A study on the usage of third party libraries in Java applications

Marta Teixeira and João Cachopo
{marta.teixeira,joao.cachopo}@ist.utl.pt

Software Engineering Group, Inesc-ID,
R. Alves Redol 9, 1000 Lisboa, Portugal
https://fenix.ist.utl.pt/investigacao/inesc-id/ESW

Abstract. The high number of libraries available for the Java platform certainly contributes to the popularity of the JVM as the target of many modern applications. By using libraries, developers not only spend less time programming, but they also improve the quality of their own applications by relying on well-tested and highly-optimized solutions. So, nowadays, programmers spend more time mixing-and-matching software modules from libraries than ever, which means also that applications spend increasingly more time executing code that is in fact from libraries. If the execution of code from libraries contributes significantly to the total execution time of certain applications, then an enticing approach to optimize those applications is to parallelize the libraries they use, so that they make better use of the modern multicore hardware. Whether this approach works or not depends on how often and for how long are libraries used.
To evaluate the role that third party libraries play in Java modern applications and the eventual effect that their parallelization may have, this paper presents a study based on two different kinds of analyses: A source code analysis that identifies the most commonly used libraries across a large selection of Java programs, and then a runtime analysis that details the usage of the libraries by specifically evaluating the execution times and number of executions of portions of code from each library. The dataset used for this work was composed of more than two thousand of Java open source applications available in the Sourceforge repository. Our analysis reveals that 40% of the projects in the dataset uses third party libraries and that, in some cases, the time spent executing code from libraries is sufficiently high to become profitable to parallelize those libraries.

Keywords: Java Libraries, Modern applications, Open source, Multicore Architectures, Parallel Programming.

1 Introduction

Moore's Law [1] states that the number of transistors on a chip doubles approximately every two years. For years, clock speed of transistors increased without

overheating according to this law and, consequently, processing speed or memory capacity increased too without any intervention of the programmers.

Yet, the overheat caused by the increasing clock speed made manufacturers turn into a different architecture. Instead of building more powerful chips, they decided to put multiple processors together communicating directly through shared hardware caches (known as chip multiprocessors or multicore systems). As the number of processors on a single chip will continue to double at Moore's law rate, so will the number of instructions executed per second, which means that even without increasing clock speed, this parallel architecture offers a potential solution to achieve better performances. However, it is not easy for software developers to embrace parallelism in their applications.

On the one hand, despite all of the extensive literature on parallel programming (e.g., [2–6]), designing, writing, debugging, proving correctness or finding non-deterministic bugs is far more difficult in parallel programming than in sequential programming. On the other hand, to parallelize effectively applications that make extensive use of third party libraries may require the parallelization of those libraries too, if we want to achieve the desired performance. Modern Java applications, for example, often use algorithms from libraries, even though different programs use different libraries in different ways.

We may say that a Java class is part of a library if it is used by more than one application. Classes belonging to the standard Java platform fit naturally within this classification, but anyone can program class libraries to be used in different programs—that is, *third party class libraries*. For the purpose of this work, we are interested only in finding how frequently used are third party libraries. We assume that almost all Java programs make use of the standard Java libraries and that these libraries, being part of the Java platform, are likely more optimized and are harder to parallelize.

Moreover, programs can make a single call for a library that executes for a long time, or make multiple calls where each executes for a very short time. In the latter case, we expect significant overhead if we parallelize the corresponding library operations. So, we have to take into account these two measures to evaluate which gains to expect from the parallelization.

To better evaluate the role that third party libraries play in Java modern applications and the eventual gains obtained by their parallelization, with this work we intend to answer the following research questions:

- **Q1**: How often do developers use third party libraries?
- **Q2**: Which third party libraries do developers use?
- **Q3**: What are the potential gains obtained by parallelizing third party libraries in Java?

The answers to questions Q1 and Q2 will be determinant to have an overall vision about the use of third party libraries and to have general information on the gains that can be achieved with their parallelization. The answer to question Q2 also allows to know which specific tasks are done by third party libraries and evaluate possible reasons programmers have to use these libraries instead

of other approaches (for example, using standard libraries provided by the Java Platform).

Question Q3 is more specific and difficult to answer. Yet, it is crucial to understand the role that libraries have on each application. To assess the potential gains obtained by the parallelization of libraries, we need to know how often each application calls third party libraries and for how long each third party library runs during each application total runtime.

To identify a set of candidate third party libraries to be parallelized and to discover the usage of these libraries, we decided to use large open source repositories. There are several of these repositories, such as Sourceforge [22], Google Code Hosting [23], and Apache [24]. Due to the challenges described in the following sections we obtained the dataset for this study only from the Sourceforge repository.

By analyzing a large number of real Java applications we are obtaining empirical results. These results are more accurate in the sense that the applications are being used and do not come from benchmarks that may only be useful to achieve comparable results. Yet, by facing real world applications instead of benchmarks, there are other challenges, because programs are not developed to be comparable among them.

The paper begins, in Section 2, with an additional motivation to this work. Section 3 describes our study; it presents the high-level infrastructure, the main challenges, and the solutions we found to those challenges. Then, Section 4 presents the evaluation of the results produced by the study. In Section 5, we present the related work, and, finally, we conclude and discuss future work in Section 6.

2 Why parallelizing libraries?

The idea of achieving better performances by parallelizing libraries is not new. First, because explicit parallelization is expensive and automatic parallelization is limited to simple programs. Second, because when we parallelize a library we may be decreasing not only the execution time of one single program but of all the programs that use that library.

Libraries containing numeric algorithms implementations have been parallelized successfully for a long time. Spiral [7], for example, is an active project where the parallelization results prove to be good in several mathematical functions and in linear transforms including the discrete Fourier transform, discrete cosine transforms, convolution, and the discrete wavelet transform.

Besides numeric libraries, non-numeric libraries started to be parallelized, too. The C++ Standard Template Library (STL) [8] has been the starting point for many works, because it is a part of the C++ programming language and offers many well-known and useful algorithms. Implementing parallel libraries, such as the Multi-Core Standard Template Library (MCSTL) [9], the Deque-free work-optimal parallel STL algorithms [10], Parallelization of Bulk Operations for STL Dictionaries [11], or the Standard Template Adaptive Parallel Library

(STAPL) [12], is a very complex and difficult task. However, these implementations have achieved better performances in many programs that use them, making them a useful addition to the toolbox of the multicore programmer.

To the best of our knowledge, there are no equivalent libraries in the Java platform. Yet, the problems addressed by the previously mentioned work are analogous in both programming languages and the results obtained give us a good reason to explore these approaches in Java, too. We believe that by identifying the most used third party Java libraries we are able to identify a good set of libraries that may benefit from a similar parallelization effort.

3 Identifying third party libraries often used

The efficient parallelization of Java libraries has the potential to improve not only the performance of one single application, but of all applications that use these libraries. So, it is important to find a good representative dataset of Java programs to cover as many applications as possible. Existing Java benchmarks rarely use Java libraries, making them a less useful option for the purpose of this work.

Here, we analyzed a large collection of Java projects that were obtained from the Sourceforge repository and that were operating system independent. In the end, we used 2602 projects out of 5006 extracted projects. The information about whether a project is operating system independent and implemented in Java is defined by the project's maintainers at Sourceforge. We relied on that information to select the initial set of projects, but then, at different steps of our process, we discarded some projects that were not correctly classified, so that we improve the reliability of our findings.

We consider that a class is part of a library when it is used by more than one project. So, we need to be able to find out when does a project uses a class. There are various ways on how this can be done and the approach that we use, which we describe below, does not cover all cases (for instance, it does not take into account the use of reflection to access a certain class), but it is relatively simple to implement and we believe that it gives a very good approximation of the correct result.

Moreover, in our analysis, we do not use a larger granularity than the class, such as the package granularity, because we are trying to identify which libraries may benefit from being parallelized and, for that, we need to know exactly which code was called.

In fact, our analysis is composed both of a source code analysis and a runtime analysis. Given the size of the original projects dataset, both in terms of number of projects and file size on disk (more than 50GB of information), and that each project may have its own building process, it would not be feasible to compile and run each project to determine which third party libraries they use (which we could do by looking at the class loading done by the JVM, for instance). So, instead, we perform a first source code analysis that eliminates all projects that do not have any imports of a class in common with any other of

the testing projects. The runtime analysis is more complex as it needs to execute the projects, which must be previously compiled manually. This step, however, is extremely important to assess the real usage of third party libraries, as programs may import classes and never use them. Furthermore, only with a runtime analysis it is possible to know how many times a program calls a specific class and how long each call takes. We believe that these two measures are crucial to decide, in the future, which of the library classes are the best candidates to parallelize.

3.1 Analysis infrastructure

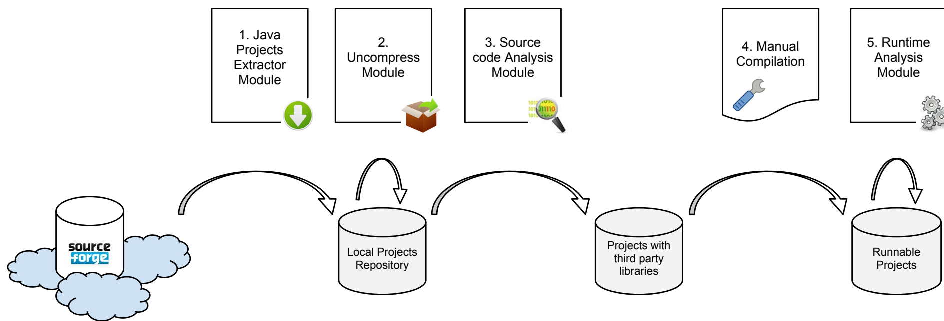

Fig. 1. Infrastructure of the System

To answer the research questions that we formulated in the introduction, we created the infrastructure shown in Figure 1. This infrastructure also takes into account the challenges described below. Our infrastructure is composed of five main modules:

1. **Java Projects Extractor Module.** Collecting source code from large open source repositories represented the first challenge. This kind of repositories does not have features that allow us to perform the data collection needed. The best way that we found to obtain the source code was to construct manually the search links, taking into account the desired options (for example, that we only wanted Java projects) and analyze the website results to get download links for the projects. Then, we had to download the projects from those links. Yet, the use of different download protocols and the different formats/contents of the web pages made automating this collection process only possible for the Sourceforge repository.
   So, this module is responsible for collecting the dataset from the Sourceforge repository. It has configurable properties that adapt the first search link and allow to extract different and selected information from the repository. This makes the module more flexible, so that it may be used in other contexts, as

long as Sourceforge maintains its links' architecture. For the purpose of this work, the module is configured to extract only programs written in the Java programming language and that are independent from the operating system. At first, the module gets all the project links found according to the desired properties and then it proceeds to the projects extraction link by link.

2. **Uncompress Module.** The projects obtained from the Sourceforge repository are compressed in different well-known formats, such as tar, zip, or rar (among others). This module is composed of a shell script that uncompresses the projects according to their formats. Even though we chose the option available in Sourceforge to select only programs written in Java, some of the uncompressed projects revealed to be written in different programming languages. Therefore, this module also filters all projects after the uncompressor process and discards all non-Java projects.

3. **Source Code Analysis Module.** Once projects have been downloaded and extracted, we want to identify the use of libraries on the dataset, by performing a source code analysis. This corresponds to the next challenge. Some libraries, such as the JDK libraries (e.g., collections from *java.util*), are well known and, so, it is easier to identify if a certain program uses them. On the contrary, third-party libraries (i.e., libraries that are not JDK libraries) are more difficult to find as they are not known in advance. As libraries are composed of classes that are used in more than one project, the most used libraries will be the libraries used in more programs.

We designed this source code analysis module in a way that, given the source code of each project, it produces a list of classes associated with the number and the names of projects that use each class. Yet, as mentioned above, there is no single way of knowing when a program uses a certain class. In our approach, we considered that a project uses a class only when somewhere in the code of the project there is an import of that class.

4. **Manual Compilation.** To perform a runtime analysis and fully extract information from the project's code, the code must be complete with all the dependencies resolved. Unfortunately, with a dataset obtained from a large open source repository available on the Internet, there is no guarantee that the code is complete. Incomplete files, missing dependencies, and missing explanations on how to compile programs are quite common and, considering that we are looking at thousands of projects, makes the manual compilation and the preparation of the programs to a runtime analysis a much more difficult task.

Given the results from the previous module, we have the potential programs that use third party libraries (because they import them). In this step, considering this new reduced set of projects we try to compile and run manually program by program. This step requires a lot of time due to the difficulties explained above and the huge dimension and complexity of the candidate projects.

5. **Runtime Analysis Module.** The runtime analysis allows us to produce more accurate results on the projects that use third party libraries. More than statically looking for imports of classes, this analysis gives us which of the classes are effectively called and is also able to measure the time that each library class takes running.

To achieve these goals we may use existing profiling tools. However, is was necessary to install the tools on each testing machine and make huge changes to the tools to automatically process all the testing projects.
So, our Runtime Analysis Module is a Java program that uses the Javassist (Java Programming Assistant) framework [13, 14] to perform code intercession during runtime in each testing program. This framework is a class library for doing bytecode rewriting in Java. We used it to modify the projects class files when the JVM loads them. We insert around advices on the caller side, each time that it performs a call to a class of a third party library, thereby allowing the identification of the exact class library called and the measurement of time spent during the execution of that method call. These around advices also write the informations into log files, which are then processed by another part of the Runtime Analysis Module to analyze the results obtained during the execution of each project.

With these five main modules, we believe that we may now answer our three research questions.

The Source Code Analysis Module produces results that give us preliminary answers to our first and second research questions, because it tells us how many and which class libraries are imported by the various projects in our dataset.

The Runtime Analysis Module produces more accurate results to answer those two questions and provides also answers to the third research question: Its results allow us to make a better assessment of the advantages of parallelizing specific third party libraries, because it gives us some measures about their real usage inside applications.

4 Results

This section presents the results of the analysis described in the previous section. The presentation is organized in two parts. Section 4.1 shows the source code/static analysis results. These results allow us to have preliminary answers to our first and second research questions and to have a better starting point to answer the last research question. The results for the runtime analysis are in Section 4.2 and, with them, we intend to answer our last research question and to gain insight on how we may improve the performance of Java applications running on multicore machines by parallelizing the third party libraries that they call.

4.1 Source code Analysis

The source code analysis was performed over 2602 real Java applications extracted and uncompressed from the Sourceforge repository. The results show that 1043 projects (40% of the total projects) declare one or more classes in common to at least another of the tested projects.

In those 1043 projects there are 8830 different classes that are called by more than one project—that is, 8830 classes belonging to some third-party library.

The class most commonly used across all of the projects is *org.jdom.Element*, which is called by 87 different applications. The library containing this class is well known and provides a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code. Libraries packages are composed of many different classes. Despite we have the packages of classes in the import statement, it is impossible to perform an automatic association of classes and the corresponding packages. Imagine, for example, the packages *org.jgraph* and *org.jfree*. They seem to belong to the same package *org* but in fact they are two complete different packages from complete different libraries. Yet, the packages *jgraph.layout* and *jgraph.graph* may be automatically identified as different packages when they belong to the same library. Despite being impossible to determine automatically the corresponding packages to each of the classes found, we decided to include in our results the top ten libraries packages presented on table 1. This table is also important to show the starting point for the runtime analysis that we performed after this source code analysis, because we have here the main, i.e. the most imported, third party Java libraries we found.

There is a natural interaction between different classes in the same package, and we also verified it in our results where most of the classes from the same package come together, with the same projects importing them. We believe that the fact that programmers import single classes instead of their packages is regarded to modern IDEs that automatically import class by class.

So, it is not very interesting to show here all of those classes and, therefore, we merged the most used classes into their packages, which we show on Table 1. In this table, we observe that the most used of these library packages are used by a significant portion of the total projects on the dataset. For instance, by parallelizing 29 classes from *org.jdom* we may be improving the performance of up to 3,69% of all the projects available in our dataset and up to 9,20% of the programs that use third party libraries. What we still need to know is how much time do these percentages represent in the runtime of the applications.

Class Name	#Classes	#Projects	%Libs Projects	%Overall Projects
org.jdom	29	96	9,20%	3,69%
org.jgraph	27	91	8,72%	3,50%
org.springframework	373	88	8,44%	3,38%
org.jfree	382	62	5,94%	2,38%
org.hibernate	629	57	5,47%	2,19%
org.dom4j	36	52	4,99%	2,00%
sun.misc	10	47	4,51%	1,81%
org.slf4j	21	40	3,84%	1,54%
bsh	11	33	3,16%	1,27%
org.osgi	12	31	2,97%	1,19%

Table 1. Top ten libraries packages

Another interesting result from our preliminary analysis is shown in Figure 2: In general, one single application does not use too many different libraries. In fact, 12,08% of the applications that use third party libraries only use one library class. More than 50% of the programs that use third party libraries use less than

10 different classes, which is a promising result for our original goal, because it suggests that we may achieve better performance of a large set of programs, by parallelizing a restricted set of library classes. Projects that use many library classes are not easy to find. Still, we found both the project called *beeing* that uses 1367 library classes and the *jasperreports* project that uses 1182 classes. Again, these are not common examples and, given the large number of classes that they use, it will surely be more difficult to obtain significant gains with the parallelization of a single library. Thus, we are also not taking them into account in the runtime analysis.

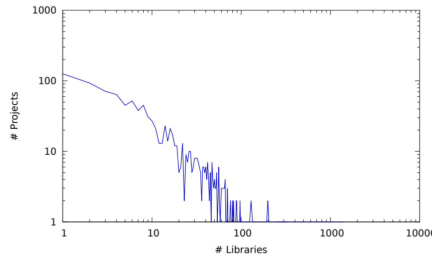

Fig. 2. Number of projects and the number of libraries they use

4.2 Runtime Analysis

Besides the challenges described on the manual compilation process, we had an additional challenge for achieving comparable results in the runtime analysis. As our testing programs are real applications, they tend to have graphical user interfaces. These interfaces do not allow us to run the programs unattended, requiring instead some form of user interaction. This, however, is not only not practical, as it introduces a lot of variety in the execution time of various runs of the same program. This becomes a problem, because we measure exactly the time each program spends executing code from libraries but we cannot know what does this time represents in the total execution time of the programs.

So, the results we present here are relatively few due to the significant modifications that we had to perform in programs to have reproducible execution times. Moreover, we had to take into account the purpose of the third part libraries. If libraries take part in graphical user interfaces, for example, we may not want to remove or to modify them. We had to find trade offs of maintaining the methods that use the third party libraries or achieving comparable times between different executions.

According to the results obtained in the source code analysis section, we tried to compile and run programs that use the top four libraries presented on Table 1: $org.jdom$, $org.jgraph$, $org.springframework$, and $org.jfree$.

We present the main results for the runtime analysis in Table 2. This table indicates the testing programs, also available under the Sourceforge repository, that have better chances to gain with the parallelization of the libraries found

previously. The high percentage of time that the wikimapper spends executing code from the *org.jgraph* library is explained by the modifications performed in the application. As it had a user interface, we needed to disable those functionalities. However, the time that the user may spend interacting with the application can be considered useless in this context as processors are not realizing any task during that time.

Program	Class	Method	#Calls	%Time
brihaspati2	org.jdom.Element	getNameSpace	1	0,01%
wikimapper	org.jgraph.graph.GraphLayoutCache	insert	3	82,51%
dtreejungle	org.jgraph.layout.JGraphLayoutAlgorithm	applyLayout	4	65,73%
graphalg	org.jgraph.graph.DefaultCellViewFactory	createEdgeView	14	12,34%
dtreejungle	org.jfree.base.AbstractBoot	performBoot	1	0,2%

Table 2. Most called methods during the Runtime Analysis

Still, the results presented in this section show that applications call libraries many times but also show that they spend relevant execution time executing code from libraries.

5 Related Work

There are several studies (e.g., [15–18]) on the usage of libraries or programing language features. These studies, however, rely only on source code analysis. To the best of our knowledge, ours is the first large-scale study that uses both source code and runtime analysis to evaluate the usage of third party libraries in Java, thus increasing the accuracy of the statistics.

Pankratius [17] proposes to extend the Eclipse IDE to evaluate the usability of parallel language constructs. By recording the usage patterns of users, they are able to infer correlations using datamining techniques.

Another study from Dey et al. [19] on the practice of multicore programming focused on identifying how contention for particular shared resources has impact in the performance of parallel applications. We also identify parts of programs, i.e. third party libraries, that may be shared resources when used by multi-threaded applications or by different applications at the same time. Yet, we are not analyzing the contention for these third party libraries but only how their parallelization may improve the performance of the programs that use them.

Grechanik et al. [20] collected more than 2,000 Java applications and analyzed their source code artifacts. Whereas they analyzed only basic Java constructs, we focused on a specific software characteristic: the usage of third party libraries.

The work from Torres et al. [21] is the closest to ours. Their study measures the usage of concurrent programming constructs in around 2,000 Java applications. The results of a source code analysis give the number of classes extending Thread and the number of synchronized blocks. In contrast, our study not only performs a source code analysis counting imports of third party libraries but also has a runtime analysis evaluating if libraries are called, the number of times they are called, and for how long.

6 Conclusions and Future Work

One of the goals of our study is to understand how often do developers use third party libraries (Q1). We found that 40% of the analyzed projects use one or more of 8830 different third party libraries classes. This represents a significant percentage of applications that may benefit from a third party libraries efficient parallelization.

Research question Q2 aims to identify which third party libraries do developers use. Our study shows that developers use most libraries for accessing, manipulating, and outputting XML data from Java code, libraries that provide diagramming software components, and libraries that provide a comprehensive programming and configuration model for modern Java-based enterprise applications.

Finally, our last goal is to measuse the gains obtained by parallelizing third party libraries in Java (Q3). According to the explored projects there exist some libraries, for example *org.jgraph*, that perform many calculations that take many of the total execution times of the programs that use them. This leads us to good examples to explore parallelism.

Considering the results obtained and the motivation presented in Section 2, we believe that if we parallelize the execution of libraries on multicore machines, we may speedup a significant fraction of some applications' execution time. Another potential advantage for this parallelization is that we are not improving the performance of one specific application, but of all the applications that use the parallelized library. So, part of the future work is certainly to look into the main third party libraries found in this study and try to parallelize them.

Moreover, it may be an important effort to perform runtime analysis over more projects, to discover more cases with potential gains in parallelizing libraries and eventually evaluating the usage of the standard Java libraries, too.

7 Acknowledgments

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under the RuLAM project (PTDC/EIA-EIA/108240/2008).

References

1. Moore, G.: Cramming more components onto integrated circuits. In Electronics Magazine, 19 April 1965.
2. Grama,A., Gupta,A., Karypis,G., and Kumar,V.: Introduction to Parallel Computing (2nd edition). Addison-Wesley, 2003.
3. Timothy G., Mattson, Beverly A. Sanders, and Berna L. Massingill: Patterns for Parallel Programming. Addison-Wesley Professional, 2005.
4. Goetz,B., Peierls,T., Bloch,J., Bowbeer,J., Holmes,D., and Lea,D.: Java Concurrency in Practice. Addison-Wesley Professional, 2006.
5. Lea,D.: Concurrent Programming in Java l: Design Principles and Pattern. Prentice Hall, 1999.

6. Lester,B.: The Art of Parallel Programming. 1st World Publishing, Inc., 2006.
7. Püschel,M., Moura,J., Johnson,J., Padua,D., Veloso,M., Singer,B., Xiong,J., Franchetti,F., Gacic,A., Voronenko,Y., Chen,K., Johnson,R., and Rizzolo,N.: SPIRAL: Code Generation for DSP Transforms. *In Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, volume 93, pages 232-275, 2005.
8. Plauger,P., Stepanov,A., Lee,M., and Musser,D.: The C++ Standard Template Library. Prentice-Hall, 2000.
9. Putze,F., Sanders,P., and Singler,J.: MCSTL: the multicore standard template library. *In PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 144-145, New York, NY, USA, 2007. ACM.
10. Traore,D., Roch,J., Maillard, N., Gautier,T., and Bernard,J.: Deque-free work-optimal parallel stl algorithms. *In Proceedings of Euro-Par'08*, pages 887–897, 2008.
11. Frias,L., and Singler,J.: Parallelization of bulk operations for STL dictionaries. In *Workshop on Highly Parallel Processing on a Chip (HPPC)*, number 4854 in LNCS, pages 49–59, 2007.
12. Buss, Harshvardhan,A., Papadopoulos,I., Pearce,O., Smith,T., Tanase,G., Thomas,N., Xu,X., Bianco,M., Amato,N., and Rauchwerger,L.: STAPL: Standard Template Adaptive Parallel Library. *In SYSTOR '10: Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, New York, NY, USA: ACM, pages 1-10, 2010.
13. Chiba,S.: Javassist — A Reflection-based Programming Wizard for Java. *In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
14. Chiba,S., and Nishizawa,M.: An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03), LNCS 2830, pp.364-376, Springer-Verlag, 2003.
15. Bajracharya,K.S., Ossher,J., Lopes,C.V. Sourcerer - An Infrastructure for Large-scale Collection and Analysis of Open-source Code. *In WASDeTT-3*, Belgium, 2010.
16. Dig,D., Marrero,J., and Ernst,M. How do programs become more concurrent? a story of program transformations. *In IWMSE '11*, 2011.
17. Pankratius,V., Schaefer,C., Jannesari,A., and Tichy,W. Software engineering for multicore systems: an experience report. *In IWMSE'08*, 2008.
18. Parnin,C., Bird,C., and Murphy-Hill,E. Java generics adoption: how new features are introduced, championed, or ignored. *In MSR*, pages 3-12, 2011.
19. Dey,T., Wang,W., Davidson,J., and Soffa,M. Characterizing multi-threaded applications based on shared-resource contention. *In Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:76–86, 2011.
20. Grechanik,M., McMillan,C., DeFerrari,L., Comi,M., Crespi,S., Poshyvanyk,D., Chen Fu, Qing Xie, and Ghezzi,C. 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.
21. Torres,W., Pinto,G., Fernandes,B., Oliveira,J., Ximenes,F., and Castor,F.: Are Java programmers transitioning to multicore? A large scale study of java FLOSS. *In SPLASH '11 Workshops*, 2011.
22. Web Site for Sourceforge. http://sourceforge.net.
23. Web Site for Google Code Hosting. http://code.google.com/projecthosting.
24. Web Site for Apache Software Foundation. http://apache.org.