Runtime Information Integration into System Dependency Graphs

Alexander Barbie

Kiel University
Department of Computer Science
24098 Kiel, Germany

Abstract. Within the last decade, the importance of multi-core processors increased, due to a leak in performance improvement of single-core processors. As a consequence, software engineers need knowledge about concurrency issues. They must be qualified to meet performance requirements and to find bugs in concurrent programs.
We enhance a semi-automatic, pattern based approach to support software engineers in the parallelization process of sequential Java programs, by adding runtime information to the system dependence graph in a graph database. Therefore, we use an extension of Soot to find system dependencies and save them in a Neo4J graph database. The runtime information is gathered by Kieker. Further, we propose a tool, based on a pipe and filter architecture realized with TeeTime, to add these runtime information via Cypher queries to the Neo4J database. All gathered information can be visualized via the browser. This tool adds runtime information correctly to the graph database. However, the tool's correctness depends on the applications we utilize. If the system dependence graph contains errors, a possible relationship will not get the supposed runtime information.

1 Introduction

Since a decade the performance of single-core processors cannot be noticeably improved by increasing the clock frequency. [14] Therefore, multi-core processors provide a way out of the performance stagnation. They allow to compute processes concurrently. Since most of the applications we use are coded sequentially, the multi-core processors do not compute all assignments parallel. Hence, many parallelization approaches have been proposed [14], e.g, parallel compilers [2] or recommendation systems [7]. All these approaches have a leak of performance increase for many different applications, since they do not restructure the original source code. However, often performance improvements can be achieved by breaking dependencies to exploit further parallelization potential [11] . Fully automatic approaches inherit the issue that they need to over-approximate dependencies that are unknown or indeterminable at compile-time. All well known parallelization approaches have in common they need experts in concurrency issues of computer programs. [14]

In [14] the author proposes a semi-automatic parallelization approach for non-expert software engineers that provides solutions to the problems described above. The approach allows to iteratively introduce parallelization by applying a pattern-matching restructuring technique on the system dependency graph of the given software system. Due to the missing implementation of the runtime information in the system dependence graph, we enhance this approach and propose a tool to add runtime information to the system dependence graph.
Structure of this paper: Section 2 presents the foundations one must to know to follow our approach. We present our approach in Section 3 and evaluate it in Section 4. The conclusions and future work follow in Section 5.

2 Foundations

This enhanced approach utilized different Java APIs. In the following, these APIs are presented with a short description.

2.1 The Kieker Framework

Kieker is a monitoring framework that provides methods for dynamic analysis, i.e., for monitoring and analyzing a software system's runtime behaviour. It allows application performance monitoring and architecture discovery [5]. Furthermore, Kieker gathers and analyses monitoring data on different abstraction levels. It also records operation response times and traces of a software run [12]. A trace is an hierarchical data structure that combines single method calls to a tree of calls. Each method call knows the parent method in which it was called and its execution duration. Thus, a trace represents a sequence in which the methods were called. The monitored trace of an HelloWorld application is shown in Figure 1.

```
|-> main(..)
    |-> println(..)
```

Fig. 1. A reconstructed trace of an HelloWorld application.

2.2 The Neo4J Graph Database

Neo4j is an open-source graph database implemented in Java. The database offers a stand-alone server or is accessible as embedded database in Java applications. Further, software written in other languages can access Neo4J by using the Cypher query language through a REST interface. Information is stored in form of either a relationship, a node, or a property. Nodes and relationships allow any number of properties. Also, both the nodes and relationships can be labelled.

Labels can be used to narrow searches. New nodes, relationships and properties can be added at runtime by Cypher queries [8].
The example in Figure 2 illustrates a Neo4J database with three nodes and relationships. Each node is of type *Person*. Relationships can be of the types *loves* or *knows*. Cypher queries (see Section 2.6) can retrieve these nodes and relationships, manipulate them, or create new ones.

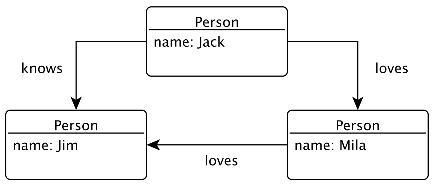


Fig. 2. Schematic graph database in Neo4J.

2.3 The Dataflow Analysis Framework Soot

Soot is a framework transforming and analysing Java applications. Since Soot is, at its core, a compiler, users can develop static analysis tools for Java programs [10]. There are many extensions that implement additional compiler phases, which analyse or transform Soot intermediate representations. Some key features are a simplified three-address intermediate representation of Java byte-code, a number of pointer analysis and call graph construction algorithms. Furthermore, it can produce executable Java byte-code as output [6].

2.4 The Pipe and Filter Framework TeeTime

TeeTime is a Pipe-and-Filter Framework for Java and was developed at the University of Kiel. It allows all users to create an analyses or filter (stage) in an easy way and contains many primitive and composite ready-to-use stages. Some of the contained stages already interact with Kieker logs (Section 2.1), like reconstructing traces from monitoring logs.
Teetime supports the possible performance improvement of pipe-and-filter architectures, since it allows a single-threaded, with no overhead, or a multi-threaded, with minimal overhead, execution [17]. The clean architecture of the abstract filters makes it easy to implement your own stages and connect them with other filters, on a type-safe way.

2.5 System Dependence Graph

A System Dependence Graph (SDG) is an extension of a Program Dependence Graph (PDG). While PDGs are used to model dependencies between statements

within a procedure, SDGs combine PDGs to model inter-procedural dependencies [4]. Hence, a SDG is a directed multigraph which maps out control and data dependencies between program statements. We categorise the statement according to whether they contribute to the program's structure or behaviour. As a consequence, each category is represented differently on the graph. Thus, the large number of different types of nodes and relationships makes it difficult to visualize the graph in an efficient way [13].
Figure 3 shows the graph representation of a method call that sums up two numbers. Due to length of this paper, the gentle reader may get detailed information in [13].

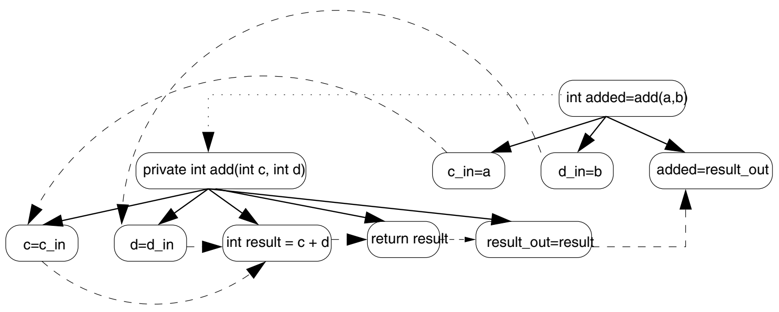


Fig. 3. Example of a simple method call in the SDG [13].

2.6 The Cyper Query Language

Cypher is a declarative graph query language and is used, e.g., in the graph database Neo4J. Its structure is borrowed from SQL. Since Cypher is a declarative language its focus is on what to retrieve from a graph, not on how to retrieve it [9].
Figure 4 demonstrates an example Cypher query. First, all relationships between the nodes n and m that got a directed relationship of type *AGGREGATED_ CALLS* from n to m, where the nodes got the names StartVertex and TargetVertex, will be retrieved. Afterwards, we add runtime information to the relationship, by setting the property *runtime* with a duration we recorded.

3 Approach

We aim to enhance the visualization of a sequential program in a Java System Dependence Graph (SDG, Section 2.5) using Neo4J and include runtime information for as many relationships as possible. We overtake the approach's

```
MATCH (n) -[r:AGGREGATED_CALLS]-> (m)
WHERE n.name = "StartVertex", m.name = "TargetVertex"
SET r.runtime = methodCallRuntime
RETURN n.name, m.name, r.runtime
```

Fig. 4. An example Cypher query, which adds the property runtime to a relationship.

utilized application programming interfaces (APIs). Consequently, we already got an API that creates and saves the SDG (Section 2.3), one that visualizes the SDG (Section 2.2), and one that monitors the runtime of method calls (Section 2.1). Following, we solely need to implement an application interface that writes the runtime information provided by Kieker to the SDG.
In the following we will name the tool, which saves runtime information from Kieker logs to a SDG in a Neo4J database, KiekerToNeo4J. KiekerToNeo4J will read the Kieker logs, reconstruct traces of method calls from that logs, and write all runtime information via Cypher queries into the graph database. A more precise presentation of the part we implement is given in Figure 5.

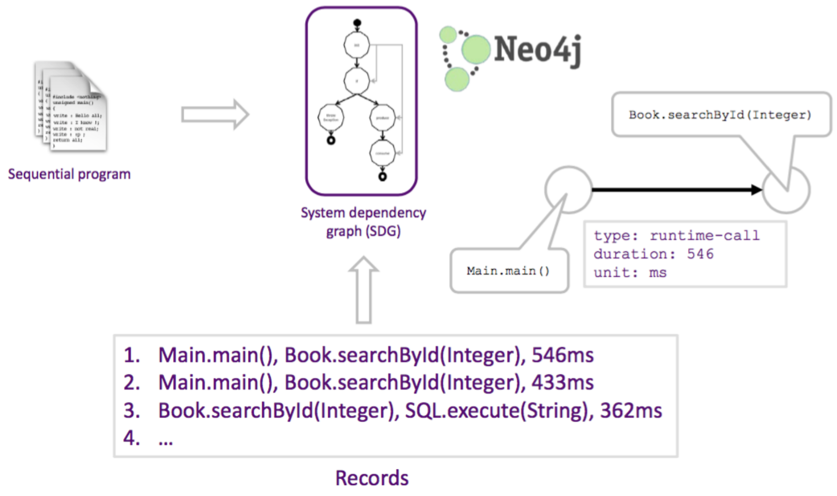

Sequential program
System dependency graph (SDG)
Neo4j
Book.searchById(Integer)
Main.main()
type: runtime-call
duration: 546
unit: ms
1. Main.main(), Book.searchById(Integer), 546ms
2. Main.main(), Book.searchById(Integer), 433ms
3. Book.searchById(Integer), SQL.execute(String), 362ms
4. ...
Records

Fig. 5. An overview how the approach will be enhanced [16].

3.1 Generate a System Dependence Graph with Soot

To give an compact overview of how this approach works, we take all steps on a very simple HelloWorld application. The Main-method prints "Hello World",

no further methods are included. The advantage is a small SDG with only one relationship to which we possibly can add runtime information. Although, the SDG is larger than only two nodes, the print method is the only method call. Hence, there is only one trace.
As described in Section 2.3, Soot has many different extensions. In this approach we use Soot Tutorial [15] to create a SDG of the HelloWorld application. This extension creates a SDG and saves it in an Neo4J graph database. We did not implement Soot Tutorial on our own. Hence, we have to assume that Soot Tutorial creates all SDGs correctly. Figure 6 shows the SDG of our HelloWorld application in Neo4J. This SDG contains several different types of nodes and

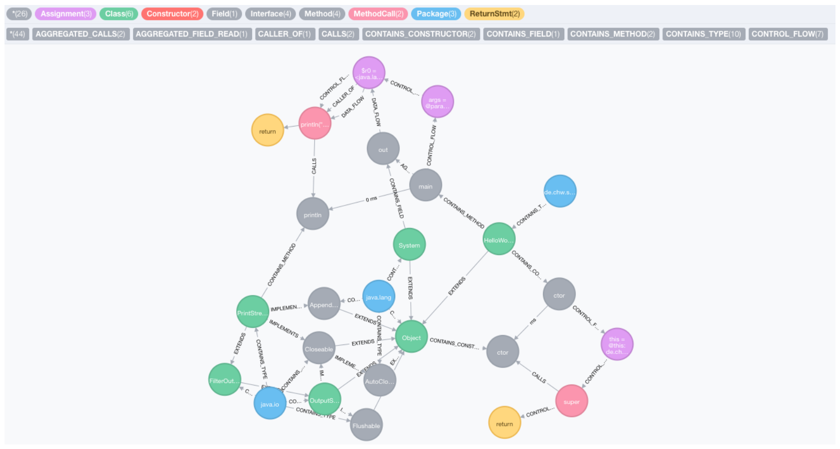


Fig. 6. A system dependence graph of an HelloWorld application.

relationships. Since we utilize Kieker to record method calls, we concentrate on well-chosen relationships of the type *AGGREGATED_ CALLS*.

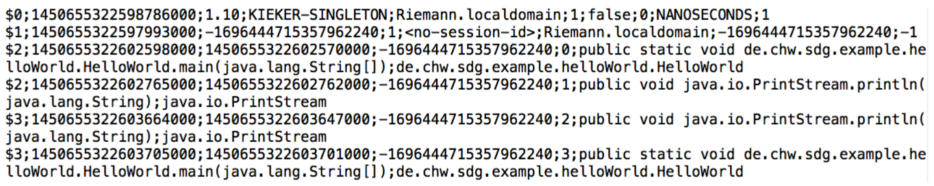
3.2 Retrieve the Runtime Information

Since Soot cannot add runtime information of method calls to the SDG, another tool has to monitor the same application that is saved as SDG in the database and gather the runtime information. Hence, we use Kieker (Section 2.1) to analyse the application and get its runtime information. Previously, we need to prepare our application.
First, we create an AspectJ aspect that puts a timestamp before and after each method call. Since Kieker has abstract aspects that match our requirements, we can utilized them in a configuration XML file. The difference of both timestamps represents the runtime of each called method. Secondly, we run Kieker and let it monitor our program [5]. It will gather all method calls, including all timestamps before and after each call, and save the results in logs in a separate folder. An

example log is shown in Figure 7. This log saves the type of a method call, the timestamps before and after calls, the called method, and also the method in which it was called. Thus, Kieker logs allow us to reproduce the call trace. A gentle reader discovers the log in Figure 7 contains only one call, although the SDG got far more nodes and relationships. It is important to remind that we only look for the runtime information of our application and do not monitor all internal Java calls. As a consequence, not all relationships in the SDG will contain runtime information. Kieker monitores the method calls that are represented in the SDG by the relationship *AGGREGATED_ CALLS*.
Further, does Kieker contain its own tools to run an analysis and returns different graphs, which represent the behaviour or structure of a monitored application. Since we cannot retrieve that data and put it in our SDG on an efficient way with these tools, we build our own tool for that purpose. This tool, named KiekerToNeo4J, is an enhancement of the original approach [14].

3.3 Recreation and Aggregation of Method Calls

First, KiekerToNeo4J has to retrieve the logs in a given directory. Secondly, all found logs have to be scanned line by line. Third, the lines that represent method calls will be gathered to create call traces. Afterwards, we aggregate multiple equal method calls. Thus, we use traces just to traverse through all method calls and to aggregate them. This aggregated method calls can be parsed to Cypher queries strings to add runtime information to the equivalent relationships in the SDG. Due to the sequential reconstruction process, we realize KiekerToNeo4J with a pipe-and-filter architecture. Therefore, we utilize the pipe and filter framework TeeTime (see Section 2.4).
TeeTime already contains ready-to-use stages to reconstruct traces from Kieker logs. Figure 1 shows the trace for the HelloWorld application. Additionally, we can use TeeTime to aggregate traces and to calculate some statistics to them, like minimal, maximum, and median durations of method calls. However, a trace contains a sequence of method calls in a tree structure. An aggregated trace does not aggregate its contained children. It aggregates equivalent traces. Since recursive method calls create new children for each call, these calls are not aggregated to a single child. In applications the same function can be called recursively thousands of times and each call would generate a new child on a deeper level in the



```
$0;1450655322598786000;1.10;KIEKER-SINGLETON;Riemann.localdomain;1;false;0;NANOSECONDS;1
$1;1450655322597993000;-1696444715357962240;1;<no-session-id>;Riemann.localdomain;-1696444715357962240;-1
$2;1450655322602598000;1450655322602570000;-1696444715357962240;0;public static void de.chw.sdg.example.he
lloWorld.HelloWorld.main(java.lang.String[]);de.chw.sdg.example.helloWorld.HelloWorld
$2;1450655322602765000;1450655322602762000;-1696444715357962240;1;public void java.io.PrintStream.println(
java.lang.String);java.io.PrintStream
$3;1450655322603664000;1450655322603647000;-1696444715357962240;2;public void java.io.PrintStream.println(
java.lang.String);java.io.PrintStream
$3;1450655322603705000;1450655322603701000;-1696444715357962240;3;public static void de.chw.sdg.example.he
lloWorld.HelloWorld.main(java.lang.String[]);de.chw.sdg.example.helloWorld.HelloWorld
```

Fig. 7. The Kieker log of an HelloWorld application.

tree. Thus, a trace of recursive method calls contains equivalent children multiple times, but on different levels. Since we are not interested in the sequence of method calls, we can filter each method call and aggregate them. Otherwise, the database could present thousand of relationships just for two nodes. Since this would be confusing, it harms our idea of finding parallelization potential for a non-expert software engineer. Furthermore, it is uninteresting if a method needed 1999 times the minimal time to proceed and once time the maximum time. The minimum and maximum duration of the aggregated method calls show exact the same at solely one instead of 2000 traces. Further, the median and average execution time are the important values a software engineer is looking for, if he wants to find parallelization potential of software. This median duration is the value that differs the maximum and minimum duration from chance. In Figure 8 is shown how KiekerToNeo4J reproduces the traces.

The stage *InitialElementProducer* retrieves all logs from a given directory. Af-

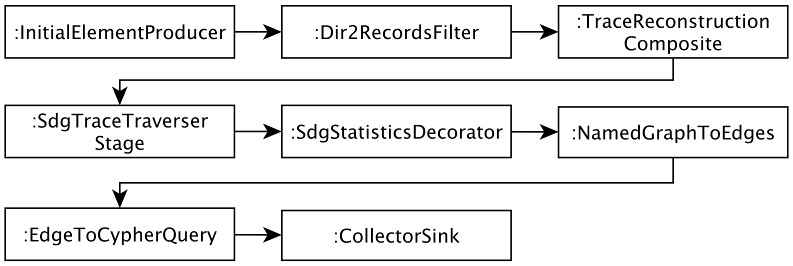


Fig. 8. Object diagram of KiekerToNeo4J

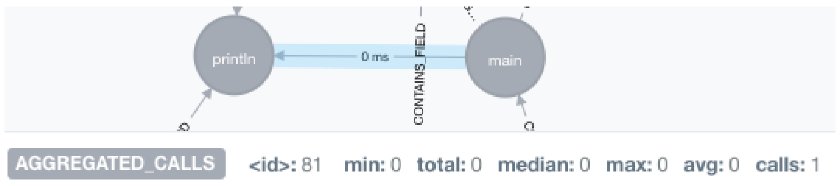
terwards, the logs will be read line by line by the *Dir2RecordsFilter* stage. This stage returns monitored records that are used to reconstruct a trace in the stage *TraceReconstructionComposite*. We traverse the trace in the stage *SdgTraceTraverser* and aggregate multiple equivalent method calls to an single method call. Equivalent means two method calls got the same parent method call and the same method declaration. All durations of equivalent calls are saved in a list in the aggregated method call. *SdgTraceTraverser* returns a graph that contains all methods (vertices) and method calls (edges). The stage *SdgStatisticsDecorator* gets the graph via its input port and computes the minimum, maximum, total, and median runtime for each aggregated method call. Afterwards, the Cypher queries can be created. Therefore, the stage *EdgeToCypherQuery* gets the graph's edges via its input-port and returns a generated string on its output-port. For each edge the Neo4J database contains a relationships. This relationship can be accessed with a Cypher query (see Figure 4).

3.4 Insert the Runtime Information to the Graph Database

The final filter we use in KiekerToNeo2J to import runtime information into a Neo2J graph database is the filter that transforms the graph's edges to strings, as shown in Figure 4. Each in the graph contained edge will be parsed to an Cypher query as string. Afterwards, the queries are collected in a list of strings. It is important to add the properties to each relationship, since Soot Tutorial did not set the properties while the SDG's creation.
Due to performance issues, we execute all queries at once. Otherwise, each query would provoke a new database connection and closing after that query. This is a possible source of error and performance bottleneck for our approach. KiekerToNeo4J uses the embedded Neo4J database. Hence all query strings can be executed in a loop.

3.5 Final Result

In Section 3 we presented our approach on a simple HelloWorld application. Since there is only one edge in the SDG, the method call *println*(..) from the *main*(..)-method, which can contain runtime information, we only need to check if the equivalent relationship in the Neo4J database contains all runtime information we retrieved. This relationship is shown in Figure 9. The runtime of $0\,ms$ is not a mistake, due to the *println* operation does not need to execute very long. Kieker monitors all timestamps in nanoseconds. Since method calls with a duration of just a few milliseconds are already represented by a large number in nanoseconds, we need a proper time unit. Humans cannot imagine and compare too large numbers very well. Thus, we convert nanoseconds to milliseconds and thus loose precision duo to rounding errors before adding the runtime information.



AGGREGATED_CALLS <id>: 81 min: 0 total: 0 median: 0 max: 0 avg: 0 calls: 1

Fig. 9. Runtime information for the method call from *main(..)* to *println(..)*.

4 Evaluation

Section 3 portrayed an approach to add runtime information of methods to a SDG. It was presented exemplary by a HelloWorld application. Hence, we have to show this approach works correct for larger applications, too.

4.1 Test Scenario

We evaluate the approach manually. Accordingly, we repeat all steps from Section 3 on a larger sequential application, than the HelloWorld application. In our evaluation this application will be named StaticAnalyzer. StaticAnalyzer reads recursively all Java files in a given directory, creates an abstract syntax tree (AST) representation, and saves all classes, methods, and loops in three separate HTML files. This application contains round about 80 callable methods. An overview of StaticAnaylzer's functionality is shown in Figure 10. The runtime of

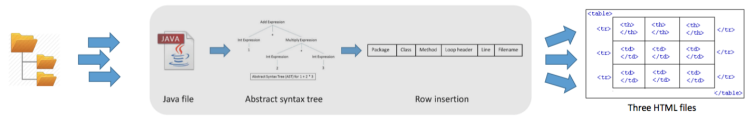


Fig. 10. Overview of StaticAnalyzer [3].

StaticAnalyzer depends on the specified filesystem, which is searched for Java files. In this example PMD's [1] Java source folder is used. PMD contains 331 Java files. StaticAnalyzer traverses the filesystem and reads all Java files. Meanwhile, Kieker monitores StaticAnaylzer while running on PMD and gathers all runtime information for each method call.

4.2 Methodology

This evaluation does not intend that Soot Tutorial creates a proper SDG. Further, we assume that Kieker monitors the runtime information correctly and TeeTime reconstructs all traces accurately, too. We focus on the criteria for a correctly added runtime to an relationship. Thus, we define our approach works properly if Kieker2Neo4J reconstructs a method call, e.g., from node A to node B, and adds the minimum, maximum, median, total, and average duration, and the number of calls to an equivalent relationship in the Neo4J database. Meaning, if the database contains a relationship from node A to node B, we add the mentioned information to that relationship.

4.3 Exprimental Setup

First a Neo4J database of the system dependence graph will be created by running Soot Tutorial [15] for StaticAnalyzer.
We define an abstract aspect to gather runtime information before and after each method call, via Kieker, in a XML configuration again. The results are saved in logs in a separate folder.
All steps taken before, set up the foundation for our approach. KiekerToNeo4J reproduces all traces, aggregates all method calls and writes them to the SDG

in the Neo4J graph database.
We launch our approach on a MacBook Pro (Midd 2010), including an Intel Core i5 (2,4Ghz) processor and 4GB DDR3 memory. This approach runs with the Java JDK 1.8, Neo4J 2.3, Kieker 1.12, and TeeTime 2.1 (including Kieker-TeeTime-Stages branch).

4.4 Results

Soot Tutorial [15] creates a SDG saved in a Neo4J graph database. The generated SDG contains about 1037 nodes and 3027 relationships. Due to the SDG's size, we can not present the whole graph. Subsequently, Kieker monitores StaticAnalyzer and generates logs far larger than the logs of the HelloWorld application. In Figure 11 we used Kieker's build-in analysis tools to create an assembly component dependency graph of StaticAnalyzer to be able to compare the Kieker results to the output of KiekerToNeo4J.

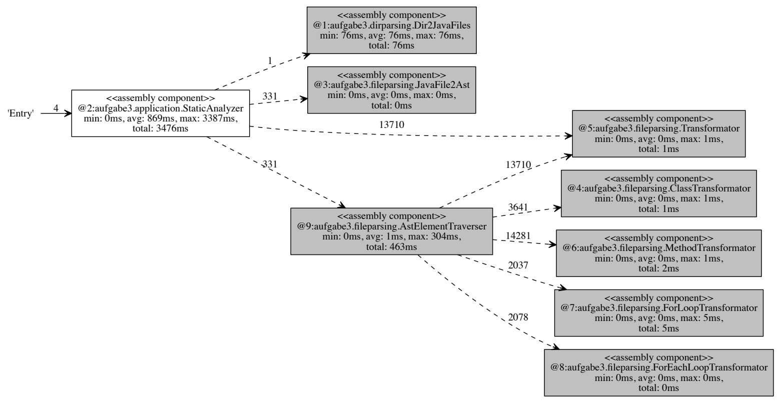
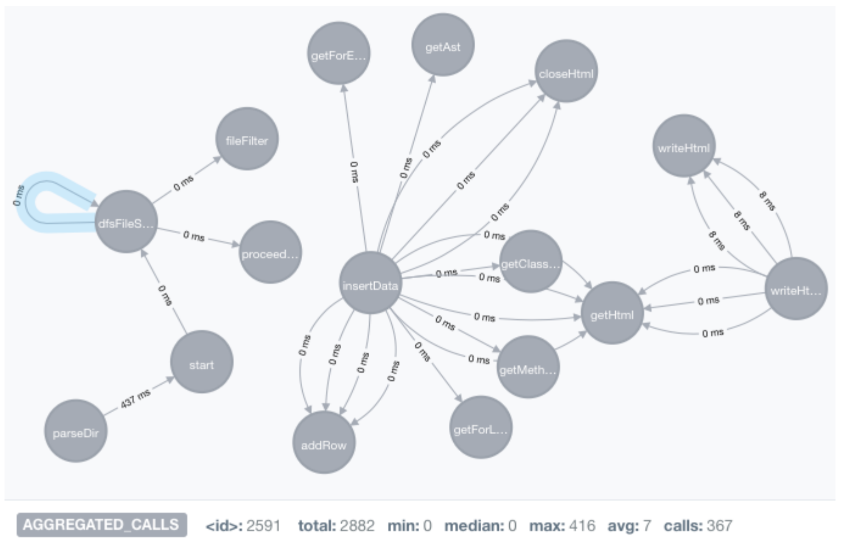

'Entry'
4
<<assembly component>> @2:aufgabe3.application.StaticAnalyzer min: 0ms, avg: 869ms, max: 3387ms, total: 3476ms
1
<<assembly component>> @1:aufgabe3.dirparsing.Dir2JavaFiles min: 76ms, avg: 76ms, max: 76ms, total: 76ms
331
<<assembly component>> @3:aufgabe3.fileparsing.JavaFile2Ast min: 0ms, avg: 0ms, max: 0ms, total: 0ms
13710
<<assembly component>> @5:aufgabe3.fileparsing.Transformator min: 0ms, avg: 0ms, max: 1ms, total: 1ms
331
<<assembly component>> @9:aufgabe3.fileparsing.AstElementTraverser min: 0ms, avg: 1ms, max: 304ms, total: 463ms
13710
3641
<<assembly component>> @4:aufgabe3.fileparsing.ClassTransformator min: 0ms, avg: 0ms, max: 1ms, total: 1ms
14281
<<assembly component>> @6:aufgabe3.fileparsing.MethodTransformator min: 0ms, avg: 0ms, max: 1ms, total: 2ms
2037
<<assembly component>> @7:aufgabe3.fileparsing.ForLoopTransformator min: 0ms, avg: 0ms, max: 5ms, total: 5ms
2078
<<assembly component>> @8:aufgabe3.fileparsing.ForEachLoopTransformator min: 0ms, avg: 0ms, max: 0ms, total: 0ms

Fig. 11. An assembly component dependency graph of StaticAnaylzer generated by Kieker.

KiekerToNeo4J reproduces 4 traces with 57387 method calls from the Kieker logs. These four traces can be found in Figure 11, too. Further, does the graph show that KiekerToNeo4J recreated all traces and method-calls. The following method call filter compounds them to 40 aggregated method calls. Hence, the sink stage collects 40 Cypher queries as string. All in all, runtime information were added to 29 relationships in the graph database. KiekerToNeo4J takes 808 ms to read the logs, 202 ms to create the traces, and 3368 ms to aggregate the method calls. Both stages, the one to compute the statistics for all method calls and the Cypher query generator stage got no measurable impact on the runtime. The execution of this 40 queries took $7348\,ms$. Due to the SDG's size of StaticAnalyzer, we cannot show the whole SDG. Thus, we only show the relationships we added runtime information to. 12. We will not discuss all



AGGREGATED_CALLS <id>: 2591 total: 2882 min: 0 median: 0 max: 416 avg: 7 calls: 367

Fig. 12. All nodes and relationships that contain runtime information in the SDG.

single relationships. Instead we show the results exemplary on the method call $dfsFileSearch$, which searches recursively for all Java files in a directory. This method was called 367 times and took in total $2882\,ms$ to execute. Following, each call was executed in average $7\,ms$. The fasted call was executed in minimum $0\,ms$ and the slowest in maximum of $416\,ms$, the median of all calls is $0\,ms$. There were no errors in the queries and it took about $11,726\,ms$ seconds to read the Kieker logs, reconstruct all trace, create aggregated method calls, parse the method calls to Cypher queries, and execute all queries.

4.5 Discussion

Due to the representation of a SDG, not all relationships illustrate a method call. A SDG contains many more dependencies, than just method calls. Accordingly, we cannot add runtime information to all relationships in the Neo4J database. Hence, there cannot be more than 40 relationships that hold runtime information. On the contrary, it would be a bug in our approach if all relationships would hold runtime information.
Kieker recorded 40 different method calls in StaticAnaylzer and KiekerToNeo4J executes 40 Cypher queries. StaticAnaylzer contains 80 method calls. Since

some methods are called multiple times or will not be recorded by Kieker, i.e., toString() methods, our results are valid.
4 of that method calls got no parent call. Since these 4 methods were executed in the constructor of StaticAnalyzer, this missing parent calls are marked as *'Entry'*. KiekerToNeo4J will parse this calls to Cypher queries, but Neo4J will not find any relationship that matches this query. Thus, only 36 relationships could possibly contain runtime information.
As shown in Figure 12, some methods got multiple identical relationships of $AGGREGATED\_CALLS$ to another method. Thus, some of the 29 shown relationships got equivalent runtime information. Ignoring equal relationships, we got 15 different relationships holding runtime information. However, 15 of possible 36 relationships looks like an error in our query generation stage.
On this account, we ran Cypher queries manually to look up the missing relationships. As a result, we found this relationships are missing in the SDG. Due to the usage of Soot Tutorial to create this SDG, it must be a failure in Soot Tutorial. It did not create all relationships from the application's code. Furthermore, do the SDGs created by Soot Tutorial differ from each other in different versions of Soot Tutorial. A further key aspect of the approach's correctness is its performance. Section 4.4 presents the runtime information of each step. From the point of the size of the Kieker logs, this seems to be a appropriate runtime. On the other hand, we also parsed all 57387 method calls to Cypher queries and executed them. Neo4J took $513272\,ms$ to execute all queries. All in all, KiekerToNeo4J needed about 8 minutes for all steps. Due to the time it took to parse and save the queries in a list of strings ($3652\,ms$) for that run, this larger runtime must be the reason of the time it takes the embedded version of Neo4J to execute the queries. In average each query was executed in $8.95\,ms$. Meaning, this is not a bug in our approach. However, Neo4J is a potential bottleneck in this approach.
All in all our approach works properly. Recursive or multiple equal method calls will be aggregated. KiekerToNeo4J parses all queries and executes them. Our discussion shows, that Soot Tutorial got bugs and does not create all relationships properly.

4.6 Threats to Vadility

We evaluated our approach manually. Therefore, we presented our approach on a simple HelloWorld application and evaluated the correctness with a larger application, named StaticAnalyzer. Due to the utilization of various open-source tools for our approach, we have to assume all of them work accordingly. Further, we cannot guarantee Soot Tutorial generated a proper and full SDG, since there is no evaluation that validates the correctness of Soot Tutorial.
All runtime information are taken by Kieker on one software run. Thus, the runtime information can be distorted, since all applications are executed on one personal computer. Hence, background processes can corrupt the runtime of method calls. However, we intended to show we add runtime information correctly to the SDG, not how they are measured.

5 Conclusions and Future Work

In this paper we enhanced an approach for finding parallelization potential based on a static analysis. We utilized an extension of Soot, called Soot Tutorial [15], to generate an Java system dependence graph, saved in a Neo4J graph database, of an HelloWorld application. Afterwards, the same application was monitored by Kieker to retrieve runtime information. All Kieker logs were read by our in Section 3 proposed KiekerToNeo4J tool. Since this tool is based on a pipe-and-filter architecture, TeeTime is utilized to implement this architecture. Furthermore, TeeTime already got some ready-to-use stages to reconstruct all traces monitored by Kieker. We implemented new stages to parse the traces to Cypher queries and to run them all in a embedded version of Neo4J. The results were presented in Figure 9.
Since the approach in Section 3 was proposed exemplary on a small HelloWorld application, we had to show in our evaluation (Section 4) that this approach works correctly on larger applications (StaticAnalyzer), too. Accordingly, we used Soot Tutorial again to create the Java SDG of this application. Afterwards, all method calls were recorded with Kieker. KiekerToNeo4J recreated all traces from the Kieker logs and executed all parsed Cypher queries. Since the SDG is to large to show it as whole, we presented a small part of the results in Figure 12.
All in all our tool works correctly, as far as we could evaluate it. The presentation of the SDG is confusing, consequently only the method-calls should be shown to find parallelization potential. Bugs can be in the implementation of Soot Tutorial [15], in the Kieker logs, or in the trace reconstruction filters of TeeTime. Furthermore, the execution of a large number of queries takes quite long. We indicated the embedded version of Neo4J as a bottleneck for our approach. Following, we maybe find potential here to increase the efficiency of our approach.

5.1 Future Work

In our approach we use the standard representation of a graph in Neo4J. This graph is far to large and confusing. As a consequence, we maybe gain not the full potential of our enhanced approach. Hence, the representation of the SDG could be optimized a lot. Additionally, a GUI could simplify the usage of Kieker2Neo4J. Unnecessary queries, like queries for method calls without a parent call, could be filtered out before execution.
Another aspect could be the usage of the REST interface Neo4J provides. Since Kieker can monitore programms written in other programming languages than Java, e.g., C [5], this approach could deliver the runtime information for a SDG in other programming languages. However, this implies the usage of Neo4J's REST interface. Since we propose an approach to add runtime information, we did not focus on the efficiency of this approach. Although it is based on a pipe-and-filter architecture, the whole approach has potential to parallelize tasks.

References

[1] T. Copeland and X. Le Vourch. Pmd - don't shoot me messanger, 2015. URL `http://pmd.github.io`.

[2] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):662–731, 2005.

[3] W. Hasselbring and C. Wulf. Assignment 5 (task 3) in software engineering for parallel and distributed systems, 2015.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[5] Kieker Project. *Kieker User Guide*, Apr. 2013. URL `http://kieker-monitoring.net/documentation/`.

[6] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[7] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th international conference on Computing frontiers*, pages 143–152. ACM, 2007.

[8] Neo4J. Neo4j website, 2015. URL `http://www.neo4j.com/`.

[9] neotechnology. The Neo4J Manuel v.2.3.1, 2015. URL `http://neo4j.com/docs/stable/cypher-introduction.html`.

[10] Sable Research Group. Soot. URL `http://sable.github.io/soot/`.

[11] A. Udupa, K. Rajan, and W. Thies. Alter: exploiting breakable dependences for parallelization. *ACM SIGPLAN Notices*, 46(6):480–491, 2011.

[12] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, Apr. 2012. ISBN 978-1-4503-1202-8.

[13] N. Walkinshaw, M. Roper, and M. Wood. The java system dependence graph. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64. IEEE, 2003.

[14] C. Wulf. Pattern-based detection and utilization of potential parallelism in software systems. In *Software Engineering 2014*, 2014.

[15] C. Wulf. Soot tutorial, 2015. URL `https://build.se.informatik.uni-kiel.de/gitlab/chw/SootTutorial`.

[16] C. Wulf. Slides: Runtime information integration into system dependency graphs, 2015.

[17] C. Wulf and N. Tavares de Sousa. Teetime - the next-generation pipe-and-filter framework for java, 2015. URL `http://teetime.sf.net/`.