Tracking Runtime Concurrent Dependences in Java Threads Using Thread Control Profiling

Lulu Wang[1], Jingyue Li[2], Bixin Li[1] *

[1] School of Computer Science and Engineering, Southeast University, Nanjing, China

[2] Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway

[Abstract] More than 75% of recent Java projects include some form of concurrent programming. Due to complex interactions between multi-threads, concurrent programs are often harder to understand and test than single threaded programs. To facilitate understanding and testing of concurrent programs, we developed a new profiling method called TCP (Thread Control Profiling). Outputs of TCP presents frequencies of control dependence, which includes thread creation, thread synchronization, interruption, and so on, of the executed thread. TCP first performs static analysis of detailed concurrency syntax and semantics of Java to construct the profiling graph model TCDG (Thread Control Dependence Graph). TCDG is then used for instrumentation and for generating profiles. We have evaluated TCP using a case study and a few experiments. The case study shows that TCP method can effectively prioritize test cases for testing concurrent programs. One experiment shows that outputs from TCP facilitate developers' understanding of concurrent code. Other experiments evaluate various possible overheads introduced by the TCP method. Results show that TCP can provide rich and useful information with reasonable costs.

Keywords: profiling, dynamic analysis, concurrent, Java thread, synchronization

1. Introduction

Concurrency is widely used in programs, and it allows threads' running interleaving and interaction. Concurrency has very language-specific semantics. Java is an object-oriented programming language that provides concurrent mechanism [5]. In practice, concurrency is a very commonly used mechanism in Java projects [65]. Different from single threaded programs, it is often quite challenging to decompose the messy interactions between multi-threads. As a result, concurrent programs are usually hard to understand and test. Programmers often misuse concurrent programming constructs. Only about 3% of projects handle thread exceptions, which may result in bugs or deterioration in applications' performance [65]. In real-world applications, it is almost impossible to ensure concurrent programs to behave as expected. It is therefore quite important to rank concurrent elements to show which elements should be given higher priority for testing and quality assurance.

In this paper, we use "execution frequency" to prioritize thread control dependences. Thread control dependence is the dependence between one running thread and another thread it controls. Thread control dependency is one of the most important aspects that leads to thread interaction complexity in concurrent programs. As unexecuted dependences could not contribute to or harm the program's function, the importance of a dependence to the program's execution is largely decided by its "execution frequency", i.e., profile.

Program profiling gives a dynamic view of running frequency in various program elements, including CFG (Control Flow Graph) edges [1] and paths [2], events [52], resources [3], dependences [22], and performances [4], and so on. Such profiles provide an insight view of the distribution on how a program runs in a large number of execution processes. The analysis results can be used in program analysis and improvement, including optimization, testing, and program comprehension [35, 33]. A profiling process usually contains three steps (a) probe instrumentation, i.e., inserting probes, which are statements that does not change the original program function, into the source code; (b) information collection during program execution; (c) profile generation, where the execution frequency of probes is calculated.

* Corresponds to bx.li@seu.edu.cn

To our knowledge, there has been no technique that profiles thread control dependences. In this study, we focus on a practical profiling technique that could directly be used on Java concurrent programs through utilizing the powerful synchronization scheme of Java. The purpose of the study is to develop tools to facilitate analysis, understanding, and testing of concurrent programs, especially concurrent Java programs. In concurrent Java programs, the execution order of program units is not determinate by the static structure. Instead, the execution order of program is decided when threads are generated and executed. The dependences and rules in threads are set in advance, but the running has infinite possibilities. So, it is not easy to understand, test, or analyze programs with concurrency. With profiles on thread control dependences, we can address such problems and get better knowledge of how threads affect each other at runtime.

The main issue we address in our study is how to generate profiles for threads' dependences in dynamic control flows. Our profiling method TCP is a novel method to detect thread control dependences for Java programs (with version JDK 1.7, also known as JDK 7). TCP first constructs the TCDG, then processes the instrumentation of TCP on TCDG, and outputs profiles as a result after executing the instrumented program. Our evaluation results show that TCP can provide rich concurrency information with reasonable cost.

The rest of this paper is organized follows: Section 2 explains concepts and theories related to this study; Section 3 explains design and implementation of our TCP method; Section 4 presents evaluation results; Section 5 summarizes and presents related work; and Section 6 concludes.

2. Related Concepts and Theories

It is quite difficult to write high-quality concurrent programs. Issues, such as dead lock, incorrect invocation, and thread-starvation, etc., make it difficult for programmers to write bug-free software.

A popular way to address these issues is to use formal verification, such as model checking [57]. Formal verification methods usually establish abstract approximations on how the concurrent program behaves and provide safe conclusions on if the concurrency is right or not. However, formal verification methods have the following disadvantages:

- Making models is very costly, especially on the complex, flexible, and large-scale real-world programs;
- The models are very hard to be totally accurate, because it requires very insightful and careful transitions from program to models;
- The verification outputs are probably presented in the state space instead of program statements, which makes it difficult to locate the bugs in the code and fix the bugs.

Profiling provides another way to help detect and correct bugs of concurrent programs. Since *reproducing the interaction between threads* is one of the key problems in the domain, it is therefore very useful to profile how threads control the execution of each other.

2.1 Related profiling techniques

Different from program tracing which collects comprehensive running data, program profiling is lightweight and provides execution frequencies of some parts of the program. The unified process of profiling usually includes three steps:

1. Instrumentation. The first step of a profiling technique is to insert instrumentation probes into the original program. The instrumentations are used to collect necessary information in program execution. The probes should never change the function of original programs, i.e., the two programs before and after instrumentation should have the same output with the same input.
2. Execution. During program execution, the instrumented probes make essential operations when they execute. The probes may encode or give a mark on how the profiling target is activated in current execution.
3. Calculation. In repeated execution of the program with instrumented probes, the frequency of the profiling target is added up efficiently (decoding should be performed if encoding is used

beforehand) to generate the final profile.

A profile consists of two parts: the subjects in programs to be profiled, and their corresponding execution frequency in program execution. To profile different targets in the programs, *probes* and *instrumentations* should be particularly designed, and the corresponding *calculations* should be kept consistent.

The most related profiling techniques are listed in Table 1, which shows the different profiling subjects (what to be profiled), and whether the profiling technique supports encoding, selective setting (to profile partial subjects instead of all subjects), cyclic, inter-procedural, and concurrent/parallel programs.

Table 1. Comparison of various existing profiling Techniques.

Profiling Technique	**Profiling subjects**	**Encoding**	**Selective**	**Cyclic**	**Inter-procedural**	**Concurrent/ Parallel**
[2]	Execution paths	√	×	×	×	×
[15,16,17,19]		√	√	×	×	×
[11,12,14]		√	×	√	×	×
[20]		√	√	√	×	×
[13]		√	×	×	√	×
[10]		√	×	×	×	√
[22]	Dependence paths	√	×	√	√	×
[53]	Instructions and events	×	×	√	√	√
[54]	Callpaths	×	×	-	√	√
[55]	Performance	×	×	-	√	√
[3,4]	Resource (memory, events, etc.)	√	×	-	√	√
[52]	Thread event	×	×	-	√	√
[56]	Memory access	×	√	-	×	√
TCP	Thread control dependence	√	×	√	√	√

In this study, our profiling target is the thread control dependences. Thus, our profile consists of control dependences and their execution frequency. A dependence in our study is described as either an encoding number or the program location (at which line of which file) with corresponding dependence types.

2.2 Related concepts

2.2.1 Control dependence

Control dependence [6] is a binary relation, which was first presented in program slicing. Control dependence is constructed from control flow graph and is intra-procedural.

Control Dependence

Suppose G is a control flow graph, and X and Y are nodes in G. Y is control dependent on X iff (1) there exists a directed path P from X to Y with every node in P (excluding X and Y) is post-dominated by Y and (2) X is not post-dominated by Y.

Inter-procedural control dependence can be used to describe the program elements that have influence on the execution of each other, such as method invocation, thread synchronization, and host communication. Thread synchronization and host communication are implemented by using method invocation with specific semantics.

Control dependence and data dependence interact with each other. They together give a complete view of relationship between program elements. Specifically, control dependence can be extracted independently, and is the basis of data flow analysis.

2.2.2 Java concurrency

Java and the Java virtual machine (JVM) support concurrent programming. In Java, code is executed as threads. Each thread is associated with an instance of the class *Thread*. Threads can be managed either

directly using *Thread* objects or using abstract mechanisms of thread method invocation.

Threads must be synchronized. Thread synchronization ensures that objects are modified by only one thread at a time. Thus, threads are not allowed to access partially updated objects, which are the objects being modified by another thread. Java has many ways, such as *Notify*, *FutureGet*, *Countdown*, *Signal*, and *Semaphore*, to implement synchronization.

Threads have also dependencies with each other. In Java programs, the thread dependence can mainly be classified into two categories:

- *Thread control dependence*. Thread α **directly** affects β's execution (including starting, suspending, resuming, and ending) by invoking specific methods[1] in concurrency mechanisms, and may therefore **indirectly** affect the variable values related to β.
- *Thread data dependence*. Thread α **directly** affects the variable values related to β and may therefore **indirectly** affect β's execution.

In our study, we focus on thread control dependence. Thread control dependence are widely used in concurrent program and could provide straightforward information on the control flow in multi-threaded program execution.

2.2.3 Java thread control dependences

There are many types of dependence graphs related to Java concurrency. Some of the dependence graphs (such as [48] and [49]) do not have detailed inter-thread dependences classification. Other dependence graphs (such as [44] [50] [51]) basically consist of several types of dependences, namely intra-thread control and data dependences, method invocations, parameter dependences, synchronization dependences, and inter-thread data dependence.

However, when we focus on the thread control dependences in Java programs, and when we want to know how one thread controls the running of another, the control dependency graphs generated from existing methods are not applicable, because:

- Control dependency graphs generated from existing methods contain data dependences, which makes the graph complicated.
- Different types of dependences may interweave in a Java-specific thread control dependence (e.g., in an "Authority", the dependences of invocation, parameter, and synchronization should all be used).

Therefore, a clear and fine-grained inter-thread dependences classification is necessary.

3. TCP design and implementation

In this section, we explain how to generate profiles of a certain program using our TCP method. Figure 1 shows the main process of TCP. Like general profiling process, our TCP method also includes three high-level steps, i.e., ***instrumentation***, ***execution, and calculation***.

TCP's ***instrumentation*** includes the following sub-steps:

- Static analysis of the source code of concurrent programs, which is used to detect:
  a) Thread objects. Finding out where a new thread starts with a thread object (whether anonymous or not) in the source code.
  b) Thread control dependences. Finding out the thread control dependences in the source code with established types and features.
- Constructing TCDG. Forming the TCDG by using nodes and edges of CFGs, and thread control dependences.
- Making instrumentation. Getting the necessary probes that should be instrumented into the source code based on TCDG, and then making the instrumentation.

[1] Including method invocations on a thread or a synchronization object. For example, one can start a thread's running by invoking its start(), or one can notify other related threads by invoking obj.notify().

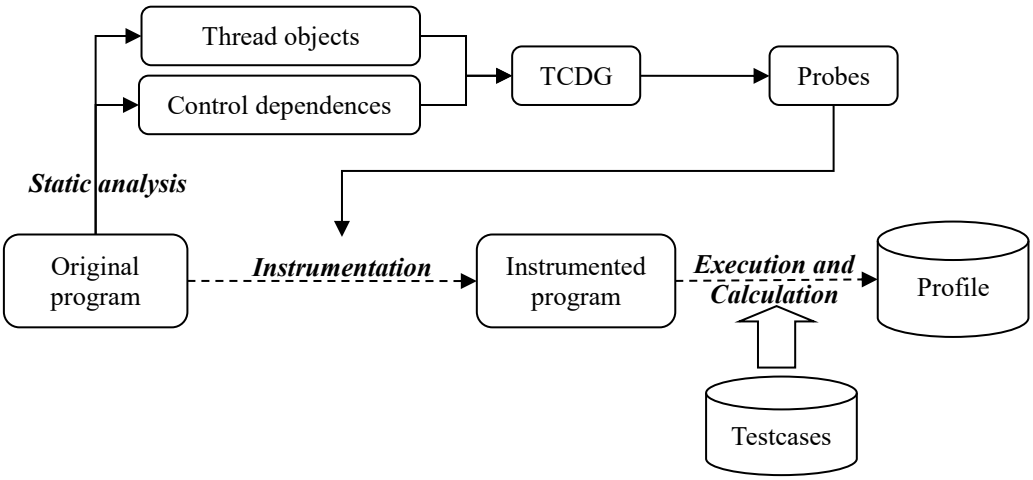


Figure 1. Process of TCP

In the following Sections 3.1 to 3.3, we give more detailed explanations of the ***instrumentation*** step of our TCP method.

- In Section 3.1, we explain how we summarize and classify Java thread control dependences into different types, and how we extract corresponding features of each type;
- In Section 3.2, we explain how the thread control dependences are automatically detected and used to construct the TCDG;
- In Section 3.3, we explain how the instrumentation algorithm is performed on the targeted program with TCDG, and how the probes are inserted into source code.

3.1 Our summary of thread control dependence types and their features

In Java programs, the thread control dependences can be classified into six categories, namely **Thread starting**, **Interruption, Notification**, **Authority acquire**, **Authority release**, and **Future-Get**. To the best of our knowledge, each type of thread control has the following features respectively.

Thread starting

A thread that is already running triggers a new thread through one of the following two possible method and gets the new one started.

- Initializing a thread object and invoking the method *start* explicitly.
- Initializing a thread object and invoking its starting implicitly, such as *ForkJoinPool*， *Executor,* and *ExecutorService*.

Interruption

An interrupt is an indication to a thread that the thread should stop what it is doing and do something else. The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking *Thread.interrupt* will set this flag to be true. In Java, an interruption tells a thread that it should stop. However, stopping or not is decided by the interrupted thread itself.

Notification

During threads' execution, the threads communicate and affect each other's execution. Usually two types of nodes are used to describe the notification relation: the *block node* and the *notify node*. A *block node* suspends the running of a thread under control, and a *notify node* continues its running.

There are four types of *block-notify* in Java programs:

- The methods *notify* and *notifyAll* are inherited from the super class Object.

- The elimination in *countdown* mechanism.
- The methods *signal* and *signalAll* for conditional variables.
- The statement *arrive* based on the Phaser classes.

For each type of *block-notify* in Java programs, the corresponding block nodes are *wait*, *await*, *await*, and *awaitAdvance* respectively.

Authority acquire and authority release

Two types of objects, i.e., *ReentrantLock* and *Semaphore,* are used to implement the mechanism of authority acquire and authority release to schedule threads, when the threads need access to the same critical resource.

ReentrantLock is a reentrant mutual exclusion lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities. A *ReentrantLock* is owned by the last thread that successfully locks, but not yet unlocks it. *ReentrantLock* is used mainly by invocations of two methods, i.e., *lock***()** to acquire the authority, and *unlock***()** to release the authority. A thread invoking *lock()* returns, after successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. [7]

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource [8]. Conceptually, a *Semaphore* maintains a set of permits. The corresponding method invocations are *acquire***()** and *release***()**. Each *acquire***()** blocks other acquirer, if necessary, until a permit is available, and then takes it. Each *release***()** adds a permit, potentially releases a blocking acquirer. However, no actual permit objects are used. The *Semaphore* just keeps a count of the number available and acts accordingly. In a word, *Semaphore* has a similar structure as *ReentrantLock*, but *Semaphore* is not reentrant.

Future-Get

A *Future* represents the result of an asynchronous computation. Methods are provided to check if a computation is completed, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using the method *get*(), when the computation is completed. The Future-Get mechanism blocks the access to the computation result, if necessary, until it is ready [9].

In a word, the mechanism Future-Get is used to access the results of multi-threads asynchronously. Future-Get is used by invoking ***t.get*()** or ***t.join*()**, where ***t*** is the thread object, and the type of ***t*** should be *Thread*, *Runnable*, *Callable*, *FutureTask,* or *RecursiveTask*.

3.2 How to detect and construct TCDG

By using the features of all thread controls mentioned above, we can detect the necessary control information from Java source code. To construct TCDG, we will identify the thread object, which is used to operate the thread, and then construct various types of thread control dependence edges. To help find a proper way for instrumentation, TCDG includes the inter-thread control dependences and the intra-thread control flows.

We first need to figure out that a thread object appears. There are eight ways in total to create a thread object:

- From a class that extends *RecursiveTask*<T>，*FutureTask*<T>，*RecursiveAction,* or *Thread*.
- From a class that implements *Callable*<T>，*Future*<T> ，*RunnableFuture*<T>, or *Runnable*.

When an object, which is created from one of the above class, is found, we can figure out that a thread object appears.

In our study, we focus on six types of thread control dependences. By extracting information from source code using source code analysis, we can generate all six types of thread control dependence edges with the corresponding features. The list of edge types and their structures are shown in Table 2.

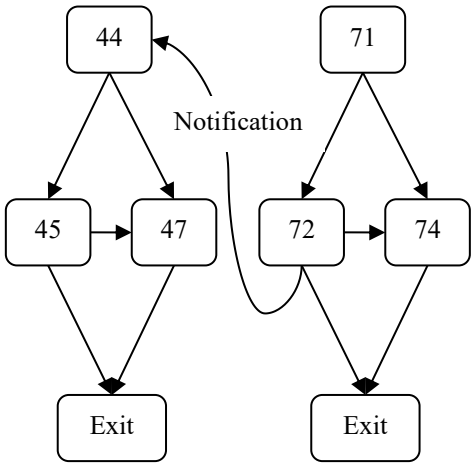
Table 2. The structure of dependence edges.

Edge type	Edge source node	Edge target node
Thread starting	The statement that starts a thread.	The entrance of the started thread.
Interruption	The statement that interrupts a thread.	The entrance of the interrupted thread.
Notification	The statement that notifies a thread.	The statement that blocks the same thread.
Authority acquire	The declaration of the lock object.	The statement that acquires the lock.
Authority release	The declaration of the lock object.	The statement that releases the lock.
Future-Get	The statement that invokes get() or join().	The exits of all the waiting threads.

To construct TCDG, we will utilize thread objects, control dependences, and CFG information. Figure 2 shows an example of TCDG constructed from the source code acquired from the class *com.dianping.test.concurrent.CountdownLatchTest* on Github. Figure 2(a) shows that *latch.await*() makes the current thread wait, while *latch.countDown*() notifies it. Thus, as shown in Figure 2(b), there is an edge in TCDG from the latter statement (i.e., statement 72) to the former one (i.e., statement 44) with the type *Notification*.

```
public class CountdownLatchTest {
    ...
42    public void run() {
43        try {
44            latch.await();
45            System.out.println("Latch barrier passed for " + this);
46        } catch (InterruptedException ex) {
47            System.out.println(this + " interrupted");
48        }
49    }
}

class TaskPortion implements Runnable {
    ...
69    public void run() {
70        try {
71            doWork();
72            latch.countDown();
73        } catch (InterruptedException ex) {
74            // Acceptable way to exit
75        }
76    }
}
```



(a) (b)

Figure 2. Example of TCDG construction: (a) the source code; (b) the TCDG graph

TCDG is a static view of thread controls. Since the same code section may map into several threads in program execution, the threads need to be separated at runtime by using instrumentation.

3.3 How to perform program instrumentation

Program instrumentation inserts probe statements into the source code, which record necessary information accompanied with the source code execution, but do not change the original functions of the source code.

3.3.1 Preparation

It is critical to answers two questions before applying any profiling techniques, and especially before the TCP method.

- What is necessary to record? To observe how thread controls happen at runtime, it is necessary to record not only which code affects another thread, but also the location of the code where the effect originates. Instrumentation needs to record three key information in program execution, i.e., the location, the edge type, and the thread.
- Where to put the instrumentation? In Java syntax, functional statements that can be executed in program execution are mainly allowed in methods, i.e., code blocks. It is illegal to instrument

around independent statements that declare fields. For each type of control dependence edge listed in Table 2, either or both of its source and target should be a statement, which can be instrumented. When such a statement is executed, the dependence edge is activated.

3.3.2 Probe generation

Ball et al. present an algorithm for optimal instrumentation on edges, which can record all vertices and edges in profiling with minimal cost on probes [1]. Our study is different because we focus on concurrent programs. In the scenario of concurrency, the executions of instrumented probe statements may not be totally independent and arbitrary skips of profiling calculation may happen.

Our TCP instrumentation algorithm is described in Algorithm 1, which generates probes from the TCDG. The complexity of our TCP instruction is $O(n^2)$ where n is the total number of thread control dependences in TCDG[2]. Algorithm 1 is explained as follows:

- The main idea of Algorithm 1 is: **successive vertices** should not be all instrumented, since execution of one implies the execution of others.[3]
- Step 1 turns all thread control dependences into probes. Particularly, the *Authority acquire* and *Authority release* edges make probes by their source nodes (because the source is an executable statement, while the target is a declaration of authority object), while other edges make probes by their target nodes (because a control operation is completed at the target).
- Step 2 inspects the CFG of each thread and tries to find some probes that could be replaced by existing ones, so that the probes which can be replaced do not need to be instrumented. Such a "replacement" is recorded by "***omittedProbes***".
- Step 3 removes the replaceable probes and outputs the final instrumentation results.

Algorithm 1: Instrumentation on TCDG

Input: Graph ***tcdg***
Output: List<Probe> ***probes***
Map<Probe, Probe> ***omittedProbes***

```
/*Step 1: make probes for all thread control dependences*/
for(Edge e : tcdg.getThreadControlEdges()){
 if((e.type = "Authority acquire")|(e.type = "Authority release"))
      probes.add(new Probe(e.target, e.type));
 else
      probes.add(new Probe(e.src, e.type));
}
/*Step 2: recognize the successive nodes with probes*/
probes.sort();                    //sort by file first, by line with same file
for(int index = 0; index < probes.size; index++){
       Probe p1 = probes.get(index);
       Probe p2= probes.get(index+1);
       if(p2.node is the only successor of p1.node in CFG)
            omittedProbes.put(p2, p1);
            // The omitted probe is the key, and its representation is the value.
}
/*Step 3: remove unnecessary probes*/
probes.removeAll(omittedProbes.keys());
```

Algorithm 1 has two outputs. *Probes* are used for instrumentation. The reason for having ***omittedProbes*** is that we can use the frequency of the replacer to identify the frequency of replaced probes. For example, if $<p_1, p_2>$ and $<p_2, p_3>$ are all in *omittedProbes*, and p_3 is found in the profile with two executions, then we should count two executions of p_1 and two executions of p_2 as well. It is because p_3 is a replacer of p_2

[2] In Algorithm 1, Step 1 costs $O(n)$, Step 2 costs $O(nlogn)$, and Step 3 costs $O(n^2)$.

[3] A statement between two thread control statements may be an invocation that causes certain termination, so the probes on pre-dominators or post-dominators cannot be saved.

and p_2 is a replacer of p_1. By using ***omittedProbes,*** we can get the profile of the replaced probes and the profile of their replacers.

3.3.3 Instrumentation implementation

The implementation is used to inject the probes into the source code. Like normal profiling techniques, we use one Java statement to implement each probe. Our instrumentation is implemented using a new method called *record* and its invocations. We define the *record* method as follows.

record(String *file*, String *line*, String *edgeType*, long *threadID*)

The *file* and *line* parameters are used to record the location of the code executed, and the *edgeType* and *threadID* are used to record information of the type of the edge and the ID of a thread.

The *record* method outputs the collected information to a file on disk. This method *record* itself is put in a class called Probe and is stored in an extra package. The extra package needs to be copied into the target project to make it possible to invocate the *record* method. For example, an invocation of the *record* method is:

profile.Probe.*record*("D:\\Suite.java", "68", " *authorityRelease0* ", Thread.currentThread().getId());

where *authorityRelease0* means that this is the source of an authority release edge, and the target node of the edge would be *authorityRelease1* correspondingly.

With encoding of TCDG edges, the invocation can be simplified into

record(int *edge*, long *threadID*)[4]

and the profile would be greatly compressed.

In addition, the instrumentation is only processed within accessible source code. Thread controls related to the external jars cannot be instrumented.

After the instrumentation, like other profiling techniques, profiling results are produced when instrumented program executes.

We have implemented the TCP method in Java based on *ASTParser* (a tool which extracts the abstract syntax tree of Java source code, from JDT (Java Development Tools)) for our experiments. In addition, we use several plug-ins:

- We use *org.eclipse.jdt.core* and *org.eclipse.equinox.common* for parsing source code of the programs to be profiled and for creating the TCDG.
- We use *org.eclipse.text* for recording our instrumentation and for mapping it back into the source code files.
- We use *jxl* for exporting experimental results into an excel file.
- Other jars are used only if they are imported by the programs to be profiled.

4. Evaluation and results

Zhao et al., analyzed the program dependences in concurrent programs, and discussed about the applications on slicing, debugging, testing, and understanding [58]. The purpose of our study is to use profiling to facilitate test prioritization and code understanding of concurrent programs, without bringing too much overhead. To evaluate our method and code, we try to answer three questions in our evaluation.

- RQ1: How well can our approach and results of profiling facilitate test prioritization?
- RQ2: How well can the profiling information help code understanding of concurrent programs?
- RQ3: How much overhead does our profiling bring to execution of the original code?

[4] Here the edges are encoded, and one integer represents one edge.

4.1 How well can our approach facilitate test case prioritization?

Test priority is very useful for mass test cases or test tasks with limited resources. It is even more useful in concurrent programs, because testers may have too many targets to test, and because even the same test cases may generate different outputs in different executions.

Researchers have considered to improve testing efficiency by priority assignment [61], which uses metrics and weights to show which modules should be tested first. The performance of the technique in [61 is subjective and is quite dependent on testers' experience. Thus, different focuses or solutions of the testers may give diverse outcomes.

In contrast, our TCP method is not affected by experience, and may be more suitable for general uses. Here we investigate if the profile helps ***regression*** test case prioritization of concurrent programs. In the evaluation, we use Junit4 as the case study. Junit4 is currently the most popular version of Junit. Junit is a testing tool, and an instance of the xUnit architecture for unit testing. We chose to use Junit4 because:

- It is a concurrent Java program, because Junit4 uses threads to run test cases independently.
- It is often used as a benchmark in program analysis.
- It has well-established test suite, which can help us answer RQ1.
- Its test cases are easy to understand and can therefore be used as function description to help us answer RQ2.

We use three test suites (*junit.tests.framework. junit.tests.runner, and junit.tests.extensions*) provided by *junit.tests.AllTests*. The three test suites all together consist of 114 test cases. Our evaluation (as shown in Figure 3) has several steps.

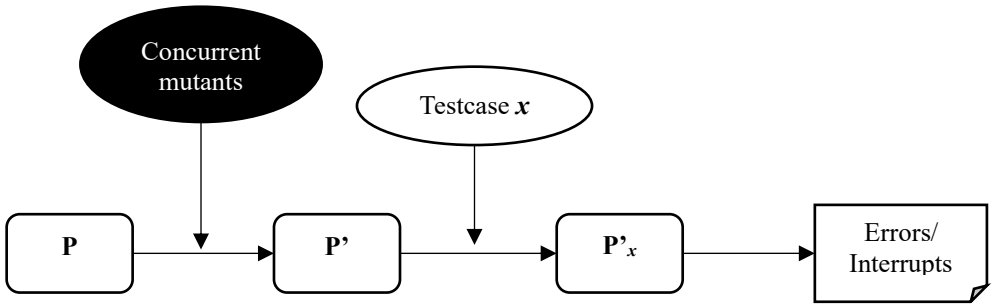


Figure 3. The process of using mutant testing to demo how TCP can help prioritize tests

Step 1. Prioritizing test cases for regression testing. We first generate the TCP profile of Junit4. Then, we compute the test priority of each test case. The test cases are prioritized based on the accumulation of all the frequencies of the thread dependences in their execution, as shown in the following formula. The test cases with higher priorities should be tested earlier.

$$priority(\text{tc}) = \sum_{td\ runs\ with\ tc} frequency(td)$$

Step 2. Injecting mutants. Two types of mutants (Type A mutants would cause an error[5] with the probability 1‰, while Type B mutants would cause an interruption[6] of the current thread with the probability 1‰) are separately injected into the source code on every thread control dependences (the original version is called **P**, and the injected version is called **P'**, as shown in Figure 3).

Step 3. Executing regression tests to kill mutants. Each test case ***x*** is separately used to kill all the mutants in its running paths, and the fixed version is called **P'**$_x$ correspondingly. All the 114 versions of **P'**$_x$ are executed to collect the number of errors or interrupts that are not killed. The more errors or

[5] Here we use a general type of error instead of specific types.

[6] An interruption causes the current running thread to interrupt.

interrupts $\mathbf{P'}_x$ has, the worse testing effect the test case $\boldsymbol{x}$ has, because $\boldsymbol{x}$ could not find the mutants that should have been killed.

After step 1, based on the accumulation of all the frequencies of the thread dependences a test case executes, the 114 test cases can be categorized into three priority sets, namely (5000, 6000), (6000, 7000), and (7000, 8000).

Results of step 3, as shown in Figure 4, illustrate that, on average:

- Test cases in priority set (5000, 6000) have left 11.57 errors (with Type A mutants) and 9.32 interrupts (with Type B mutants) in $\mathbf{P'}_x$ that are not killed.
- Test cases in priority set (6000, 7000) have left 5.97 errors and 6.00 interrupts that are not killed.
- Test cases in (7000, 8000) have left only 0.17 errors and 0.00 interrupts that are not killed.

It is obvious that test cases which are classified as high priority by using our TCP profile have killed more mutants in the regression testing than those test cases that are classified as low priority. So, we can conclude that TCP is helpful to prioritize test cases for testing concurrent program.

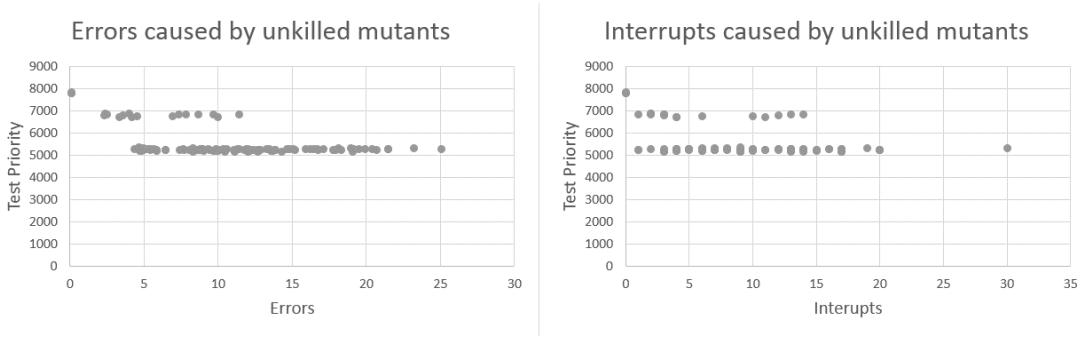


Figure 4. The errors and interrupts for 114 versions of $\mathbf{P'}_x$

4.2 How well can profiling information help developers understand the code?

In legacy software, it is necessary for developers to understand the program first, before they repair, reuse, or optimize the code [59] [60]. To facilitate effective code understanding, it is very important to show to developers which is the key function or element of the program. In Java concurrent programs, the main burden (of Java concurrency constructs) still falls on the programmer's thorough understanding of concurrency and its implications [65].

We believe that TCP results may help developers understand concurrent Java programs through decomposition of concurrent programs. Program decomposition (such as slicing) removes irrelevant parts from the program, so that the remaining parts are smaller and easier to understand. TCP results could show how many threads are not included in execution. If some threads are irrelevant to the execution, the threads should be regarded as noise and disturbance, with respect to understanding the program's function during its execution.

We extract information from TCP and use the information to identify which declared thread-control program elements (packages, classes, methods and statements)[7] are irrelevant to the program entrance in current program execution. Results of a case study using Junit4, as shown in Figure 5, illustrate that more than half of the elements can be detected as irrelevant to executions of all **Main** methods in the source code. In real-world applications, it is well-known that a certain program function is only related to a small part of source code. TCP can effectively show, under the current execution, which part of the concurrent program is related to the thread control code. Thus, TCP could narrow down the scope of necessary code analysis, and improve the code understanding efficiency.

[7] The inner classes are identified separately. The statements are counted only if they lead to thread control dependences, i.e., we ignore the statements without thread controls.

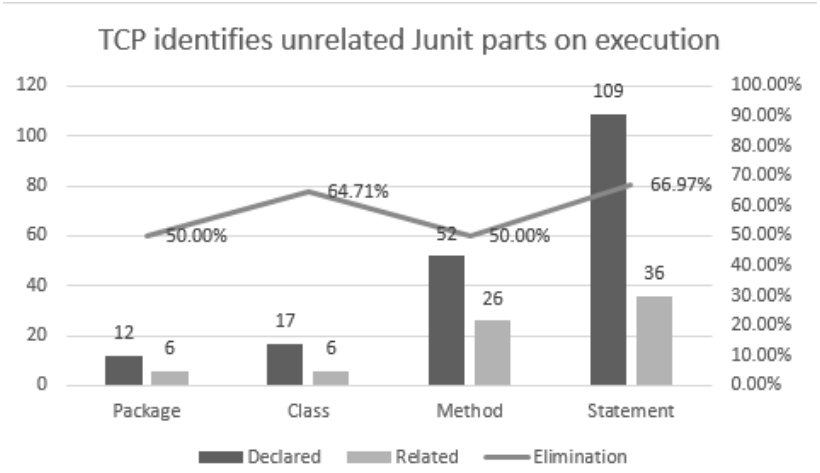


Figure 5. Unrelated concurrent program elements detected by TCP on Junit4

To give evidence that TCP can help developers understand concurrent programs by narrowing the program range, we perform an experiment with sixteen junior programmers, who are mostly the first or second year software engineering Master students.

For program understanding, we investigate how well developers can identify which part of the code is related to which functions of the program. Accordingly, this experiment is designed and executed as follows.

- First, we give test cases to the subjects as descriptions of the program's functions. We choose 16 test cases from Junit4, because the test cases do not go through the same code paths during execution.
- Then, the Junit4 source code is distributed to the subjects to read and to understand.
- After reading and understanding the code, the subjects are asked to answer how two test cases are related to different code. We design 20 single-choice questions (Q1-Q20) with three options (as listed in the **Appendix**).

To compare how well developers can understand the code by using the TCP vs. without using TCP, we ask half of the subjects (A, B, …, and H in short) to answer Q1-Q10 without TCP information, and Q11-Q20 with the TCP information. The other half of the subjects (I, J, …, and P in short) did the opposite, i.e., answering Q1-Q10 without TCP information, and Q11-Q20 with the TCP information. For each subject, the whole experiment, including code understanding and answering question, is limited to 600 minutes.

By computing how well the subjects answer the questions, we can compute their code understanding accuracy. The code understanding accuracy of each subject is shown in Figure 6, and illustrate that:

- The average accuracy overall on Q1-Q10 is 66.88%, and on Q11-Q20 it is 67.50%. This shows that the two groups of questions have similar difficulty, and Q11-Q20 may be a little harder than Q1-Q20.
- Twelve out of 16 participants answered more accurately after given TCP information. Three participants (i.e., A, D and P) had the same accuracy with and without TCP information. One participant (i.e., B) answered more inaccurately with TCP information. On average, the code understanding accuracy without TCP information is 59.38%, and the accuracy with TCP it is 75.00%. Results show that the TCP result is helpful for program understanding on most people and is almost harmless for the rest.
- In order to test the difference between the two groups of accuracy data with/without TCP information (named G_1 and G_2), we formulated null hypothesis and an alternative hypothesis. The null hypothesis is "*there is no difference of accuracy of understanding the code between using the TCP information and without using the TCP information*." The alternative hypothesis is "*there is difference of accuracy of understanding the code between using the TCP information and without using the TCP information*" To test the hypothesis, we performed *Independent Samples T-Test*.

The raw data of the T-Test are in Table 3 and the results show that:

- G_1 and G_2 both pass the normality test (Kolmogorov-Smirnova), with the significance 0.184 (G_1) and 0.200 (G_2).
- The F value is 0.022 (<0.05) and the statistical significance is 0.883, which supports that G_1 and G_2 are different. Then we reject our null hypothesis and conclude that using TCP information can improve the developers' understanding of the code significantly.

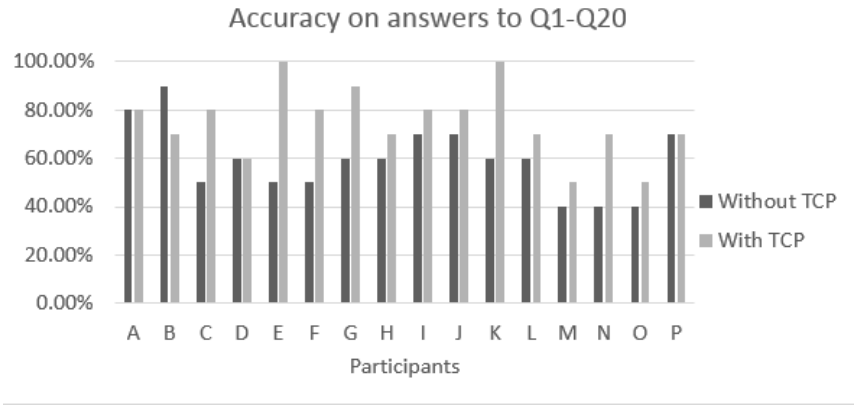


Figure 6. The accuracy of answering questions Q1-Q20

Table 3. Number of correct answers of Q1-Q20 in group G_1 and G_2 (The underlined numbers are the numbers of correct answers provided by the participants through using TCP)

	Participants															
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q1-Q10	8	9	5	6	5	5	6	6	8	8	10	7	5	7	5	7
Q11-Q20	8	7	8	6	10	8	9	7	7	7	6	6	4	4	4	7

After the experiment, to understand the reasons for getting a better code understanding with TCP information, we interviewed all participants to list their interpretations why TCP is helpful or not helpful for code understanding. We coded their answers and classified their answers into the following four categories. The listed reasons are as follows. The numbers in the parentheses indicate how many participants out of the 16 participants list the reason.

- TCP indicates the thread interactions and lead them to understand the process of program execution (10/16).
- TCP shows some potential similar test cases, which could be understood together (7/16).
- TCP limits the scope of threads and removes some irrelevant program parts (7/16).
- TCP gives hints on some key program points, which may be easily overlooked without TCP (5/16).

4.3 How much overhead does our profiling bring to execution of the original code?

To study the possible overhead introduced by our TCP method, we performed several experiments to investigate its instrumentation cost, runtime cost, and storage cost. The instrumentation cost is to evaluate how many statements are necessary to be inserted into the source code. The runtime cost is to evaluate how much extra time is needed in program execution by instrumentation. The storage cost is to evaluate how extra disk is needed to store the profile information. Our experiments are executed in the following environment:

- Windows 7 with Eclipse Neon and JRE 1.8
- Intel i5-3337U (2.40 GHz) CPU
- 4G memory

To measure the overhead, we choose five Java projects (shown in Table 4) as our benchmark projects. The benchmark projects are selected based on three criteria:

- The project should contain at least one thread control type in its source code.
- The project should allow inserting instrumentation, compiling, and running the code, the project should contain the files ".project" and ".classpath", and can be restored into its development status.
- The project can be invoked with all **main** methods to execute some thread controls and will return no errors.

Table 4. Information of benchmark projects.

Benchmark	LOC	LOC after instrumentation
xusleep/Concurrent (C1 for short)	7,132	8,074
zhongliangjun1/Concurrent (C2 for short)	975	1,226
Junit4 (Concurrent)	34,959	35,465
Dacapo (Concurrent)	6,479	6,688
NAS Parallel Benchmarks (NPB for short)	22,380	24,331

The corresponding TCDG information of each benchmark project is shown in Figure 7.

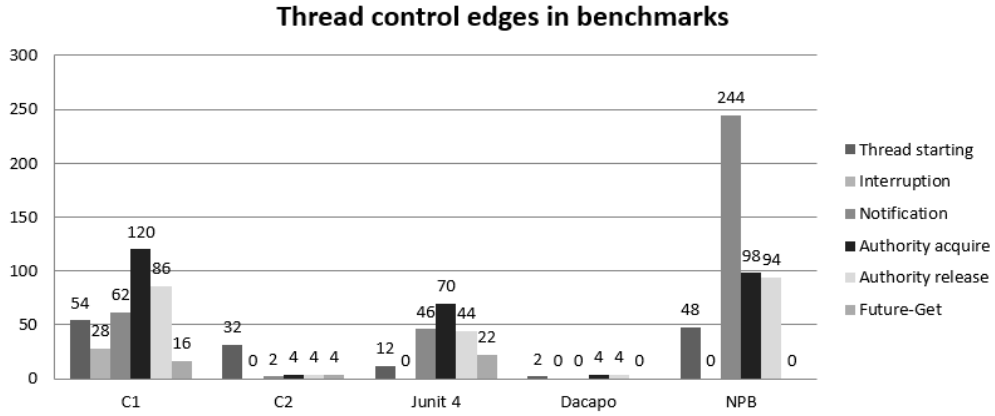


Figure 7. TCDG information of benchmark projects

4.3.1 Instrumentation cost

The instrumentation consists of two parts, one is the extra package for probe execution, and another is the statements at each thread control point. In our experiment, we calculated, on average, how many statements are necessary to be inserted for each thread control dependence. The calculation results are shown in Table 5.

Table 5. The instrumentation costs.

Benchmark	Instrumented statements	Thread controls	Lines inserted per thread control
C1	942	366	2.57
C2	251	46	5.46
Junit4	506	194	2.61
Dacapo	209	10	20.9
NPB	1951	484	4.03

4.3.2 Running cost

To measure running cost, as a preparation, all the *System.exit* in the source code is turned off, so that the whole running is not interrupted by any internal operations. To collect the cost data,

- All methods named as **main** in any classes are invoked by the Java reflection mechanism with a default class instance.
- Each **main** method is putting a new thread to run independently with a time interval of 50ms.
- The original and instrumented programs are both iteratively executed for ten times to eliminate noise.

The average runtime cost is shown in Figure 8, which indicates that the instrumentation brings 8.90%, 1.32%, 5.07%, 0.26% and 3.38% more running time for the five benchmark projects respectively.

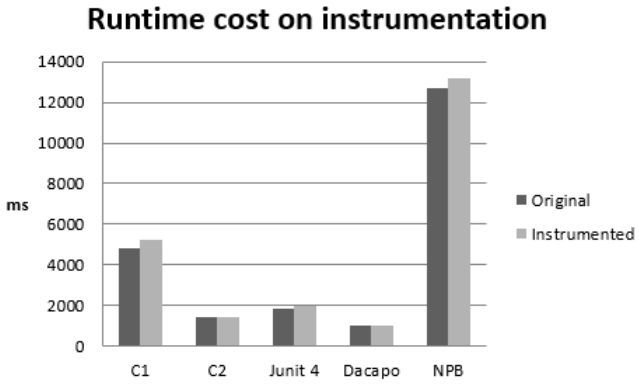


Figure 8. The running costs

4.3.3 Storage cost

After the instrumented program gets executed, the profile is exported to disk. Data in Table 6 show the size of the original profile, the size of the file after compressed by edge encoding, number of executed TCDG edges, and how many times those edges are executed in total.

Table 6. The storage costs.

Benchmark projects	Profile size (Byte)	Compressed profile size (Byte)	Number of thread controls edges executed	How many times the thread controls are executed in total
C1	1,878,897	117,560	964	17,267
C2	23,496	4,881	41	202
Junit4	2,779,880	1,058,975	10,660	28,612
Dacapo	626	319	3	6
NPB	15,310,849	1,238,510	214	168,497

4.3.4 More investigations of the overhead

From the overhead results, it is interesting to know their relationships, so that we can find what leads to such costs. Results of such investigation may contribute the usage of TCP in real-world applications.

1. Instrumentation cost. Theoretically, the instrumentation is performed at the thread control statements. Data in Table 5 clearly show that instrumentation cost increases with more thread controls.
2. Running cost. The running cost is caused by extra execution work on instrumented probes. Thus, we want to know if the running cost value has a proportional relationship to how many times thread controls execute. Results our investigation are shown in Figure 9 and show such proportional relationship. As the compiler may give different optimization on the instrumented program and change the runtime, the proportional relationship between the increased running time and how many times the thread controls execute is not a perfect proportional relationship.
3. Storage cost. To investigate the relationship between storage costs and the average execution frequency of thread controls, we calculated *compression ratio* and *average execution frequency of thread controls*. To calculate *compression ratio*, we use the values of *profile size* in Table 6 to divide the corresponding *compressed profile size* values. To calculate *average frequency of thread control execution*, we use the number of *thread controls edges executed* to divide the total times that thread controls execute. Results of *compression ratio* and *average execution frequency of thread controls* are shown in Figure 10, and illustrate that the bigger the average frequency is, the smaller the compression ratio is (i.e., the more efficient the compression is).

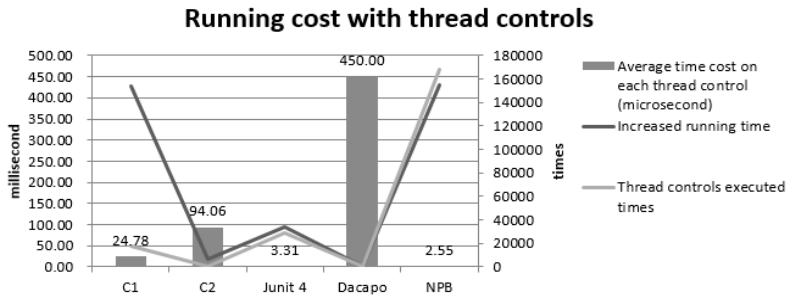


Figure 9. Running costs' relevance to thread controls' execution

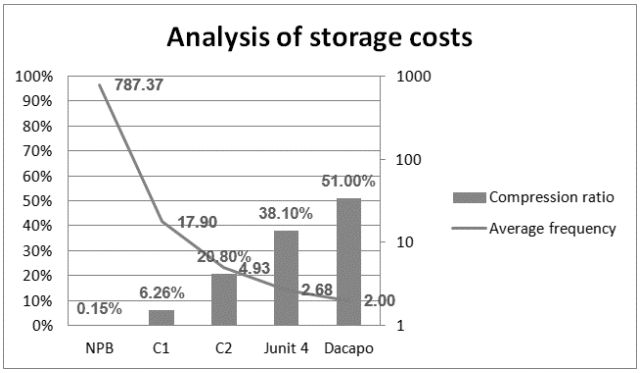


Figure 10. Storage costs' relevance to average execution frequency of thread controls

Furthermore, we observe the profile files to see how the thread controls distribute in program execution. We arrange the executed thread control edges in descending order of frequency. One accumulation is shown in Figure 11, which uses the frequency on thread control edges distinguished by different running threads (even if they exist in the same source code line). Another accumulation is shown Figure 12, which uses the frequency on thread control statements (Does no matter whether they are in the same running thread or not).

The *Gini Coefficients* on the benchmark projects are calculated to discover how unevenly the dependences get executed by the following steps:

1. The execution frequency of each profiled element is listed in descending order.
2. An accumulation is performed on the frequencies.
3. A line chart is used to show how the accumulation grows with more edges, and the *Gini Coefficient*[8] is calculated as well.

Results of *Gini Coefficients* in Figure 11 are 0.93 (C1), 0.72 (C2), 0.49 (Junit4), 0 (Dacapo), and 0.80 (NPB). Results of *Gini Coefficients* in Figure 12 are 0.97 (C1), 0.77 (C2), 0.84 (Junit4), 0 (Dacapo), and 0.77 (NPB). So, in most cases, there are some hot dynamic thread controls or some hot static thread-interact statements that cover the most execution, i.e., the executing frequency is not evenly distributed.

[8] *Gini Coefficient* is calculated by: $1 - {(2 * \sum_{i=1}^{n} x_i - x_n)}/{(n * x_n)}$, where $x_i = \sum_{j=1}^{i} A_j$ with $\{A_1, A_2, \ldots, A_n\}$ as the frequency value ascending from small to large.

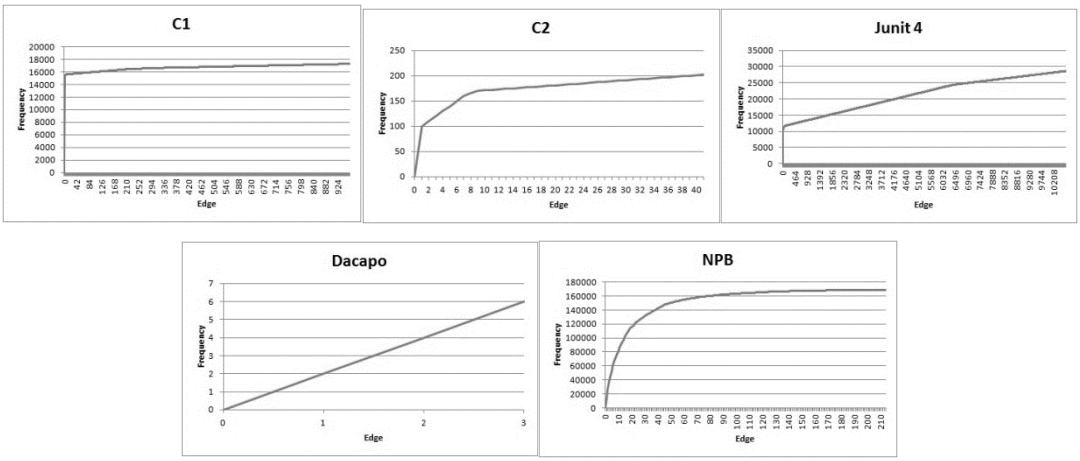


Figure 11. The accumulation of execution frequency on thread control edges (distinguished by dynamic threads)

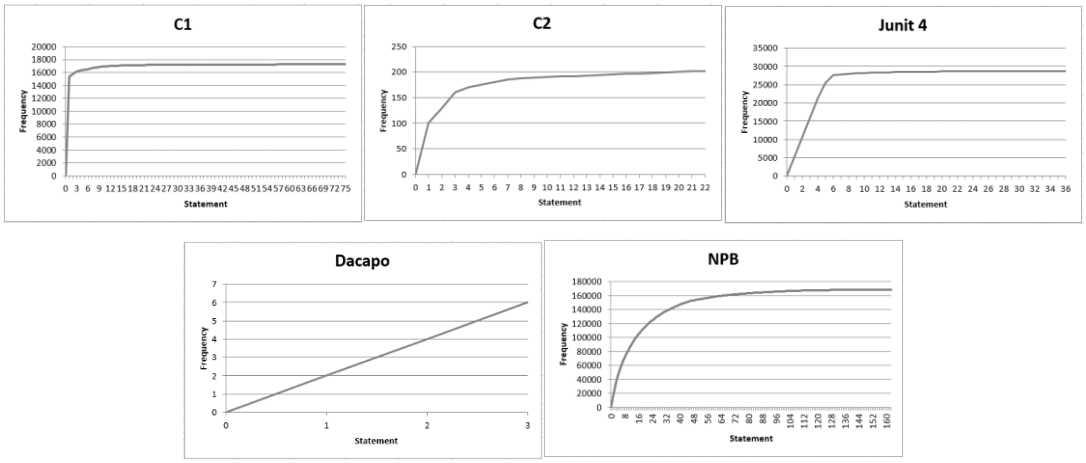


Figure 12. The accumulation of execution frequency on thread control statements (distinguished by static code lines)

Based on the above analysis, we get more insights of the overhead introduced by our TCP method. Although the overhead of TCP varies over different benchmark projects, in general, the overhead still has specific relations to the factors of the source code or the execution. The three types of overhead introduced by TCP follow different rules. The **instrumentation cost** grows if the number of **thread control dependences** increases**.** The **running cost** grows if **thread control dependences** execute more. The **uncompressed profile** is in direct proportion to the **execution time of thread controls.** Storage cost can be saved by compression with high **execution frequency of thread control dependences.** As shown in Figure 11, the profile can briefly reveal the inequality of different thread control dependences in execution.

4.4 Threats to validity

In this Section, we will discuss the most important internal validity and external validity threats of our TCP evaluations.

4.4.1 Internal validity

There are following possible threats to internal validity:

1. On the conclusion to RQ1 given by the experiment in Section 4.1:
   a) The defined mutants only produce exceptions or interrupts. In real applications, bugs may cause various errors which cause different behaviors in different situation.
   b) We only use the built-in test cases to run with the injected Junit4. It may perform differently with real testing where Junit is used as a testing tool.
2. On the conclusion to RQ2 given by the experiment in Section 4.2:

a) The design of questions Q1-Q20. Each question only tests if one correctly understands a certain part of source code and a certain aspect of program functions. The questions cannot cover the whole benchmark.
b) Difference in difficulties of Q1-Q20. Ideally, one should answer the same question with/without TCP information independently, so that we can accurately find the effect of TCP. But that is hard to realize. The difficulties of Q1-Q20 may be different, which has impacts on the experimental results.

3. On the conclusion to RQ3 given by the experiment in Section 4.3:
   a) Biased selection of the benchmark projects. The first step of our experiments to answer RQ3 is to select benchmark projects. To enable source code instrumentation, our evaluation requires the benchmark project to be source-code accessible, to be well compiled locally, to be executable after instrumentation, to be in an easy invocation form, and can run with thread controls. Very few programs can satisfy all the conditions[9]. Such restrictions of the benchmark projects may make our selected projects not representative.
   b) Accuracy of instrumentation. The instrumentation is based on JDT on the platform of Eclipse, which is supposed to have high reliability. To minimize the possible threat caused by bugs in JDT, we have manually checked JDT's correctness by running some tests.
   c) Experimental noise. The metrics applied to answer RQ3 involve program execution, which may be affected by environmental factors, and therefore contain noise. We try to reduce the possible noise by running the programs repeatedly, and use only the average values.

4.4.2 External validity

We identified three possible threats to external validity:

1. Representation of participants. In program understanding of our experiment, the participants are only junior programmers. Their understanding ability may be different from the software engineers. That causes risks of using TCP in real-world applications.
2. Representation of selected subjects. Our benchmark projects of RQ3 cover different common types of thread control dependences and have various program scales. Although the benchmark projects may not be representative, we believe that our observations of the overhead introduced by the TCP methods from RQ3 are still somewhat generalizable. Although, our evaluations of RQ1 and RQ2 are limited to Junit4, the observations from the case study of RQ1 and the experiment of RQ2 can still give evidence of the benefits of using our TCP method.
3. Programming languages. Our current TCP method is language-specific and is applicable only for programs in JDK 1.7. The TCDG constructed is backward compatible for programs in old JDK versions. For other languages, our profiling method and our categories of thread control dependences may need to be adapted.

5. Related Works

Program profiling covers rich contents. This section reviews three research fields related to TCP, namely profiling techniques which generate profiles, the usages of profiles for program benefits, and the concurrency analyses that are related to thread control dependences.

5.1 Profiling techniques

Different from tracing techniques which collect the entire program execution history, profiling techniques focus on dynamic frequency, and turn to efficiently answer what are the hottest program elements in iterative execution. Originally, there is edge profiling [1] that identifies the hottest edge in control flow and can directly be used in compiler optimization.

Afterwards, path profiling becomes an important research topic. Path profiling tries to calculate frequency on running paths, either acyclic paths [2, 10] or cyclic paths [11, 12], intra-procedural [2] or inter-procedural [13, 14] paths, universal [12] or particular paths [15-20], and so on. The path profiling has time cost about 30% on average [2]. Comparatively, the instrumentation overhead introduced by TCP is less than

[9] We searched and tried 60 Github projects and only got C1 and C2 that are usable.

8.9% in our evaluation experiments, which should be reasonable.

Furthermore, some researchers focus on other factors to be profiled, such as execution time [21], relevance in statements [22], resources [3], program performance [4, 23, 24], or system level metrics [25].

So far, a variety of programs have been profiled, including C++ programs [26], Java programs [27], GPU programs [28], dataflow programs (such as stream and media processing applications) [29, 30], and android applications [31].

5.2 Profile usages

With various information that profiles provide, a wide range of applications are invented and applied. Traditionally, the profiles are used in compiler optimization, program comprehension, testing, and so on [32]. We can classify recent research on profile usages into three categories, namely bug detection, performance optimization, and dynamic analysis.

Program profiles can provide the dynamic data of program behavior, and thus can be used to observe the abnormal running, especially when bugs mislead the control flows in execution. Such a series of profile usage mainly includes detect anomaly in running [33], improve the debugging accuracy [34], and testing efficiency [35].

With the information on execution frequency, it is possible to trade off the performance of less frequently executed paths in favor of more frequently executed paths. In this way, profiles are used to guide which execution is hot and should be first guaranteed for high efficiency [36-38].

In addition, profiles can also help various analysis techniques of program execution, such as object lazy allocation [39], dynamic impact analysis [40], global scheduling [41], and even exploit detection [42].

5.3 Concurrency analysis and understanding

The program concurrency has been a hot topic for decades, and thread dependence analysis is an important part of it.

Thread dependence model. The thread dependence graph is already presented for slicing concurrent programs. Such a model could be on either universal programs or specific ones, while the former is more unspecified and the latter has a more complete solution. One study gave a dependence model on universal concurrent programs [43], and another study focused on Java programs [44]. Compared to existing models, our work targets the new Java version, and the TCDG model is expected to be directly constructed from actual programs.

Dependence visualization. Visualization has been a good way to help manual debugging with an intuitive view. A tool named *Atropos* provides a visualization of dynamic dependences in executed concurrent programs, and helps trace the events in execution [45]. With a proper way of visualization, TCP results may be more convenient to be used.

Thread data dependences. Thread Level Speculation (TLS) is a dynamic parallelization technique that depends on out-of-order execution to achieve speedup on multi-processors. The dependences between threads make it is difficult to perform TLS. As mentioned in Section 2, thread dependences consist of two types, control dependences and data dependences, which interact with each other. Cross-thread data dependences have already proved very useful in TLS [46]. Recently, profiling of data dependences is also presented for better TLS performance [47]. Hopefully, profiling of control dependences could contribute to this field.

Understanding concurrent programs. Researchers usually manage to extract specific information from the concurrent program to help code understanding, mainly using graphs and models [66-68]. Some of them analyze the data dependences in concurrency [63, 69]. Furthermore, the code understanding is used to solve problems in refactoring and performance optimization [60, 70]. To the best of our knowledge, there has no technique that aims at thread control dependences yet.

6. Conclusion and future work

The concurrent programs are generally more difficult to develop, understand, and test than single-thread

program. To facilitate testing and understanding of concurrent program, we have invented a TCP method targeted at Java programs. First, we analyze the thread control dependences in Java programs and categorize them into five types. Second, we describe the features of each dependence type to automatically acquire them from source code and use them to construct the TCDG graph. Third, we collect the profiling solution using instrumentation.

To evaluate our TCP method, we evaluated how well it can help prioritize regression test cases of concurrent programs, how well it can help developers understand the concurrent code, and how much overhead it introduces to the original code. The evaluation results show that our TCP brings benefits to testing and code understanding of concurrent programs without bring too much overhead. In addition, we observe that that very few thread controls cover most code execution. The profiles from our TCP method can potentially also be used in:

- Facilitate debugging. TCP shows the dynamic behavior of threads. So, TCP may help debuggers detect which execution is abnormal and how it happens.
- Increasing test coverage. It is very hard to cover all possible thread interactions in testing. So, it is very useful to measure how the interactions get covered using the TCP method. Then, we can add more test cases to test the hot thread interactions more thoroughly.
- Providing quantified dynamic impact analysis. Dynamic analysis techniques are mainly qualified to show what is affected by software evolution in a range of execution. Some impacts are common while some are rare or even impossible at all. With the profile generated from TCP, the impact analysis can be quantified to be more accurate.

We have identified several research challenges that can be focuses of our future work.

- Besides profiling scattered dependences, it is more complicated and useful to profile thread-control-dependence paths, which is more accurate and informative to reveal how the control chains work in program execution.
- The data dependences between threads should also be studied. By combing data and control dependency, we can get more information with complete thread dependences.
- Programs in languages other than Java are also very popular and important. So, it will be valuable to perform dependence profiling on other program languages, especially with those ones that have common concurrency mechanisms as Java.

Acknowledgement

This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1003901, in part by the National Natural Science Foundation of China under Grant 61402103, Grant 61872078, and Grant 61572126, and in part by the Cooperation Project with Huawei Technologies Co., Ltd., under Grant YBN2016020009.

Reference

1. Ball T, Larus J R. Optimally profiling and tracing programs[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, 16(4): 1319-1360.
2. Ball T, Larus J R. Efficient path profiling[C]//Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, 1996: 46-57.
3. Finkler U. An analytic framework for detailed resource profiling in large and parallel programs and its application for memory use[J]. IEEE Transactions on Computers, 2010, 59(3): 358-370.
4. Mi H, Wang H, Cai H, et al. P-tracer: Path-based performance profiling in cloud computing systems[C]//Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual. IEEE, 2012: 509-514.
5. https://en.wikipedia.org/wiki/Java_(programming_language)
6. Ball T, Horwitz S. Slicing programs with arbitrary control-flow[C]//International Workshop on Automated and Algorithmic Debugging. Springer Berlin Heidelberg, 1993: 206-222.
7. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html
8. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html
9. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html
10. Afraz M, Saha D, Kanade A. P3: Partitioned path profiling[C]//Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015: 485-495.

11. Roy S, Srikant Y N. Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm[C]//Code Generation and Optimization, 2009. CGO 2009. International Symposium on. IEEE, 2009: 70-80.
12. Li B, Wang L, Leung H, et al. Profiling all paths: A new profiling technique for both cyclic and acyclic paths[J]. Journal of Systems and Software, 2012, 85(7): 1558-1576.
13. Melski D, Reps T. Interprocedural path profiling[C]//International Conference on Compiler Construction. Springer Berlin Heidelberg, 1999: 47-62.
14. Tallam S, Zhang X, Gupta R. Extending path profiling across loop backedges and procedure boundaries[C]//Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. IEEE Computer Society, 2004: 251.
15. Apiwattanapong T, Harrold M J. Selective path profiling[C]//ACM SIGSOFT Software Engineering Notes. ACM, 2002, 28(1): 35-42.
16. Joshi R, Bond M D, Zilles C. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems[C]//Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004: 239-250.
17. Bond M D, McKinley K S. Practical path profiling for dynamic optimizers[C]//Proceedings of the international symposium on Code generation and optimization. IEEE Computer Society, 2005: 205-216.
18. Zhu W, Bridges P G, Maccabe A B. Online critical path profiling for parallel applications[C]//Cluster Computing, 2005. IEEE International. IEEE, 2005: 1-9.
19. Vaswani K, Nori A V, Chilimbi T M. Preferential path profiling: compactly numbering interesting paths[C]//ACM Sigplan Notices. ACM, 2007, 42(1): 351-362.
20. Li B X, Wang L L, Leung H. Profiling selected paths with loops[J]. Science China Information Sciences, 2014, 57(7): 1-15.
21. Perelman E, Chilimbi T, Calder B. Variational path profiling[C]//Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on. IEEE, 2005: 7-16.
22. Baswana S, Roy S, Chouhan R. Pertinent path profiling: Tracking interactions among relevant statements[C]//Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE Computer Society, 2013: 1-12.
23. Ferrandi F, Lattuada M, Pilato C, et al. Performance estimation for task graphs combining sequential path profiling and control dependence regions[C]//Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign. IEEE Press, 2009: 131-140.
24. Roggio R F. Performance resource profiling using a computer-based number discrimination test system[C]//Computer-Based Medical Systems, 1991. Proceedings of the Fourth Annual IEEE Symposium. IEEE, 1991: 36-43.
25. Marathe A, Gahvari H, Yeom J S, et al. LibPowerMon: A Lightweight Profiling Framework to Profile Program Context and System-Level Metrics[C]//Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016: 1132-1141.
26. Schweitzer P, Mazel C, Hill D R C, et al. Inputs of aspect oriented programming for the profiling of C++ parallel applications on manycore platforms[C]//High Performance Computing & Simulation (HPCS), 2014 International Conference on. IEEE, 2014: 793-802.
27. Pandya U, Kent K B, Aubanel E, et al. A profiling tool for exploiting use of packed objects in Java programs[C]//Electrical and Computer Engineering (CCECE), 2016 IEEE Canadian Conference on. IEEE, 2016: 1-6.
28. Zheng M, Ravi V T, Ma W, et al. Gmprof: A low-overhead, fine-grained profiling approach for gpu programs[C]//High Performance Computing (HiPC), 2012 19th International Conference on. IEEE, 2012: 1-10.
29. Brunet S C, Mattavelli M, Janneck J W. Profiling of dataflow programs using post mortem causation traces[C]//Signal Processing Systems (SiPS), 2012 IEEE Workshop on. IEEE, 2012: 220-225.
30. Janneck J W, Miller I D, Parlour D B. Profiling dataflow programs[C]//Multimedia and Expo, 2008 IEEE International Conference on. IEEE, 2008: 1065-1068.
31. Wang Y, Rountev A. Profiling the responsiveness of Android applications via automated resource amplification[C]//Proceedings of the International Workshop on Mobile Software Engineering and Systems. ACM, 2016: 48-58.
32. Ball T, Larus J R, Ball T, et al. Programs follow paths[J]. 1999.
33. Dong W, Luo L, Chen C, et al. Post-Deployment Anomaly Detection and Diagnosis in Networked Embedded Systems by Program Profiling and Symptom Mining[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(12): 3588-3601.
34. Chilimbi T M, Liblit B, Mehra K, et al. HOLMES: Effective statistical debugging via efficient path profiling[C]//Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE,

2009: 34-44.
35. Hu Y, Yan J, Zhang J, et al. Profile directed systematic testing of concurrent programs[C]//Proceedings of the 8th International Workshop on Automation of Software Test. IEEE Press, 2013: 47-52.
36. Gupta R, Benson D A, Fang J Z. Path profile guided partial dead code elimination using predication[C]//Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on. IEEE, 1997: 102-113.
37. Gupta R, Berson D A, Fang J Z. Path profile guided partial redundancy elimination using speculation[C]//Computer Languages, 1998. Proceedings. 1998 International Conference on. IEEE, 1998: 230-239.
38. Tallent N R, Adhianto L, Mellor-Crummey J M. Scalable identification of load imbalance in parallel executions using call path profiles[C]//Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010: 1-11.
39. Shi J, Ji W, Zhang L, et al. Profiling and analysis of object lazy allocation in Java programs[C]//Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2016 17th IEEE/ACIS International Conference on. IEEE, 2016: 591-596.
40. Law J, Rothermel G. Path profile-based dynamic impact analysis[C]//Software Maintenance, 2002. Proceedings. International Conference on. IEEE, 2002: 262-262.
41. Young C, Smith M D. Better global scheduling using path profiles[C]//Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society Press, 1998: 115-123.
42. Stergiopoulos G, Petsanas P, Katsaros P, et al. Automated exploit detection using path profiling: The disposition should matter, not the position[C]//e-Business and Telecommunications (ICETE), 2015 12th International Joint Conference on. IEEE, 2015, 4: 100-111.
43. Zhao J, Cheng J, Ushijima K. A program dependence model for concurrent logic programs and its applications[C]//Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01). IEEE Computer Society, 2001: 672.
44. Zhao J. Multithreaded Dependence Graphs for Concurrent Java Programs[C]//pdse. 1999: 13-23.
45. Lönnberg J, Ben-Ari M, Malmi L. Visualising concurrent programs with dynamic dependence graphs[C]//Visualizing software for understanding and analysis (VISSOFT), 2011 6th IEEE international workshop on. IEEE, 2011: 1-4.
46. Colohan C B, Ailamaki A, Steffan J G, et al. Tolerating dependences between large speculative threads via sub-threads[C]//ACM SIGARCH Computer Architecture News. IEEE Computer Society, 2006, 34(2): 216-226.
47. Bhattacharyya A. Do inputs matter? Using data-dependence profiling to evaluate thread level speculation in BG/Q[C]//Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on. IEEE, 2013: 401-401.
48. Hammacher C, Streit K, Hack S, et al. Profiling Java programs for parallelism[C]// ICSE Workshop on Multicore Software Engineering. IEEE Computer Society, 2009:49-55.
49. Lönnberg J, Ben-Ari M, Malmi L. Visualising concurrent programs with dynamic dependence graphs[C]// IEEE International Workshop on Visualizing Software for Understanding and Analysis. IEEE, 2011:1-4.
50. Chen Z, Xu B. Slicing concurrent java programs[M]. ACM, 2001.
51. Singh J, Munjal D, Mohapatra D P. Context Sensitive Dynamic Slicing of Concurrent Aspect-Oriented Programs[C]// Software Engineering Conference. IEEE, 2015:167-174.
52. Malony A D, Huck K. General Hybrid Parallel Profiling[C]// Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE Computer Society, 2014:204-212.
53. Heil T, Smith J E. Relational profiling: enabling thread-level parallelism in virtual machines[C]// ACM/IEEE International Symposium on Microarchitecture. ACM, 2000:281-290.
54. Szebenyi, Z, Wolf, F, Wylie, B.J.N. Space-efficient time-series call-path profiling of parallel applications[C]// High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on. IEEE, 2009:1-12.
55. Malony A D, Shende S, Morris A. Phase-Based Parallel Performance Profiling[C]// Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain. DBLP, 2006:203-210.
56. Han Z, Qu G, Burkard D, et al. A Memory Access Pattern-Based Program Profiling System for Dynamic Parallelism Prediction[C]// Trustcom/bigdatase/ispa. IEEE, 2017:1448-1454.
57. Qadeer S, Rehof J. Context-Bounded Model Checking of Concurrent Software[C]// International

Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2005:93-107.
58. Zhao J, Cheng J, Ushijima K. Program dependence analysis of concurrent logic programs and its applications[C]// International Conference on Parallel and Distributed Systems. IEEE Computer Society, 1996:282.
59. Shao Z, Peng J, Jin H. Data Race Detection by Understanding Synchronization Relationships of Thread Segments[C]// Euromicro International Conference on Parallel, Distributed and Network-Based Processing. IEEE, 2017:229-232.
60. Pinto G, Canino A, Castor F, et al. Understanding and Overcoming Parallelism Bottlenecks in ForkJoin Applications[C]// International Conference on Automated Software Engineering. 2017.
61. Hirayama M, Yamamoto T, Okayasu J, et al. A Selective Software Testing Method Based on Priorities Assigned to Functional Modules[C]// Quality Software, 2001. Proceedings. Second Asia-Pacific Conference on. IEEE, 2001:259-267.
62. Manson J, Pugh W, Adve S V. The Java memory model[C]// Acm Sigplan-sigact Symposium on Principles of Programming Languages. ACM, 2005:378-391.
63. Ta T, Troendle D, Hu X, Jang B. Understanding the Impact of Fine-Grained Data Sharing and Thread Communication on Heterogeneous Workload Development[C]// International Symposium on Parallel and Distributed Computing (ISPDC). IEEE Computer Society, 2017:132 – 139.
64. Lohr K P, Vratislavsky A. Jan - Java animation for program understanding[C]// IEEE Symposium on Human Centric Computing Languages and Environments. IEEE Computer Society, 2003:67-75.
65. Pinto G, Torres W, Fernandes B, et al. A large-scale study on the usage of Java's concurrent programming constructs[J]. Journal of Systems & Software, 2015, 106(C):59-81.
66. Zernick D, Snir M, Malki D. Using Visualization Tools to Understand Concurrency[J]. IEEE Software, 1992, 9(3):87-92.
67. Younger E J, Ward M P. Understanding concurrent programs using program transformations[C]// Program Comprehension, 1993. Proceedings. IEEE Second Workshop on. IEEE, 1993:160-168.
68. Xie S, Kraemer E, Stirewalt R E K. Design and Evaluation of a Diagrammatic Notation to Aid in the Understanding of Concurrency Concepts[C]// International Conference on Software Engineering. IEEE, 2007:727-731.
69. Milanova A, Liu Y. Static Analysis for Understanding Shared Objects in Open Concurrent Java Programs[J]. 2010:45-54.
70. Okur S. Understanding, Refactoring, and Fixing Concurrency in C#[C]// Ieee/acm International Conference on Automated Software Engineering. IEEE Computer Society, 2015:898-901.

Appendix: Q1-Q20 of experiment in Section 4.2

We randomly choose 16 test cases of Junit4:

No.	Test cases	Class
1	testShadowedTests	junit.tests.framework.SuiteTest
2	testAssertEqualsNull	junit.tests.framework.AssertTest
3	testActiveRepeatedTest1	junit.tests.extensions.ActiveTestTest
4	testAssertNaNEqualsNaN	junit.tests.framework.FloatAssertTest
5	testComparisonErrorOverlappingMatches Context	junit.tests.framework.ComparisonCompactorTest
6	testRunFailureResultCanBeSerialised	junit.tests.runner.ResultTest
7	testRunSuccessResultCanBeSerialised	junit.tests.runner.ResultTest
8	testComparisonErrorStartSameComplete	junit.tests.framework.ComparisonCompactorTest
9	testError	junit.tests.runner.TextRunnerTest
10	testError	junit.tests.framework.TestCaseTest
11	testOneTest	junit.tests.runner.TextFeedbackTest
12	testActiveTest	junit.tests.extensions.ActiveTestTest
13	testThrowing	junit.tests.framework.ComparisonFailureTest
14	testActiveRepeatedTest0	junit.tests.extensions.ActiveTestTest
15	testNotExistingTestCase	junit.tests.framework.SuiteTest
16	testActiveRepeatedTest	junit.tests.extensions.ActiveTestTest

Based on such test cases, Q1~Q20 are designed as following: three test cases are chosen from the 16 ones, and a question asks the participants to choose the test case which runs most different source code from the other two test cases.

For example, we use No.3, No.14 and No.16 in Q20:

Q20. Choose the one that executes most different source code from the other two:

A. testActiveRepeatedTest1 (from *junit.tests.extensions.ActiveTestTest*)
B. testActiveRepeatedTest0 (from *junit.tests.extensions.ActiveTestTest*)
C. testActiveRepeatedTest (from *junit.tests.extensions.ActiveTestTest*)

All Q1-Q20 are listed as following:

Qustion	Option (A)	Option (B)	Option (C)	Correct Answer
Q1	No.1	No.6	No.7	A
Q2	No.2	No.12	No.16	A
Q3	No.3	No.14	No.15	A
Q4	No.3	No.4	No.8	A
Q5	No.4	No.12	No.16	A
Q6	No.3	No.4	No.16	B
Q7	No.3	No.5	No.12	B
Q8	No.1	No.2	No.6	C
Q9	No.2	No.5	No.6	C
Q10	No.7	No.11	No.13	A
Q11	No.7	No.10	No.13	A
Q12	No.3	No.6	No.12	C
Q13	No.9	No.12	No.16	A
Q14	No.3	No.11	No.16	B
Q15	No.1	No.11	No.12	C
Q16	No.4	No.12	No.15	B
Q17	No.2	No.10	No.12	C
Q18	No.1	No.15	No.16	C
Q19	No.4	No.11	No.16	C
Q20	No.3	No.14	No.16	B