201(1-3):81–88, apr 1999.
[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter 22.4 Topological sort, pages 549–552. The MIT Press, 2001.
[11] Chaabane Djeraba Dan A. Simovici. *Mathematical Tools for Data Mining*. Springer London, 2008.
[12] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. *SIGPLAN Not.*, 49(10):599–616, October 2014.
[13] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–144, Seattle, Washington, July 1999.
[14] D. V. Levin, R. McGrath, and W. Akkerman. strace. linux syscall tracer. http://sourceforge.net/projects/strace, August 2018.
[15] R. C. Prim. Shortest Connection Networks And Some Generalizations. *Bell System Technical Journal*, 36(6):1389–1401, nov 1957.
[16] Ivan Rival, editor. *Graphs and Order*. Springer Netherlands, 1985.
[17] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering 2012*, pages 366–369, September 2012.
[18] Bo Zhang, V. Tenev, and M. Becker. Android build dependency analysis. In *Proc. IEEE 24th Int. Conf. Program Comprehension (ICPC)*, pages 1–4, May 2016.