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, to get 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 help other programmers to be more efficient when using the available constructs. Moreover, this evidence 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 spatial dimension, we 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 dimension we took a closer look at various versions of six projects and analyzed how the use of concurrency constructs has evolved over time. In addition, we tried to establish if uses of concurrency control constructs were aimed at leveraging multicore processors. We have downloaded more than two thousand Java projects including their various versions, in addition to individual analysing about six well known open-source projects.

Categories and Subject Descriptors: D.1.3 [Software Engineering]: Concurrent Programming

General Terms: Design, Experimentation

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. However, building this kind of application is a demanding and error-prone task [8, 15]. 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, and whether programmers are aware about evolution/transition from single core to multicore.

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, knowing how commonly programmers use these constructs, 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 an 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 - How does open-source software reflect the transition to multicore?

We obtained the source code of 2097 Java projects from SourceForge and performed an automatic analysis, collecting more than 50 different metrics related to concurrency from these projects. Six other well known open source projects (Apache Tomcat, Lucene, Mobicents, Backports, Fura and jMonkeyEngine). These projects comprise approximately 560 million lines of source code spread throughout more than 15,000 versions. We chose the Java language because it is a widely used object-oriented programming language. Moreover, Java 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. [9] 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

SPLASH'11 Workshops, October 23–24, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-1183-0/11/10…$10.00

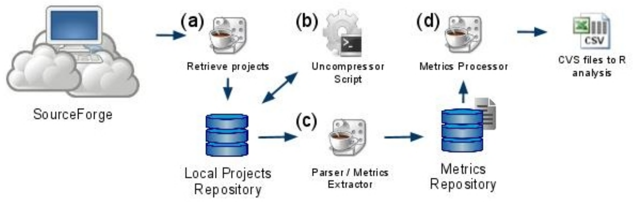

Figure 1. High-level view to our infraestrucutre.

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 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 higher level/more efficient 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 at least one of its classes or interfaces extends the `Thread` class or the `Runnable` interface or implements the `Runnable` interface or employs any concurrency control mechanism, such as `synchronized` blocks or `synchronized` methods. Beyond projects from SourceForge we also analyzed three Apache projects because they are known for 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’. In 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 analyzed 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 parses 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 [12].

The crawlers are an extention of Crawler4j[1], an open source web crawler framework. This framework is 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 was 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. The Metrics 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 of `Thread`, implements of `Runnable`, imports of j.u.c, `synchronized` methods, `synchronized` blocks, `Hashtable`, `HashMap`, `ConcurrentHashMap`, `AtomicInteger` and Lines of Code. The full list is available on the website [19].

3. Research Questions

One of the goals of our study is to understand how concurrency constructs are used in real world applications. In order to answer RQ1 we have analyzed the latest versions of more than 2000 projects, to understand which constructs developers of concurrent systems employ and how frequently they use them.

Question RQ2 aims to identify some characteristics of how programmers usually evolve code that requires concurrent skills, and if they are really moving to multicore.

On the other hand, question RQ2 is extremely complex and wide, and this paper does not cover it entirely. Moreover, to begin this study we need to analyze each project individually, looking for individual transformations of use, or disuse, of the most common constructs related to concurrency. This task is costly. Therefore, 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 these projects are very large, so we guided our analysis by searching in the source code for concurrency keywords and comparing the source code of different 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, both in the open source community and commercially. 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.

jMonkeyEngine: jMonkeyEngine (JME) is a game engine, made especially for game developers who want to create 3D games with modern technology standards. It is written entirely in Java, 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.

Backports: The goal of this project is to provide a concurrency library that works with uncompromised performance on all Java

[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

#Projects (subprojects included)	2.343
#Small concurrent projects	1.300
#Small non-concurrent projects	489
#Medium concurrent projects	635
#Medium non-concurrent projects	32
#Big concurrent projects	199
#Big non-concurrent projects	0
# of LoC of the last version of the biggest project)	1.702.972
Size on disk (all versions of all projects)	124GB

Table 1. General information about the projects.

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 presentation is organized in two parts. Section 4.1 addresses RQ1 and Section 4.2 addresses 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 metrics for the projects we have downloaded. We analyzed 2343 project (subprojects included) in total, but only 1830 are considered concurrent and only 439 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 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 average, considerably larger than the non-concurrent ones. This is expected: most complex projects involve concurrency at some level.

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 of 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. The complete results of the study are available on the website[19].

Synchronized modifier. We broke the analysis for the synchronized modifier in two, based on its two forms. `synchronized` blocks are present in 709 of 1300 small concurrent projects, 541 of 635 medium concurrent projects and 189 of 199 big concurrent projects. The standard deviation for synchronized block is 22.56, 65.88 and 189.73 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. `Synchronized` methods are present in 72,84% of small concurrent projects, 93,22% of medium concurrent projects and 98,49% 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 63,77% of the medium concurrent projects and 76,38% of the big concurrent projects extend `Thread`, small projects only 40,46% extend `Thread`. More than 85% of small concurrent projects invoke `Thread` methods and almost 100% of medium and big projects invoke `Thread` methods. We have also measured the number of classes that implement the `Runnable` interface and the number of interfaces that extend the `Runnable` interface. In accordance with our intuition, implementing `Runnable` is the most popular approach, with higher median, mean, and 3rd quartile. As expected, a small number of projects have interfaces which extend `Runnable`.

Use of j.u.c. This metric represents the sum of all related j.u.c library metrics. Overall, 15,46% concurrent small projects, 32,44% concurrent medium projects and 49,24% concurrent big projects are using the library. As expected, big projects are the ones which use j.u.c. the most. For example, there is a big project that uses j.u.c. 740 times.

Atomic data types and concurrent collections. Contrary to our intuition, few projects employ atomic data types, 3% of small concurrent projects, 11,02% of medium concurrent projects and 23,61% of big concurrent 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 6,61% of small concurrent projects, 15,27% of medium projects and 30,65% of big projects use concurrent collections.

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, how did the use of concurrent control constructs changed throughout the version of the same system?
- Have threads been used to improve 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 release resources. To that end, basic constructs like the `synchronized` keyword,

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

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

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

2***metrics**	**Median**			**Mean**			**Std. Dev.**			**%Projects**		
	S	M	L	S	M	L	S	M	L	S	M	L
# synchronized blocks	4	14	51	11.27	39.47	119.7	22.56	65.88	189.73	54,53	85.19	94.97
# synchronized methods	6	24	90.5	12.63	50.53	152.3	20.07	74.02	188.66	72.84	93.22	98.42
# classes extending Thread	1.5	3	6	2.41	4.8	9.46	2.17	6.07	11.32	40.46	63.77	76.38
# uses of Thread methods	7	27	82.5	13.96	52.51	277.68	22.23	94.19	1699.72	86.38	97.48	98.49
# implementing Runnable	2	3	6	2.71	6.38	13.04	3.02	10.3	17.8	46.30	66.45	84.42
# interfaces extending Runnable	1	2	1	1.91	2.86	6.05	1.24	3.60	14.17	0.92	2.36	8.54

Table 2. Projects metrics 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.

2***metrics**	**Median**			**Mean**			**Std. Dev.**			**%Projects**		
	S	M	L	S	M	L	S	M	L	S	M	L
# Uses of j.u.c	5	8	16	9.62	30.38	64.07	14.32	75.27	122.32	15.46	32.44	49.24
# Atomic data types	2	3	6	3.07	9.05	16.02	2.71	16.16	21.63	3	11.02	23.61
# Concurrent collections	2	3	4	3.17	7.12	13.07	4.12	11.17	21.01	6.61	15.27	30.65
# Locks	1	2	4	2.32	7.17	6.77	3.48	17.91	8.70	3.30	12.91	26.63
# Barriers	3	2	2	4.66	7.32	14.07	6.01	14.87	21.21	1.15	4.88	14.57
# Futures	2	2	2	3.80	3.61	2.88	3.83	3.79	2.26	2.38	6.61	13.06

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

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 to page limits, the other results are present only on the website[19]), 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 the number of uses of j.u.c only started to increase significantly at version 2.1. 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, the use of the j.u.c constructs in test case classes seems to be common in open source projects, including three of the six projects that we have analyzed. The use of `Executors` and `ExecutorService` during the test execution revealed a new behavior of software developers, i.e. despite the large number of testing frameworks, programmers still prefer to use j.u.c. constructs to conduct some testing activities.

We observed that the Fura project showed a different pattern. We observed that the use of `synchronized` keyword increased about 37%. But, when we analyzed the rate of `synchronized` keyword per 100KLOC it almost did not increase. On the other hand, the rate of j.u.c per 100KLOC quadrupled from the first version to the 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, many methods used the `synchronized` keyword in an early version and in the next one use the `Lock` and `ReentrantLock` classes. Backport is the project that takes the most advantage of j.u.c. constructs, in terms of the number of uses of j.u.c per lines of code.

Moreover, analyzing data structures, it is interesting to note that the use of `HashTable` decreased along all the project versions while `HashMap` and concurrent collections increased. Perhaps programmers are aware of the inefficiency of `HashTable` and have chosen other collections to increase application performance: a potential 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 it can be difficult to understand their usage. We sorted threads into two groups: threads that handle 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.).

```
private class ParallelTask extends Thread {
   @Override
  public void run() { try {
      int n = task.runAndMaybeStats(letChildReport);
      if (anyExhaustibleTasks) {
        updateExhausted(task); }
      count += n;
    } catch (NoMoreDataException e) {
      exhausted = true; } catch (Exception e) {
      throw new RuntimeException(e); } } }
```

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

The selection of these groups was related to the fact that applications that use threads to accomplish simultaneous operations are not necessarily related to parallelism. Threads can improve resource usage but, at the same time, waste CPU cycles. In turn, the second group is directly related to the performance impacts of the transition to multicore. Thus, we would like to compare the growth of the second group with regard to the first one.

In each project we identify whether threads aim at improving concurrency or parallelism. Following theses steps:

- For each project, seek thread constructs in source code by textual search.
- Inspect the code comments and the code and try to understand what that code does.
- Do the same with the other versions.

As a result, we consider Backport and Lucene as projects that use more parallelism. Particularly, Lucene is a project that has cared about parallelism since the first version that we analyzed. This is stated in block comments, method and class names, and so on. Figure 3 shows an example using the `ParallelTask` class.

Tomcat is a project that takes 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 to the fact of Tomcat is

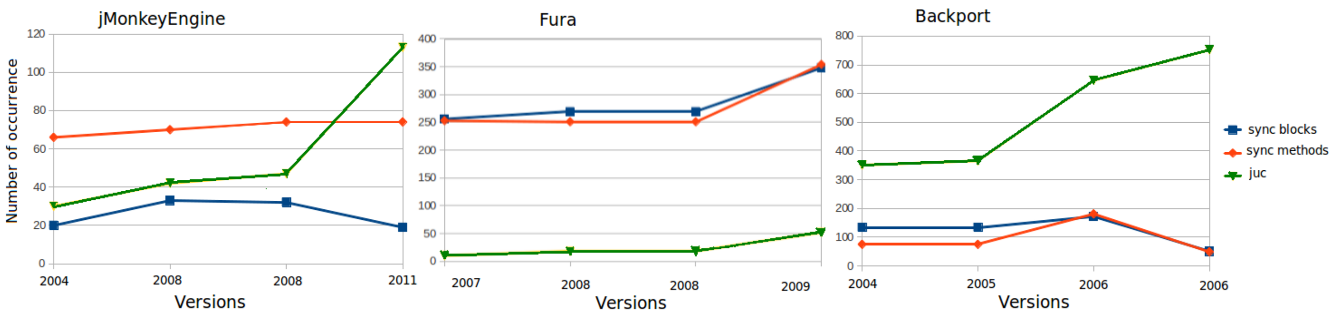

Figure 2. Temporal evolution of synchronized methods and blocks and j.u.c

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

Figure 4. An example of source code of Javagamelibrary project

```
...
private static int s_lastRequest = 0;
...
public static synchronized int getNextSequenceNumber() {
  return s_lastRequest++; }
```

Figure 5. An example of source code of Opensubsystems project

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

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 chose 100 projects out of all the 1523 concurrent projects. For each one, we randomly collected 1–3 examples of the use of the `synchronized` keyword. For each example, we manually analyzed the use of `synchronized` in that block or method to see if it would be feasible to replace it with concurrent collections or atomic data types. Dig et al. [6] present a list of code templates for situations where uses of `synchronized` can easily be replaced by uses of atomic data types. That list served as a basis for the manual inspection.

We analyzed 276 examples of `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. 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 `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 one adopted by Dig et al. [6]): 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 conventions are unknown. Hence, it is possible that some of the subprojects were analyzed as versions of the main project and some versions were analyzed as subprojects. We stress that previous studies with similar scope [9] 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. [11] made a collaborative data and analysis repository, called FLOSSMole. It was designed to gather, share, and store comparable data and analyses of open-source projects. The major difference of our study from this approach is that it gathered project metadata (e.g. project topics), whereas we collect and analyze information at the source code level. Grechanik et al. [9] 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, it was not possible to reproduce the results obtained by them, and the values do not match with ours. Even making a text search in the Tomcat source code, the values are much smaller than what 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 23% of the concurrent 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 restructuring 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.

8. Acknowledgments

We would like to thank the anonymous referees and our shepherds Caitlin Sadowski and Neha Rungta, who helped to improve this paper. Fernando is supported by CNPq/Brazil (308383/2008-7 and 475157/2010-9) and FACEPE/Brazil (APQ-0395-1.03/10). João is supported by FACEPE/Brazil. Weslley and Filipe are supported by CNPq/Brazil. Gustavo is supported by CAPES/Brazil. This work is partially supported by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

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