Are Java Programmers Transitioning to Multicore? A Large Scale Study of Java FLOSS

Weslley Torres Gustavo Pinto Benito Fernandes
João Paulo Oliveira Filipe Ximenes Fernando Castor
Informatics Center, Federal University of Pernambuco, Recife, Brazil
{wst, ghlp, jbfan, jpso, fax, castor}@cin.ufpe.br

Abstract

We would like to know if Java developers are retrofitting applications to become concurrent and, most importantly, to get a better performance on multicore machines. Also, we would like to know what concurrent programming constructs they currently use. Evidence of how programmers write concurrent programs can aid others programmers to be more efficient when using the available constructs. Moreover, it can assist researchers in devising new mechanisms and improving existing ones. For this purpose, we have conducted a study targeting a large-scale Java open source repository, SourceForge. We have analyzed a number of FLOSS projects along two dimensions: spatial and temporal. For the first one, we have studied the latest versions of more than 2000 projects. Our goal is to understand which constructs developers of concurrent systems employ and how frequently they use them. For the temporal dimensions we took a closer look at various versions of six projects and analyzed how the use of concurrency constructs has evolved along time. In addition, we try to establish whether uses of concurrency control constructs aimed to leverage multicore processors or not. We downloaded more than two thousand Java projects including their various versions, besides individual analys about six well known open-sources projects.

Keywords Java, Open-Source, Concurrent, Parallel, Multicore

1. Introduction

In order to get real performance advantages of multicore machines, programmers need to build parallel applications [24]. However, building this kind of application is a demanding and error-prone task [9, 17, 24]. Many programming languages, e.g., Go, Scala, Java, Erlang, C#, and Lua, implement their own constructs for concurrent/parallel programming.

Considering the discrepancies among the many existing approaches for concurrent programming, we would like to know how programmers use them, in terms of frequency of use, the system evolution over time, and if programs are becoming more concurrent along their versions. More generally, we would like to know what programming constructs developers actually use to build concurrent systems, especially if programmers are aware about evolution/transition of singlecore to muilt-core.

On the one hand, knowing how commonly programmers use these constructs may help researchers to design new mechanisms or improve existing ones, based on development practice. In addition, it can point out the real needs of developers, not only in terms of new or improved mechanisms, but in terms of refactoring and reengineering tools and techniques that can help them to incorporate these mechanisms into existing systems.

On the other hand, developer awareness about these usage patterns might lead to more efficient use of existing abstractions. Finally, for both researchers and developers, it is important to understand trends in software engineering and only a empirical study can gather that kind of information.

In this work we present an empirical study targeting a large-scale Java open source repository. We try to answer two research questions by examining a large body of real-world experimental data. Our main goal is to answer these research questions:

- RQ1 - How often are the Java concurrency constructs employed in real applications?
- RQ2 - Are programmers aware about the transition from singlecore to multicore?

We obtained the source code of 2283 Java projects from SourceForge and peform an automatic analysis, collecting more than 50 different metrics related to concurrency from these projects, six others well known open source projects (Apache Tomcat, Lucene, Cassandra, Subversion, Hibernate and jMonkeyEngine) were analyzed by manual checking. These projects comprise approximately 560 million lines of source code spread throughout more than 15,000 versions. We have chosen the Java language because it is a widely used object-oriented programming language. Moreover, it includes support for multithreading with both low-level and high-level mechanisms.

Mining data from the SourceForge repository poses several challenges. Some of them are inherent to the process of obtaining reliable data. These derive from mainly two factors: scale and lack of a standard organization for source code repositories. Others pertain to actually transforming the data into useful information. Grechanik et al. [11] discuss a few challenges that make it difficult to obtain evidence from source code: for example, getting the source code of all software versions is difficult because there is no naming pattern to define if a compressed file contains source code, binary code or something else. Furthermore, it is difficult to verify that an error has occurred during measurement, due to the number of projects and project versions. We addressed these challenges by creating an infrastructure for obtaining and processing large code bases, specifically targeting SourceForge. Overall, we found

TMC '11 October 2011, Portland, USA

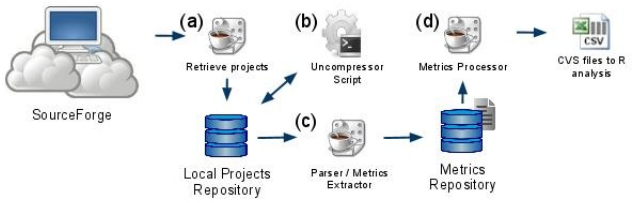

Figure 1. High-level view to our infraestrucutre.

out that most of the medium to large-sized projects employ some form of concurrency control. Most of them use mainly mutual exclusion in the form of synchronized blocks and methods. A surprisingly large amount (more than 25% of all the projects, 50% of the concurrent ones) employ monitor-based synchronization, although most of them use it sparsely. Finally, we discovered that developers are wasting many opportunities to use these higher level/more efficiente abstractions.

2. Study Setting

This section describes the configuration of our study: our basic assumptions, our mining infrastructure, the metrics suite that we employ, and our research questions.

2.1 Context

We analyzed mature and stable Java projects obtained from SourceForge. Due to the release date of Java 1.5 in late 2004, the first official release of the `java.util.concurrent` (j.u.c.) library, we only obtained projects whose latest version update was at least in 2005. We consider a program as concurrent if it extends `Thread` class or `Runnable` interface or implements `Runnable` interface at least once or employs any concurrency control mechanism, such as synchronized blocks or synchronized methods. Beyond projects from SourceForge we also analyzed some Apache projects, because they are known for their high quality implementations, which contrast with the heterogeneity of SourceForge.

To crawl projects in the repository we had to define some heuristics, for example, to get source code, the crawler searches for files whose names include keywords like 'source' or 'src'. At the end, we obtained 2097 main projects out of 9101 mature and stable Java projects. The project classification as mature or stable is defined by the project maintainers at SourceForge. We disregarded many projects to improve the reliability of our findings. Even then, we could analyze more than half a billion lines of code.

2.2 Infrastructure

Our infrastructure consists of three major crawlers, and one shell script (Figure 1). Initially, (a) the first crawler populates the project repository with Java Projects from Sorceforge, including their various versions. In (b) the shell script extracts all compressed files into our local repository. In (c) the crawler parsers the source code, collects metrics, and stores the results in the metrics repository. In (d) the crawler generates input, as CSV files, to be analyzed by R [14].

The crawlers are an extention of Crawler4j [1], an open source web crawler application, multithreaded and written in Java. We also implemented additional scripts to order project versions based on dated available at SourceForge and to check if the target project is ready to be analyzed, fixing its structure when necessary.

To collect concurrency metrics we used the `JavaCompiler` class[2] to parse the source code and build parse trees. The trees are traversed and the metrics are extracted and stored in text files. Metrics collected consist of counting numbers of lines, imports, class instantiations of the `Thread` class, method invocations, class extensions of the `Thread` class and the `Runnable` interface, implementations of the `Runnable` interface, and uses of some Java keywords such as synchronized and volatile. Some collected metrics are: numbers of extends `Thread`, implements `Runnable`, import j.u.c, sync methods, sync blocks, `Hashtable`, `HashMap`, `ConcurrentHashMap`, `AtomicInteger`, Lines of Code. The full list is available on the website [23].

3. Research Questions

3.1 How Often are the Java Concurrency Constructs Employed in Real Applications?

One of the goals of our study is to understand how concurrent code has evolved over time, in terms of the usage of concurrency constructs. Therefore, in an attempt to capture trends in terms of concurrent programming construct usage, we need to count basic statistics about the code, as mentioned in the previous section. These tasks are done by separating projects in groups sorting by year, or metrics like LOC, as we will show later.

3.2 Are Programmers Aware of Evolution/Transition from Singlecore to Multicore?

This is a very interesting question because, besides verifying the general results for RQ1 with the source code, we can also map and identify some characteristics about how programmers usually evolve code that requires concurrent skills, and if they are really moving to multicore.

On the other hand, these questions are extremely complex and wide, and this paper does not cover it entirely. Moreover, to begin this study we need to analyze each project individually and manually, looking for individual transformations of use, or disuse, of the most common contructs related to concurrency. This task is costly. Therefore, we studied six open-source projects that have one or more version from 2005 until today. These transformations were analyzed along three or four versions of each project.

To achieve this goal, we manually analyzed about three or four versions of six open-source Java projects: Tomcat [3], jMonkeyEngine [4], Lucene [5], Blackports [6], Mobicents [7] and Fura [8]. Some of this projects are very large, so we guided our analysis by searching in the source code for concurrency keywords and comparing the source code of differents versions.

Among this set of projects, Tomcat, jMonkeyEngine and Fura were individually selected because they are successful open-source projects. The reason we did this was to analyze projects that are mature and widely used, for both community and commercially. For the remaining three, we applied a random algorithm to choose the last three projects that we downloaded from source forge.

Tomcat: Apache Tomcat is a web container, or application server, enabling Java code to run in cooperation with a web server. Tomcat is the official Reference Implementation for the Java Servlet and the JavaServer Pages (JSP) specifications. Note that Tomcat represents a group of projects, here we consider only the 'Catalina' subproject, which implements the actual servlet container.

[1] http://code.google.com/p/crawler4j/

[2] http://download.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html

[3] http://tomcat.apache.org

[4] http://www.jmonkeyengine.com

[5] http://lucene.apache.org

[6] http://backport-jsr166.sourceforge.net/

[7] http://sourceforge.net/projects/mobicents

[8] http://fura.sourceforge.net

jMonkeyEngine: jMonkeyEngine (JME) is a game engine, made Especially for game developers who want to create 3D games with modern technology standards. The software is programmed in Java entirely, intended for wide accessibility and quick deployment.

Lucene: Lucene is a high-performance, full-featured text search engine library. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. It is supported by the Apache Software Foundation and is released under the Apache Software License.

Backports: The goal of this project is to provide a concurrency library that works with uncompromised performance on all Java platforms currently in use, allowing development of fully portable concurrent applications. More precisely, the target scope is Java 1.3 and above, and some limited support is offered for Java 1.2.

Mobicents: Mobicents is the leading Open Source VoIP Platform. It is the First and Only Open Source Certified implementation of JSLEE 1.1 (JSR 240), and SIP Servlets 1.1 (JSR 289). Mobicents also includes a powerful and extensible Media Server.

Fura: Fura is a self-contained grid middleware that allows the grid deployment and distribution of applications on heterogeneous computational resources. Fura's component based plug-in architecture allows grid services to be extended or replaced, and new services can be developed reusing existing components.

4. Study Results

This section presents the results of the measurement process. The data has been collected based on the set of defined metrics. The presentation is organized in two parts. Section 4.1 tries answer RQ1 and Section 4.2 tries to answer RQ2.

Initially, projects were divided into three categories, small projects (more than 999LOC and less than 20KLOC), medium projects (between 20KLOC and 100KLOC) and big projects (more than 100KLOC), some projects can be in more than one category because they can have one version with less than 20KLOC and another version with more than 20KLOC. Table 1 presents some general size metrics for the projects we have downloaded. This study analyzed 2103 project in total, but only 1523 are considered concurrent and only 364 use the j.u.c. library. The number of projects that use j.u.c. is lower than we expected, since we only got projects whose latest update occurred after the release of j.u.c. as part of the JDK. Moreover, this library had been available for general use for at least five years before it was incorporated into the JDK. The largest project we have analyzed is the Liferay Portal[9], with about 1.7 million LoC, followed by Rental Portal [10], with about 1.5 million LoC. The smallest project we analyzed is Gomoku [11], with exactly 1000 lines of Java code. Note that the concurrent projects are, on the average, considerably larger than the non-concurrent ones. This is expected: most complex projects involve concurrency at some level.

#Projecs	2.103
#Concurrent small projects	000
#Non concurrent small projects	000
#Concurrent medium projects	635
#Non concurrent medium projects	667
#Concurrent big projects	000
#Non concurrent big projects	000
#Concurrent projects that use java.util.concurrent	364
#Non concurrent projects	593
# of LoC (all versions of all small projects)	60.137.100
# of LoC (all versions of all medium projects)	212.677.935
# of LoC (all versions of all big projects)	286.429.352
Size on disk (all versions of all projects)	124GB

Table 1. General information about the projects.

4.1 How Often the Java Concurrency Constructs are Employed in Real Applications?

This section presents the results summarized in Table 3 for the basic java concurrency control mechanisms divided into categories according to size projects. Table 2 presents general results, like the number of implementations of `Runnable`, the number of classes extending `Thread` and the number of `Thread` methods invocations. We also count the number of `synchronized` blocks and methods. We collected the metrics for the concurrent projects, considering all the versions of each one. These results only account for projects whose value in each metric is at least 1. Otherwise, for some metrics, many of the results would be 0. To avoid confusion, the last column of the table also presents the number of projects whose value for the metric is greater than 0. These results are depicted below. The complete results of the study are available on the website [23].

Synchronized modifier. We broke the analysis for these constructs in two, based on its two forms. `Synchronized` blocks are present in 651 of 957 small projects, 512 of 582 medium projects and 184 of 191 big projects. The standard deviation for synchronized block is 21.61, 67.46 and 191.97 respectively. This indicates that there is a small number of projects that have a strong impact on the overall results. For example, there is a single project that uses `synchronized` blocks 1401 times. This is a recurring phenomenon. Most of the metrics have a standard deviation higher than the mean. Complementarily, `synchronized` methods are present in 88% of small projects, 96,21% of medium projects and 99,47% of big projects which indicates that almost all big projects use `synchronized` methods.

Thread and Runnable. We have collected two metrics pertaining to the `Thread` class: number of classes extending `Thread` and number of calls to `Thread` methods. We can see that 62,19% of the medium projects and 75,91% of the big projects extend `Thread`, small projects only 40% extend `Thread`. More than 88% of small projects invoke `Thread` methods and almost 100% of medium and big projects invoke `Thread`. We have also measured the number of classes that implement the `Runnable` interface. This is useful to discover which approach developers prefer, in order to create new threads. In accordance with our intuition, implementing `Runnable` is the most popular approach, with higher median, mean, and 3rd quartile.

Imports of j.u.c. We collected this metric by counting the number of `import` clauses for the j.u.c. package and its subpackages. Overall, 130 out of 957 small projects, 146 out of 582 medium projects and 85 out of 191 big projects are using the library. At the same time, the projects that do employ these solutions do so fairly frequently. If the number of imports is any indication, the 3rd quartile is higher than median of the # of synchronized blocks, except for big projects.

Atomic data types and concurrent collections. Contrary to our intuition, few projects employ atomic data types, 3.55% of small projects, 11,10% of medium projects and 23% of big projects. We assumed that these constructs would be more widespread due to their ease of use and great similarity with their non-thread-safe counterparts. On the other hand 6,47% of small projects, 15,29% of medium projects and 29,84% of big projects use concurrent collections.

[9] http://www.liferay.com

[10] http://sourceforge.net/projects/rentalportal/

[11] http://gomoku.sourceforge.net

metrics	Min.	1st Qua.	Median	Mean	3rd Qua.	Max.	Std. Dev.	#Projects
# synchronized blocks	1/1/1	2/5/14	5/15/54	11.53/41.48/123.7	12/44/156.5	325/482/1401	21.64/67.46/191.97	651/512/184
# synchronized methods	1/1/1	3/7.75/39	6/25/93	13.62/52.62/158.5	15/61/190	196/677/1058	20.54/75.55/191.94	846/560/190
# classes extending Thread	1/1/1	1/1/3	2/3/6	2.57/5.19/9.93	3/6.75/12	12/65/59	2.27/6.55/12.04	390/362/145
# uses of Thread methods	1/1/1	4/13/37	8/29/94	15.75/56.49/290.6	18/63/197	380/1186/23724	24.89/98.27/1739.48	851/571/187
# implementing Runnable	1/1/1	1/2/3	2/3/6	3.04/6.78/13.41	4/7/15	40/83/114	3.82/10.61/17.98	420/378/160

Table 2. Projects metricts by categories (small/medium/big projects, respectively), for basic Java concurrency control mechanisms, considering only concurrent projects. This table includes metrics for mutual exclusion based on synchronized blocks and basic use of threads.

metrics	Min.	1st Qua.	Median	Mean	3rd Qua.	Max.	Std. Dev.	#Projects
# imports of j.u.c	1/ 1/ 1	2/ 3/ 4	4/ 10/ 13	8.32/ 26.95/ 46.55	9/ 29.75/ 49	79/ 453/ 520	11.75/ 53.55/ 84.04	130/ 146/ 85
# Atomic data types	1/ 1/ 1	1/ 2/ 3	2/ 4/ 6	3.05/ 9.92/ 15.82	4/ 6/ 19.50	11/ 79/ 98	2.81/ 17.53/ 21.37	34/ 65/ 44
# Concurrent collections	1/ 1/ 1	1/ 2/ 2	2/ 4/ 4	3.17/ 7.57/ 13.02	4/ 7/ 14	14/ 72/ 117	3/ 11.37/ 21.40	62/ 89/ 57
# Locks	1/ 1/ 1	1/ 1/ 2	1/ 2.5/ 4	2.47/ 7.59/ 7.16	2/ 8/ 8.75	23/ 156/ 52	3.79/ 18.55/ 8.91	36/ 76/ 50
# Barriers	1/ 1/ 1	1/ 1/ 1	3/ 2/ 2	3.16/ 8.07/ 14.5	4.5/ 5/ 22.5	7/ 72/ 83	2.16/ 15.82/ 21.47	12/ 27/ 28
# Futures	1/ 1/ 1	1/ 1/ 1	2/ 2/ 2	2.85/ 3.5/ 3	4/ 4.25/ 4	11/ 12/ 10	2.53/ 3.32/ 2.32	20/ 36/ 24

Table 3. Projects metricts by categories (small/medium/big projects, respectively), for concurrency abstractions that the j.u.c. library implements.

4.2 Are developers transitioning to multicore?

This section presents the studied data from a temporal perspective. It is important to notice this question is extremely complex and part of the effort of this paper is to start to answer it therefore, we have analyzed a small number of systems to gather information that will provide us insight about the answer . We have broken this question into three more:

- During software evolution.
- Have threads been used for concurrency or parallelism?
- Are developers wasting opportunities to use j.u.c.?

4.2.1 The most common use/evolution of concurrent constructs.

Concurrent constructs are used in many different ways, although, most of the concurrency effort is to lock and to release resources. To that end, basic constructs like the `synchronized` keyword, can often be safely retrofitted to high level libraries, like j.u.c., providing more flexibility. Figure 2 synthesizes the code evolution for three projects (due page limits, others results are present only on website), comparing the use of synchronized.

We can observe many differences among the projects. JMonkeyEngine, for example, uses j.u.c. since its first analyzed version, but, only in version 2.1 the number of j.u.c had extraordinarily increased. At the same time, the use of `synchronized` did not decrease. A quick investigation of the source code reveals that about 40% of the j.u.c constructs are present in test case classes. Looking forward, this behavior is very common in open source projects, including three of the six projects that we have analyzed. The use of the `Executors` and `ExecutorService` during the test execution revealed a new behavior on software developers. It is interesting to mention this because, despite the large number of testing frameworks, programmers still prefer to use j.u.c. constructs to conduct some testing activities.

We can also observe that the Fura project showed a different pattern. The use of `synchronized` nearly doubled from one version to another. The same happened to the lines of code from the first to last version.

Finally, the last project reveals yet another pattern. The use of `synchronized` methods and blocks decreased and the use of j.u.c increased. In the Backport source code we can find and compare this fact. For example, it is very common to identify methods that, in an early version used the `synchronized` keyword and in the next one use the `Lock` and `ReentrantLock` classes. Backport is the project that takes more advantage of j.u.c. constructs, which can be verified comparing the number of uses of j.u.c per lines of code.

Moreover, analysing data structures, it is interesting to notice that the use of `HashTable` decreased along all the projects versions while `HashMap` and concurrent collections increased. This information leads us to believe those programmers are aware about the inefficiency of `HashTable` and have chosen other collections to increase application performance, which can be an indication that they are worried about the transition to multicore.

4.2.2 Threads for concurrency or threads for parallelism?

Since threads are a general purpose construct that can be use for anything, it is very difficult to understand with what intention would a programmer use that. Due this fact, we can sort threads into two initial groups: threads that handles I/O operations (like read/write operations, network input/output, database access, etc.), and threads that perform computationally expensive operation (like mathematical calculations, graphics rendering, search/sorting algorithms, etc).

The selection of these groups are related to the fact of applications that use threads to accomplish simultaneous operations are not necessarily related to parallelism. This is because threads can waste resources (eg. I/O) but, at the same time, do not use CPU. In turn, the second group are directly related to multicore transition. Thus, the main goal here is to compare the growth of the second group with regard to the first one.

In every project it is possible to visualize the existence of these groups. However, to map all occurrences requires more detailed work. To this initial report, we identified which projects have more threads for parallelism, which was done just following theses steps: i) For each project, seek thread constructs in source code by searching in Eclipse IDE; ii) Try to understand what that code does, and mark with concurrent or parallel; iii) Do the same with the others versions. As result, we consider Backports and Lucene as project that does more use of parallelism. Particulary, Lucene is a project that cares about parallelism since its first version that we analyzed. This is stated at block comments, method and class names, and so on. Figure 3 shows an example using the `ParallelTask` class.

What exactly did the study do to answer this question?

Tomcat is a project that takes much advantage of threads and concurrency in general, as expected (remember that we are only considering the subproject called Catalina). Nevertheless, its use of threads is weakly associated with parallelism, due the fact of

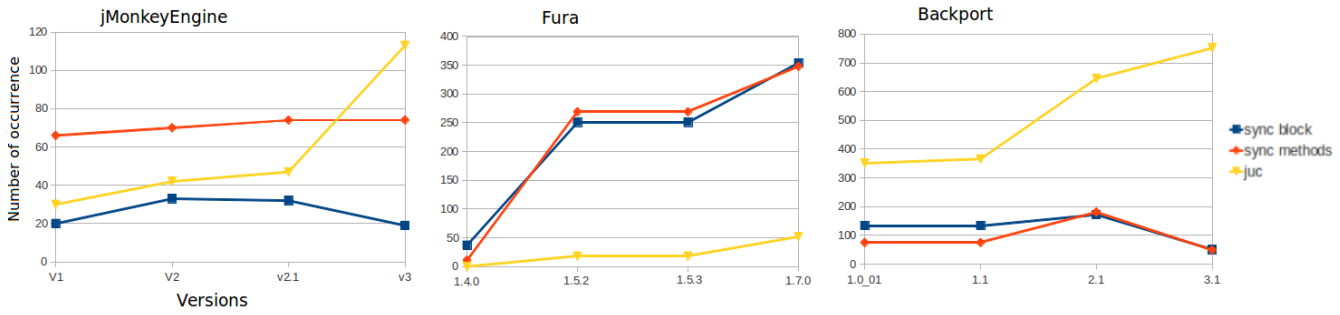

Figure 2. Synchronized use versus j.u.c. use

```
1   private class ParallelTask extends Thread {
2     @Override
3     public void run() { try {
4       int n = task.runAndMaybeStats(letChildReport);
5       if (anyExhaustibleTasks) {
6         updateExhausted(task); }
7       count += n;
8     } catch (NoMoreDataException e) {
9       exhausted = true; } catch (Exception e) {
10      throw new RuntimeException(e); } } }
```

Figure 3. An example of threads for parallelism in Lucene.

Tomcat is an application service, and much of its work is to handle HTTP requests and responses, or socket comunications.

Finally, we notice that, although Java programming language does not have any default construct to perform parallelism, threads have been successfully applied to this end, and as noticed in the investigated projects, it is not difficult to understant which case it was applied.

4.2.3 Are Developers Wasting Opportunities to use j.u.c.?

One of our questions is whether developers are using high-level libraries, like j.u.c, to make the transition to multicore. One of the basic assumptions of this study is that it is better to use high-level mechanisms than low-level ones. Besides abstraction, the former have more concrete advantages, such as the impossibility of deadlocks and some performance optimizations. In addition, they simplify the task of programming by promoting reuse of recurring solutions. Therefore, considering that only a limited number of projects use the j.u.c. library, it makes sense to ask whether developers are wasting opportunities to reap the benefits of this library. To answer this question, we randomly have chosen 100 projects out of all the 1523 concurrent projects. For each one, we have randomly collected 1–3 examples of the use of the `synchronized` keyword in these projects. For each example, we have analyzed the use of `synchronized` in that block or method to see if it would be feasible to replace it by concurrent collections or atomic data types. We have built simple tools to randomly select the projects and to randomly choose the `synchronized` blocks or methods. Due to this, we manually inspected the code snippets to check if the selected block or method could be replaced by the use of concurrent collections or atomic variables. Dig et al. [7] present a list of code templates that would be easily replaced by uses of atomic data types to remove the `synchronized` keyword. That list served as a basis for the manual inspection.

We analyzed 276 examples `synchronized` usage. Some systems had fewer than 3 occurrences of `synchronized` and, in these cases, we selected every one of them. We found 28 cases where the use of `synchronized` could be avoided in 25 projects. For these cases, statements within `synchronized` methods or blocks

```
1  public synchronized void atualiza(long tempoPassado) {
2    if (frames.size() > 1) { tempoAnim += tempoPassado;
3      if (tempoAnim >= tempoTotal) {
4        tempoAnim = tempoAnim % tempoTotal;
5        frameAtual = 0; }
6      while(tempoAnim>((Integer)(tempos.get(frameAtual))).intValue()) {
7        frameAtual++; } } }
```

Figure 4. Project: javagamelibrary Class : Animacao

```
1  ...
2  private static int s_lastRequest = 0;
3  ...
4  public static synchronized int getNextSequenceNumber() {
5    return s_lastRequest++; }
```

Figure 5. Project: Opensubsystems, Class :GlobalSequence

were very simple. Thus, the use of `synchronized` could be easily avoided with concurrent collections or atomic data types. It is noteworthy that 40% of these projects already use j.u.c. somehow. We noticed that, in most cases, the `synchronized` keyword cannot be removed because of the complexity of the operations. Figure 4 presents an illustrative example involving accesses to many variables. In this scenario, it is difficult to determine whether atomic data types and concurrent collections would be useful. It would require in-depth knowledge about the application and about concurrency control mechanisms.

Figure 5 presents an example where it is easy to remove the `synchronized` keyword and use an atomic data type. One could change the type of variable `s_lastResquest` from `int` to `AtomicInteger`. It would then be possible to remove the `synchronized` modifier. Instead of using the increment (`++`) operator for variable `s_lastRequest`, one should use the `getAndIncrement()` method. The latter works as a thread-safe increment operator for atomic integers.

This is a simplistic approach (the on adopted by Dig et al.) A more reasonable one would be involve tracking uses of the shared variables.

5. Limitations and Threats to Validity

In a study such as this, there are always many limitations. Firstly, to download the source code of the projects, we assumed that the sources were packaged in a file with the keywords “src” or “source” in its name. This is common practice in open source repositories. Nonetheless, it is not a rule and some projects are bound to adopt different naming conventions. We have ignored such projects. Moreover, we assume that most of the projects contain either versions or subprojects in each directory. However, a small number of projects contain both in the same directory. It is difficult to infer this automatically if no conventions are followed or if the conven-

tions are unknown. Hence, it is possible that some of subprojects were analyzed as versions of the main project and some versions were analyzed as subprojects. We stress that previous studies with similar scope [11] do not address this issue and may exhibit a much larger bias as a consequence.

Accuracy of measurement represents another threat to validity. Due to the large number of complex projects, it is impossible to automatically resolve all the dependencies on external libraries. As a consequence, we have to rely on purely syntactic analysis. This is sufficient to measure occurrences of `synchronized` and uses of monitor-based synchronization. However, to accurately collect some of the metrics, type information is necessary. To verify whether this purely syntactic approach would produce too many false positives, we have manually inspected samples comprising 100 randomly-selected projects. We did not find any metric for which more than 2% of the projects exhibited false positives.

6. Related Work

To the best of our knowledge, there are no large-scale studies that have attempted to gather data pertaining to the use of the concurrency constructs available in a programming language in the construction of real-world systems. Howison et al. [13] made a collaborative data and analysis repository, called FLOSSMole. It was designed to gather, share, and store comparable data and analysis of open-source projects. The major difference of our study from this approach is that it gathers project metadata (e.g. project topics), whereas we collect and analyze information at the source code level. Grechanik et al. [11] collected and analyzed the data at the source code level of OSS projects in large repositories. They described an infrastructure for conducting empirical research in source code artifacts and obtained insight into over 2,080 Java applications. While they randomly chose those java applications to study, we focus on mature, stable, and recently updated Java projects. This previous study analyzed only basic Java constructs and does not focus on any specific software characteristic. Bajracharya et al. [3] statically analyzed 2.852 java projects using SourcererDB, an aggregated repository of statically analyzed and cross-linked open-source Java projects. This work differs from ours because it does not focus on concurrent applications and performs only lexical analysis of source code.

These previous studies complement ours because they have examined the documentation of the processes that developers follow to build concurrent systems. On the other hand, our study investigates the products of these processes, the actual concurrent systems and try to answer if java programmers are transitioning to multicore. In addition, we can work at a much larger scale, because we analyze artifacts that were written in a programming language.

Dig et al. [2] analyzed five open source projects, including apache tomcat, and presented some metrics such as the number of synchronized blocks. Although we also have studied apache tomcat, unfortunately it was not possible to reproduce the results obtained by them, and the values do not match with ours. Even make a text search in the tomcat source code, the values found was much smaller than they showed in their paper.

7. Concluding Remarks and Future Work

This paper presents an empirical study of a large-scale Java open source repository. We found out that developers employ mainly simple mutual exclusion constructs. Almost 88% of the concurrent projects include at least one `synchronized` method. At the same time, approximately 27% of the projects employ higher level abstractions implemented by the j.u.c. library. We have noticed a tendency, albeit weak, of growth in the use of the j.u.c. library.

This study has revealed many opportunities for researchers working on program reestructuring approaches. We have identified that developers waste a large number of opportunities to use high level constructs for concurrent programming, in favor of lower-level, more error-prone constructs.

We also intend to investigate the organization of concurrency code in the analyzed projects. To achieve this goal, we will employ a number of metrics that aim to quantify tangling and scattering of code pertaining to specific concerns. Furthermore, we intend to analyze more specific issues. One that holds particular interest for us is the extent to which exception handling constructs complicate concurrent/parallel programming.

References

[1] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.

[2] Danny Dig, John Marrero, Michael D. Ernst. How do Programs Become More concurrent? A Story of Program Transformations. In *International Workshop on Multicore Software Engineering*, Hawai, USA, 2011.

[3] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An internet-scale software repository. In *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 1–4, 2009.

[4] A. Bernstein and A. Bachmann. When process data quality affects the number of bugs: correlations in software engineering datasets. In *MSR'2010*, Cape Town, South Africa, May 2010.

[5] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of OOPSLA'2010*, Reno, USA, October 2010.

[6] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[7] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, pages 397–407, Vancouver, Canada, 2009.

[8] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34:497–515, July 2008.

[9] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of DSN'2010*, Hong Kong, China, June 2010.

[10] Alessandro Garcia, Claudio SantAnna, Eduardo Figueiredo, Uira Kulesza, Carlos Jose Pereira de Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th ACM Conference on Aspect-Oriented Software Development*, pages 3–14, Chicago, USA, March 2005.

[11] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale java open source code repository. In *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement*, Bolzano-Bozen, Italy, September 2010.

[12] Maurice Herlihy. Linearizability. In *Encyclopedia of Algorithms*. Springer-Verlag, 2008.

[13] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and WebEngineering*, 1(3):17–26, July 2006.

[14] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal Of Computational And Graphical Statistics*, 5(3):299–314, 1996.

[15] M. G. Kendall. A new measure of rank correlation. *Biometrika*, June, 1938.

[16] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett

Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, August 2002.

[17] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.

[18] Doug Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, 2005.

[19] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006.

[20] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, March 2008.

[21] Joel Ossher, Sushil Krishna Bajracharya, and Cristina Videira Lopes. Automated dependency resolution for open source software. In *Proceedings of the 7th International Working Conference on Mining Software Repositories*, pages 130–140, Cape Town, South Africa, May 2010.

[22] Dag I. K. Sjoberg, Tore Dyba, and Magne Jorgensen. The future of empirical methods in software engineering research. In *Proceedings of 2007 Future of Software Engineering*, pages 358–378, 2007.

[23] W. Torres, G. Pinto, B. Fernandes, J. Oliveira, F. Ximenes and F. Castor. How do programmers use concurrency? http://www.cin.ufpe.br/ ghlp, Steptember, 2011.

[24] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), August 2005.

[25] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Dev.*, 46(1):5–25, January 2002.