**Karlsruhe Reports in Informatics 2011,31**
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

# On the Benefits of Combining Functional and Imperative Programming for Multicore Software

An Empirical Study Comparing Scala and Java

Victor Pankratius, Felix Schmidt, Gilda Garretón

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

# On the Benefits of Combining Functional and Imperative Programming for Multicore Software:

## An Empirical Study Comparing Scala and Java

Victor Pankratius
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
www.victorpankratius.com
pankratius@acm.org

Felix Schmidt, Gilda Garretón
Oracle Labs
Oracle Corporation
Redwood Shores, CA, USA
{felix.x.schmidt,gilda.garreton}@oracle.com

## ABSTRACT

Recent multi-paradigm programming languages combine functional and imperative programming styles to make software development easier. Given today's proliferation of multicore processors, parallel programmers are supposed to benefit from this combination, as many difficult problems can be expressed more easily in a functional style while others match an imperative style. However, due to a lack of empirical evidence from controlled studies, important software engineering questions are largely unanswered. Our paper is the first to provide thorough empirical results by using Scala and Java as a vehicle in a controlled comparative study on multicore software development. Scala combines functional and imperative programming while Java focuses on imperative shared-memory programming. We study thirteen programmers who worked on three projects, including an industrial application, in both Scala and Java. In addition to the resulting 39 Scala programs and 39 Java programs, we obtain data from an industry software engineer who worked on the same project in Scala. We analyze key issues such as effort, code, language usage, performance, and programmer satisfaction. Contrary to popular belief, the functional style does not lead to bad performance. Average Scala run-times are comparable to Java, lowest run-times are sometimes better, but Java scales better on parallel hardware. We confirm with statistical significance Scala's claim that Scala code is more compact than Java code, but clearly refute other claims of Scala on lower programming effort and lower debugging effort. Our study also provides explanations for these observations and shows directions on how to improve multi-paradigm languages in the future.

**Categories and Subject Descriptors:** D.1.3 [Programming Techniques]: Concurrent Programming – Parallel programming. **General Terms:** Human Factors, Experimentation.

Technical Report 2011-31
Karlsruhe Institute of Technology (KIT), Germany
Institute for Program Structures and Data Organization (IPD)
November 18, 2011

## 1. INTRODUCTION

Multi-paradigm programming languages conjecture that no single paradigm is suited to solve all possible problems in practice. In particular, recent proposals such as [1, 2, 3] fueled the development of languages that unify the best of functional programming and imperative programming. This direction is motivated by the need to produce more reliable software despite the growing complexity that programmers face in today's environments. Programming languages thus aim to offer a better cognitive match between their constructs and the problems that developers need to solve, while mapping constructs more effectively to computational resources. The goals of multi-paradigm languages are to increase productivity, ensure quality, and take advantage of more sophisticated performance optimizations available in modern hardware.

The proliferation of multicore processors has created additional pressure to improve parallel programming. Multicore is here to stay because of stagnating clock rates and saturated power budgets [4]. Standard desktop PCs are truly parallel machines with 4-core or 8-core processors, while servers have processors with 12, 32, or more general-purpose cores. Embedded devices and mobile phones are becoming parallel machines, too. Programmers now need to deal with the additional complexity of parallel programming or miss opportunities for performance on modern hardware.

Advocates of the functional style [5] argue that it is less error-prone and more productive, compared to an imperative style, so it should be used to make parallel programming easier. Advocates of imperative style, by contrast, favor more control to achieve better performance [5]. Earlier empirical studies set up to assess these tradeoffs typically assumed a context that differs from the one today; for example, some studies assume that programmers have to use one style exclusively, others focus on sequential programs, and still others look at highly specific parallel constructs in imperative languages [6, 7, 8, 9, 10, 11, 12, 13].

Today, languages such as Scala [1] and C# [2] allow the combination of functional and imperative programming in the same language, so developers don't need to make an exclusive choice. However, new problems arise as it is largely unclear how programmers apply mixed programming styles in larger projects. We lack empirical evidence from controlled studies to quantify the software engineering benefits, to identify potential problems, to evaluate which language features are most promising to extend, and how to build tools. As Scala compiles to Java bytecode, program performance can now easily be compared on the same programming task and multicore environment while measuring relevant software en-

gineering metrics in both languages; such comparisons were difficult to set up in the past.

To our knowledge, this is the first paper to answer key questions in a multicore context with Scala and Java, such as: Who needs more effort? How do programmers make progress in parallelization? Whose code is more compact? How are functional and imperative styles used? Who has the best performance? How satisfied are programmers? To provide answers, we study thirteen subjects, each of whom wrote three programs in each of Scala and Java, resulting in 39 Scala programs and 39 Java programs. The main object of study was the parallelization of real-world VLSI CAD tool used in chip design. Our study is based on a counter-balanced within-subjects design (see Section 4.2.2 and [14]), but also applies case study and interview techniques [15, 16] to generate insights that explain phenomena observed in the aggregated statistics. In addition, an Oracle software engineer worked on the same project in Scala and provided reference data. The measured effects are very strong and confirm with statistical significance that Scala code is more compact than Java code. However, our data clearly refutes other claims of Scala on lower project effort and testing and debugging effort. The lessons learned for the improvement of Scala and Java are nevertheless invaluable and show that multi-paradigm languages are worth pursuing.

The paper is organized as follows. Section 2 outlines multicore programming in Scala versus Java. Section 3 presents claims from the literature that form the hypotheses on how Scala's approach aims to improve Java imperative parallel programming. Section 4 details our study design. Sections 5–11 elaborate critical questions addressed in this study, such as effort, parallelization progress, code compactness, programming style, performance, and programmer feedback. Section 12 discusses threats to validity. Section 13 contrasts related work. Section 14 provides a conclusion.

# 2. MULTICORE PROGRAMMING IN SCALA VS. JAVA

Scala [1] (scalable language) is a statically typed, multi-paradigm language that compiles to bytecode on the regular Java virtual machine. A complete overview of Scala is beyond the scope of this paper. We therefore outline some key principles to set the discussion in this paper. To facilitate reading, key principles are described via examples. For further details we refer to [1, 17, 18].

## 2.1 Parallel Programming Example

Consider the well-known producer-consumer pattern [19] that is frequently used in pipelined computations [20]. Listing 1 shows what programmers would typically have to do in Java: create a shared queue, create threads that access the queue, synchronize accesses to the queue, and use *wait* and *notify* signals to let waiting producers or consumers know about an empty or full queue. Advanced programmers who aim to achieve better performance would also use explicit locks instead of shown `synchronized` blocks.

By contrast, Listing 2 shows a Scala outline with actors that run concurrently. Actors communicate based on message passing, and each actor implements send and receive operations. In Listing 2, the `case class` construct allows automatic matches of received `items` based on their type and values. Scala's actor model is implemented on top of Java's shared memory model. Scala therefore exposes programmers to different concurrent abstractions but eventually the compiler translates them into Java bytecode.

## 2.2 Functional and Imperative Programming Example

Scala integrates functional programming with object-oriented imperative programming. It supports higher-order functions, currying, algebraic data types, and native support of sequences, such as lists or sets. For example, everything is an object, and even "`1+2`" would be treated as two `Int` objects `1` and `2`, where the addition is a call to a method of object `1` named "`+`". As another example, consider the expression [1]:

```
numbersList.filter((x: Int) => x>0)
```

The expression uses the function `(x: Int)` with the body `x>0` to obtain all number objects of `numbersList` that are greater than zero, taking advantage of the `filter` method that is provided for all collection object types. Note that it is not necessary to write a `for` loop that iterates over all objects to check each one for the desired property.

Native frameworks provide frequently used data structures as mutable or immutable types. In addition, Scala offers automatic type inference which aims to make coding faster.

In Scala, programmers do not have to make an exclusive choice for functional programming but can program in an imperative style as well. For example, developers can use explicit object definitions, `while` loops, shared state, and reusing Java code from existing packages, such as the *java.util.concurrent* package.

Today, companies such as Twitter [21] employ Scala. Scala's unique features promise to make parallel software development easier. As Scala's usage continues to increase, programming effort and other aspects merit a thorough empirical analysis such as the one in this paper.

# 3. HYPOTHESES ON SCALA IN THE LITERATURE

Scala's combination of functional and imperative programming is claimed to have advantages in comparison to Java [1, 17, 18], but there is little evidence from controlled studies. We summarize important propositions as a motivation for a more thorough empirical examination.

### *Effort.*

"Scala's functional programming constructs make it easy to build interesting things quickly from simple parts" [1, p. 3]. "Scala is easy to get into" [1, p. 3]. The language constructs help programmers get started quickly [1, p. 5]. The combination of functional and object-oriented constructs have "complementary strengths" which lead to "a legible and concise programming style" [1, p. 3]. Programmers require less effort for reading and understanding Scala programs [1, p. 13]. For parallel programs, programmers tend to find Scala's shared-nothing message passing model "much easier to reason about" than Java's shared-memory model with locks [1, p. 584].

### *Code Compactness.*

"Scala programs tend to be short"; in conservative cases "a typical Scala program should have about half the number of lines of the same program written in Java" [1, p. 13]. In extreme cases Scala programs may have one tenth of the lines of code (LOC) of corresponding Java program [1, p. 13]. Scala programs are more concise due to type inference [1, p. 17], optional semicolons. [1, p. 14], control abstractions that avoid duplication [1, p.16]. High-level data structures can be queried through predicates [1, p. 15]. "Scala's syntax avoids some of the boilerplate that burdens Java programs" [1, p. 14].

```
Queue<Item> sharedQueue = ...;
// Thread 1: consumer thread
synchronized(sharedQueue) {
  while(sharedQueue.size() == 0) { sharedQueue.wait(); }
  Item item  = sharedQueue.get();
  // handle item
}
// Thread 2: producer thread
for(...){Item item=createItem();//continuously create items
  synchronized(sharedQueue) {
    sharedQueue.put(item); sharedQueue.notifyAll();
}}
```

**Listing 1: Producer consumer pattern in Java**

```
case class Item(...)
// Actor 1: consumer actor
val consumer = actor { react {
  case Item(...) => // handle item
}}
// Actor 2: producer actor
val producer = actor { for(...) {
  val item = createItem() // continuously produce items
  consumer ! item // send item to consumer
}}
```

**Listing 2: Producer consumer pattern in Scala**

*Parallel Programming.*
Actors are easier to work with than Java's native style with locks [1, p. 583]. Java's concurrency support is sufficient, but "difficult to get right in practice as programs get larger and more complex" [1, p. 583].

*Debugging.*
Scala is less error prone than Java, as Scala programs with fewer lines of code are assumed to have fewer possibilities for defects [1, p. 13–14]. Actors help avoid deadlocks and race conditions [1, p. 584, 616].

# 4. DESIGN OF THE EMPIRICAL STUDY

To validate the aforementioned claims, we study thirteen subjects who worked individually on two Scala and two Java projects during a training phase and afterwards on the actual object of study, which consists of one Scala project and one Java project extending a real-world application. All projects require subjects to create bug-free and well-performing parallel applications. Additional data was provided by an Oracle software engineer who agreed to work on the same project in Scala, and who was already familiar with the algorithms.

## 4.1 Preparations

The subjects are thirteen Master's students close to their graduation who are on average in their fourth year of Computer Science studies. Subjects had appropriate previous knowledge from prerequisite courses in software engineering (e.g., programming languages, patterns, development environments) and parallel programming (e.g., programming with shared-memory and message-passing).

Prior to the study, we conducted a feasibility study to ensure that the assignments have a solution, i.e., there are working parallelization strategies that are feasible to complete in the given period of time. In addition, the Oracle software engineer created a parallel Scala program based on the requirements of our project.

## 4.2 A Two-Phase Approach

The approach applied in this study consists of two phases where the subjects were asked to program in Java and Scala.

### 4.2.1 Phase One: Training

Initially, all subjects received the same training in programming with Java and Scala, which took four weeks. The Java training covered parallel programming with shared-memory. The Scala training included functional programming and parallel programing with actors ([1, 17, 18] were required reading). In addition, everyone was trained and tested on how to use development environments, how to debug, and how to conduct performance analyses for parallel programs. Every subject successfully delivered a working parallel implementation of the Dining Philosophers [22] and mergesort [23] algorithm both in Java and Scala (i.e., we obtained 13*2=26 Java programs and 26 Scala programs). The delivered code was used to assess how subjects understood and employed the programming concepts of Java and Scala. At the end of phase 1, everyone passed and was ready to work on a larger project. In addition, we measured the level of proficiency; a Java pretest classified seven subjects as experts and six as beginners. A Scala pretest classified seven subjects as experts, and six as beginners.

### 4.2.2 Phase Two: Industry Project

This phase focuses on the actual object of study, which is how programmers use Scala and Java in a larger and more complex parallel application. We employ a counter-balanced within-subjects design in which six randomly chosen subjects are tasked to complete a four-week project in Java first whereas the other seven have to do it in Scala (phase 2a). In another four weeks, the subjects have to deliver another parallel program for the same specification, but this time the seven subjects who started with Scala switch to Java and vice-versa (phase 2b). The subjects were unaware and were initially told that they would work on two different projects.

This approach is frequently employed to offset learning and ordering effects when aggregating results [14, 24, 25].

A competition was set up among the Java teams and Scala teams in both phases 2a and 2b, with the goal of achieving the best-performing parallel program for the given specification, input benchmark, and multicore machine. The competition not only motivated subjects to achieve their best individual result, but also reduced the incentive to collaborate (which was not allowed anyway). We also disallowed direct code reuse from the previous project and allowed using just the standard libraries and parallel constructs that come with Scala and Java (e.g., *java.util.concurrent*).

The requirements for the project were designed in collaboration with Oracle as an industrial partner. The setting provides a realistic and representative object of study that goes beyond a toy program. In particular, the Electric VLSI Design System [26] developed at Oracle Labs was used. Electric is an Open Source VLSI CAD application for the custom VLSI designs completely written in Java. Among all possible CAD tools available in Electric, the analysis tool known as DRC (Design Rules Checker) was chosen as a performance-critical parallelizable task. A design rule specifies certain geometric and connectivity restrictions to ensure sufficient margins to account for variability in the fabrication process. Basic design rules range from one layer, e.g., width, area or spacing, to multiple layer rules, such as enclosure. Due to time constraints and the complexity of dealing with all DRC rules involved in modern technologies, subjects were asked to parallelize the minimum area checking algorithm. This algorithm ensures that manufacturers do not print circuits in resolutions that are too small for a given technology and minimum rules might need to be satisfied for each layer of a chip. To facilitate the study, Electric developers offered standardized APIs for Java and Scala to create extensions for the DRC tool already available in Electric.

General literature on design rules checking (e.g., [27]) was handed out to subjects in the first week of the study, to give subjects enough time to familiarize themselves with the problem. At the start of phase 2, subjects were given a seventeen page document with more precise project and algorithm specifications. This was accompanied by a tutorial that described the problem, examples, APIs, coding guidelines, and instructions about data structures to use. Everyone received support to set up the working environment, understand boilerplate code, and compile dummy projects. Questions were answered by instructors and Oracle employees. No one had problems understanding the assignment or working in the programming environment.

The compulsory project specifications channel potential solutions into a certain range, as assessed in our feasibility study. They ensure that the submitted programs and results do not differ because subjects employ widely diversified algorithmic strategies and data structures. Briefly, our algorithm uses a list of bit sets to merge adjacent boundaries of polygons of a metal layer and to ensure that the areas of all flattened polygons satisfy the minimum area rule.

## 4.3 Sources of Evidence and Evaluation

Throughout the study, we collect evidence from several sources: (1) Weekly code submissions. (2) Weekly semi-structured interviews with every subject. (3) Student diaries and final project reports (delivered after the study). (4) Time report sheets on a daily basis on which students tracked the hours spent on various software engineering task categories (e.g., design, implementation, testing). The sheets were cross-checked with our interviews and code inspections for validity. (5) Questionnaires after the completion of each programming project (phase 1, 2a, 2b) captured feedback.

We employ statistics, case study techniques, survey techniques, and interview techniques [14, 15, 16, 24, 25] to extract the lessons learned from this study. For presentation, we aggregate most of the quantitative data into box-and-whiskers plots: lower and upper box boundaries denote lower and upper quartiles of data (visualizing variability), a horizontal line within the box marks the median, whiskers mark 1.5 times the interquartile range on both box ends, and data exceeding the whisker range is marked as outliers (depicted as circles). To gain further support beyond what is visible in the plots, we also apply where possible Wilcoxon's rank sum test [28] paired on subject results. Informally speaking, this non-parametric test evaluates whether two populations differ with statistical significance (see [28] for details). We also apply other tests for cross-checks (non-parametric and parametric, if data distributions allow it), but typically omit their presentation to avoid overload. The obtained p-values appear in the respective graphs. Low p-values mean that there is a low probability that the observed differences are accidental. As in other similar studies (e.g., [6]) we interpret $p \leq 0.05$ as a strong indication for a difference, which degrades as p increases; $p > 0.1$ is the threshold where the difference becomes insignificant.

# 5. WHO NEEDS MORE EFFORT?

Our data reveals that completing the project in Scala required more effort than in Java. As an overview, Figure 1 shows the person hours required for the Scala and Java projects, each sorted in descending order of effort.

Figure 2 illustrates the aggregated statistics. The median effort is 56 hours for Scala and 43 hours for Java (13 hours difference). The mean effort is 72 hours for Scala and 52 hours for Java, which means that on average it takes 20 hours (38%) longer to complete the Scala project. The populations differ significantly with $p = 0.059$.

As the data collection assigned person hours to particular categories, we are able to provide additional details in Figure 3, which shows how much of the implementation time subjects spent working mostly on sequential code or parallel code. In Figure 3(a), the median time for parallel coding in Scala is 14 hours (mean 18 hours) and 11 hours in Java (mean 12 hours), and there is weak statistical support for the difference with $p = 0.08$. For time spent on sequential code in Figure 3(b), the median in Scala is 8 hours (mean 8 hours) and 4 hours in Java (mean 6 hours), however, the difference is insignificant ($p = 0.45$). The most significant difference is due to testing and debugging effort in Figure 3(c): the median in Scala is 20 hours (mean 23 hours), the median in Java is 10 hours (mean 14 hours), a clear difference supported by $p = 0.041$.

As a comparison for the project effort in Scala, the Oracle software engineer spent about 18 hours to create a sequential Scala program on the same specification. He spent about 72 hours (which happens to be the mean effort of our Scala subjects) to create a parallel version; out of this time, he spent 10 hours on testing and debugging (13 hours less than our subjects' mean).

Programmer skills (as determined in our pretest, see Section 4.2) influence how each subject ranks in terms of effort, but the aggregated statistics balance out this effect because there are roughly equally many experts and beginners in both Scala and Java. We conducted a multivariate analysis of variance (MANOVA [29]) that analyzes the impact of Java and Scala skills (beginner/expert) on the Java and Scala effort of

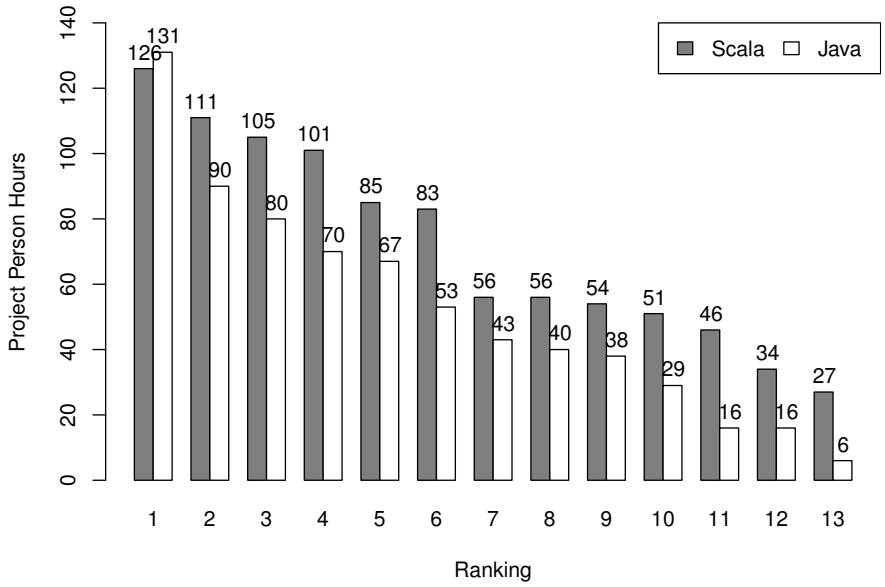


**Figure 1: Effort required to complete the project in Scala and in Java, sorted in descending order.**

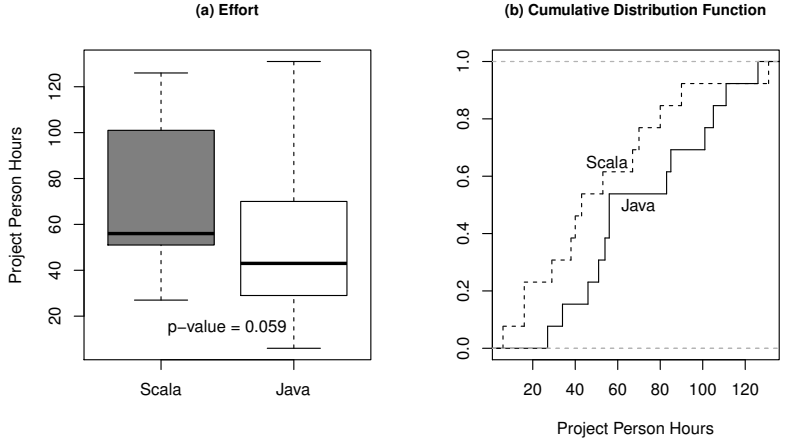


**Figure 2: Aggregated effort statistics show a significant difference between Scala and Java.**

each subject (the analysis was applicable in our case because the Box-test on equality of covariance matrices was insignificant [29]). Results show that expert skills lead to lower effort in comparison to beginners (p=0.05 for Java expertise and p=0.02 for Scala expertise). The analysis on how skills affect parallel implementation time also reveals a combined influence, i.e., that the interaction of Java skills and Scala skills together affects the parallel implementation time ($p = 0.08$). By contrast, it is remarkable to observe that Java and Scala skills do not have a significant influence on testing and debugging time ($p > 0.1$), which suggests that this big difference has nothing to do with skills.

Explanations for the difference in testing and debugging effort come from our interviews and code inspections. One of the main reasons why such effort is higher in Scala is because type system features that actually aim to make programming more productive turn out to make debugging more difficult. In particular, subjects complained that the automated type inference required them to spend more time to understand which actual type each object has when errors are encountered, and they were unsatisfied with tool support on this issue. In addition, automatic object creation and copying was another feature that required more time to track errors and optimize performance.

# 6. WHO HAS THE FIRST PARALLEL PROGRAM?

Java programmers were the first to have a working parallel program. As a measurement of parallelization progress, we tracked the week when each subject had the first working parallel program, based on code inspections and interviews. Figure 4 illustrates that all subjects submitted a parallel version by the deadline of the project. In the week before the deadline, an equal number of parallel Scala and Java programs (69%) worked.

The chart also reveals, however, that no one had working parallel Scala programs until the third project week, even

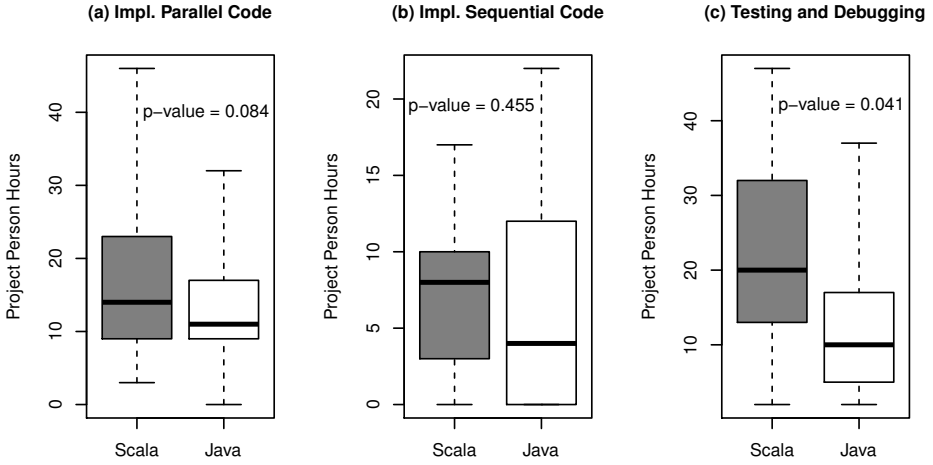


**Figure 3: Effort for all Scala projects split up into implementing sequential code, parallel code, and testing and debugging.**

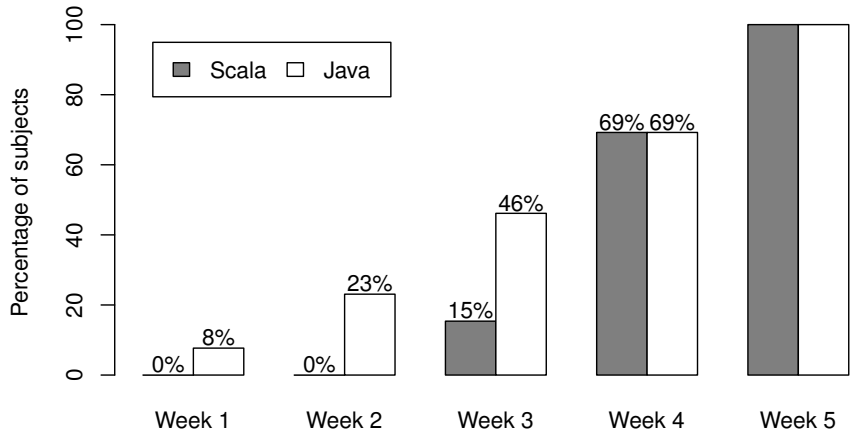


**Figure 4: Percentage of subjects who had a working parallel program in a particular project week (week 5 represents submitted programs after the 4-week deadline).**

though 23% of parallel Java programs were already working in the second week. Interview data suggests that subjects needed time to figure out exactly how to take advantage of the functional style in their particular program. The results also match the effort observations in Section 5. As soon the problems were overcome, increases in working parallel Scala programs were much steeper than for Java. This observation suggests that Scala is powerful because everyone was still able to make the deadline, but it takes time to understand how to exploit its power.

## 7. WHOSE CODE IS MORE COMPACT?

One of Scala's claims is that a Scala program needs fewer lines of code compared to a similar Java program. Our results support this claim in our project context.

Figure 5 summarizes the lines of code (LOC) of all Scala and Java programs as well as their number of characters, excluding comments and blank lines. Scala has 533 median LOC (mean 536) and Java 547 median LOC (mean 632), but the overall box and whiskers of Scala tend towards lower values. Also, no Scala program is longer than 730 LOC. The paired Wilcoxon rank sum test on each subject's solution shows support ($p = 0.078$) that Scala code is more compact.

Quantitative claims of the literature [1], however, seem overgeneralized and are not supported. In this experiment, Scala programs do not have 50% fewer lines of code compared with their Java counterparts. Figure 5 (a) refutes this claim, revealing only a median difference of 14 LOC (2.6%) and mean difference of 96 LOC (15.2%). Also, the claim that in extreme cases Scala has 10 times less code than Java does not hold for our application. The difference between extremes is 1086 LOC in Java versus 284 LOC in Scala, which is just 3.8 times less.

An additional analysis on the number of characters shows similar trends that Scala programs are more compact than Java programs, but the statistical support is weaker ($p = 0.094$). However, the medians of Scala and Java programs are farther apart for characters than for lines of code.

A final comparison baseline is a sequential and a parallel Scala program that were developed under the same specifica-

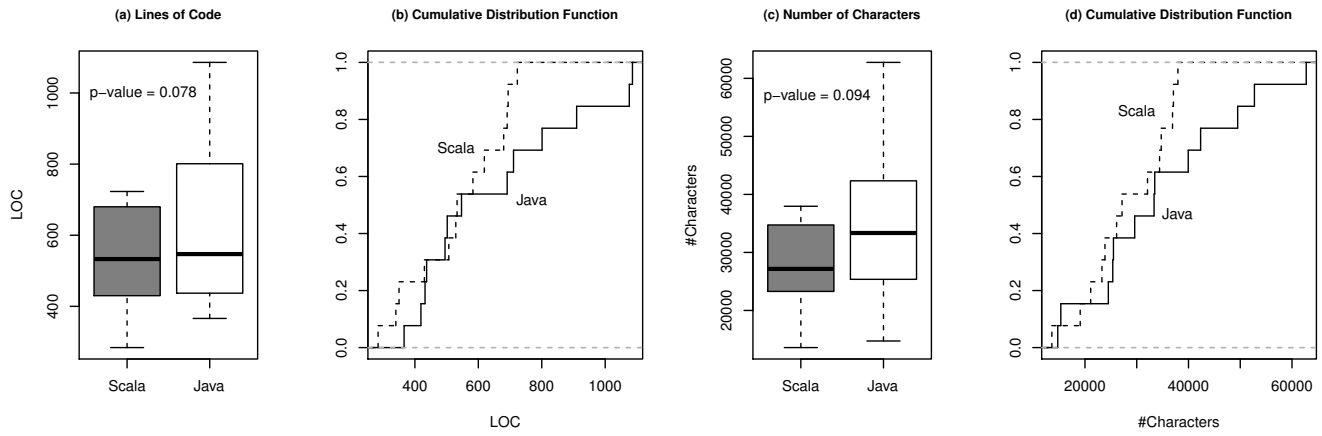


**Figure 5: Code compactness analysis of Scala versus Java.**

tions by the Oracle software engineer. His sequential program has 185 LOC and 3756 characters. His parallel program has 472 LOC (11% less than our subjects' median LOC), which shows that his program is comparable to our subjects' programs. However, his program has 10,186 characters (25% less than our best subject) showing that even more compactness is possible in Scala.

# 8. HOW ARE FUNCTIONAL AND IMPERATIVE STYLES USED?

In a multi-paradigm language like Scala, a question of interest is how subjects actually employ the functional style and imperative style in practice.

We answer this question by analyzing the code of each subject's Scala project. We also provide a comparison with the parallel projects delivered during the training phase (parallel mergesort and parallel dining philosophers).

In particular, we start by classifying key language constructs as belonging to either a typical imperative style or a functional style, according to [1]. For example, `var`, `object`, `array`, `while`, `for`, `abstract`, `import java`, etc. indicate an imperative style. By contrast, constructs such as `val`, `list`, `map`, `filter`, `flatmap`, `foreach`, `:::` (list concatenation), `::` (list cons operator), etc. indicate a functional style. We count the occurrences of all such constructs in each project and calculate the percentage of how many belong to the imperative class and how many belong to the functional class. Figure 6 summarizes the results of this analysis for each subject in the study.

In the DRC project code, Figure 6(a) shows that 8 subjects use more than 50% imperative style (right half of the diagram) and 5 use more than 50% functional style. At the extremes, one subject uses 98% imperative style and one subject 78% functional style.

The project outcomes are roughly similar for the projects of the training phase. For parallel mergesort in Figure 6(b), 5 subjects use more than 50% imperative style. At the extremes, one program uses 88% imperative style and one with 89% functional style. Using functional style in this context is natural because of the algorithm design. For the Dining Philosophers in Figure 6(c), 8 subjects use more than 50% imperative style. At the extremes, one program uses 94% imperative style and one 73% functional style.

By contrast, the sequential DRC project program created by the Oracle software engineer uses 49% imperative style and 51% functional style. His parallel version shifts towards 40% imperative style and 60% functional style.

An interesting insight to note is that many subjects use functional and imperative style in a quite balanced way. However, certain individuals heavily prefer either the functional or imperative style. This preference can be observed quite consistently for both the training projects and the parallel DRC project. However, no subject entirely rejects either style. The data shows that functional programming is indeed useful for realistic parallel programming projects.

# 9. WHO HAS THE BEST PERFORMANCE?

In our study, sequentially executed Scala programs are faster than their Java counterparts. In the parallel case, however, Java programs have better scalability with higher speedups. The fastest run-times are similar for both Scala and Java.

## 9.1 Setup

All DRC project programs are evaluated on a representative input, which consists of a real chip layout that has been successfully taped out in the past. The input file has $2,260,627$ rectangles that are distributed over $74,137$ subcells with a maximum hierarchy depth of 14. The bounding box of the entire chip is $166,946 \times 208,594$ units. We ensured that every program worked correctly on this input (programmers were given the opportunity to fix problems after the deadline, which caused just minimal code changes). All applications are benchmarked on the following machines:

- *4-core machine*: Intel Xeon X5677. This machine has a single-chip architecture with 4 cores, 2 hardware threads per core, 48 GB main memory, and runs RedHat Enterprise Linux 6.0.

- *32-core machine*: Sun SPARC T3-4. This machine has a 4-chip NUMA architecture with 8 cores per chip and 8 hardware threads per core, 256 GB main memory, and runs Solaris 10.

The Scala projects are compiled with Scala 2.8.1 and the Java projects with Java 7. Compiling all Scala projects takes 8 times longer than compiling all Java projects (e.g., on the 4-core machine it took 85 seconds for Scala and 11 seconds for Java).

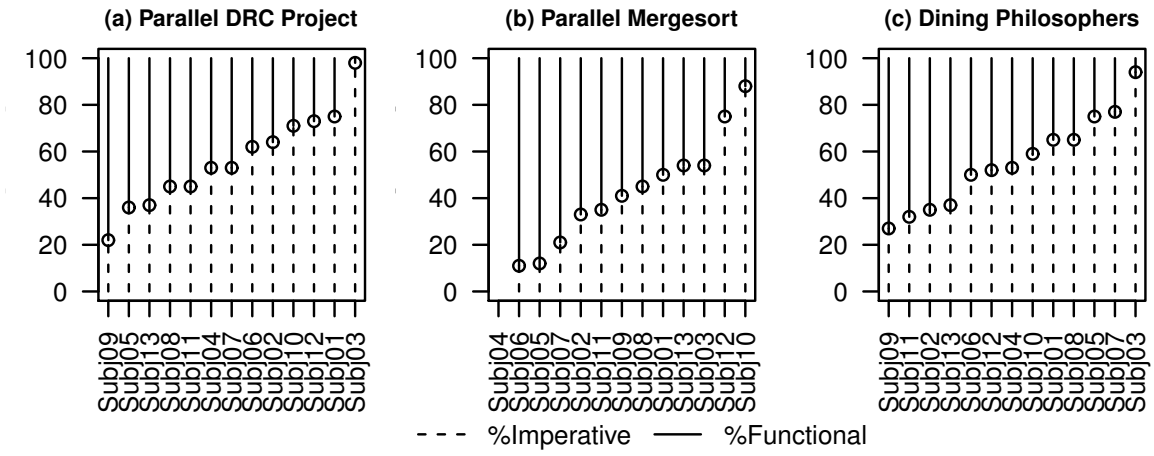


**Figure 6: Percentage of functional and imperative programming styles used by each subject in Scala.**

## 9.2 Measurements

All developers made the number of parallel threads configurable from the command line. Figure 7(a)–(d) summarizes execution times and speedups. Speedup calculations use the execution time with one thread on each machine as a baseline. All parallel Java and Scala programs are executed with 1, 2, 4, 8, 16, 32 threads to test scalability. On the 32-core machine, we added 64 and 128 threads because the hardware offers more parallelism. To avoid bias, each performance data point is an average of 10 runs on each configuration (we remark that the input size is large enough so speedups do not come from data that remains in the cache between runs). Each box plot summarizes thirteen performance data points (one for every subject) for each thread configuration and language.

## 9.3 Results

The measurements in Figures 7(a) and (b) reveal on the 4-core machine that the median execution time of all Scala programs with one thread is 87% better than Java median (see boxes S.1 and J.1 with median difference is 82 seconds). On the 32-core machine, the median execution time of Scala programs with one thread is 22% better than Java (median difference of S.1 and J.1 is 190 seconds). With increased thread count, however, Java programs exhibit better scalability and higher median speedups than the Scala programs.

On the 4-core machine, Figure 7(a) shows that the best Scala runtime is 7 seconds at 4 threads. The best Java runtime is 4 seconds at 8 threads, i.e., Java is 43% faster in the best case. However, the median runtime over all thread counts is 83 seconds for Scala and 98 seconds for Java, which shows that the "average" Scala program is 15% faster than the "average" Java program.

On the 32-core machine, Figure 7(b) shows that the best achieved Scala runtime is 34 seconds at 64 threads. The best Java runtime is close with 37 seconds at 128 threads, so Java is only 9% worse. The median runtime is 466 seconds for Scala and 576 seconds for Java, so the "average" Java runtime is 24% worse than Scala.

By contrast, the Oracle software engineer achieved the following results with his parallel Scala program: The best time on the 4-core machine was 7 seconds at 8 threads (speedup 3.6), which was the same as the best result in our study. His best Scala time on the 32-core machine was 32 seconds at 64 threads (speedup 11.3), which is 6% better than our best result. These numbers match the performance of our top subjects very well.

### 9.3.1 *Does functional programming style lead to slowdowns?*

The programmers that ranked as the top three performers on the 4-core machine used 47%, 55%, 38% functional style in their programs. On the 32-core machine, the top three performers used 47%, 55%, 64% functional style. The program of the programmer that used 2% functional style (i.e., 98% imperative style), which had the least functional style of all Scala programs, ranked in the worst three performers on both machines.

These empirical results show that a functional programming style does not need to harm performance. At the same time they provide support for the promise of multi-paradigm languages by showing that it is possible to do automated performance tuning under the hood, rather than requiring programmers to optimize everything by hand. However, the results also show the need for a combination of functional style and imperative style, as no top performer used functional style exclusively. Our data thus solidifies the ground for language designers, compiler writers, and tool developers that the multi-paradigm direction merits more investigation.

# 10. PROGRAMMER SURVEY FEEDBACK

Programmer feedback collected at the end of the study provides additional insight into the numbers presented so far. The majority of the questions had a five-level Likert scale [14] (ranging from "strongly disagree" to "strongly agree"). We provide a summarized interpretation and mention the percentage of subjects in favor or against a statement (aggregating "agree" and "strongly agree" as "agree", and "disagree" and "strongly disagree" as "disagree").

*Scala type system.*

Letting the compiler implicitly derive the types of variables can become a problem during debugging. While 46% of the subjects agreed that this feature was helpful when writing code, 85% of the subjects agreed that it leads to programming errors.

*Learning and code understanding.*

Programmers found Java programs easier to understand than Scala programs (77% say understanding Java programs

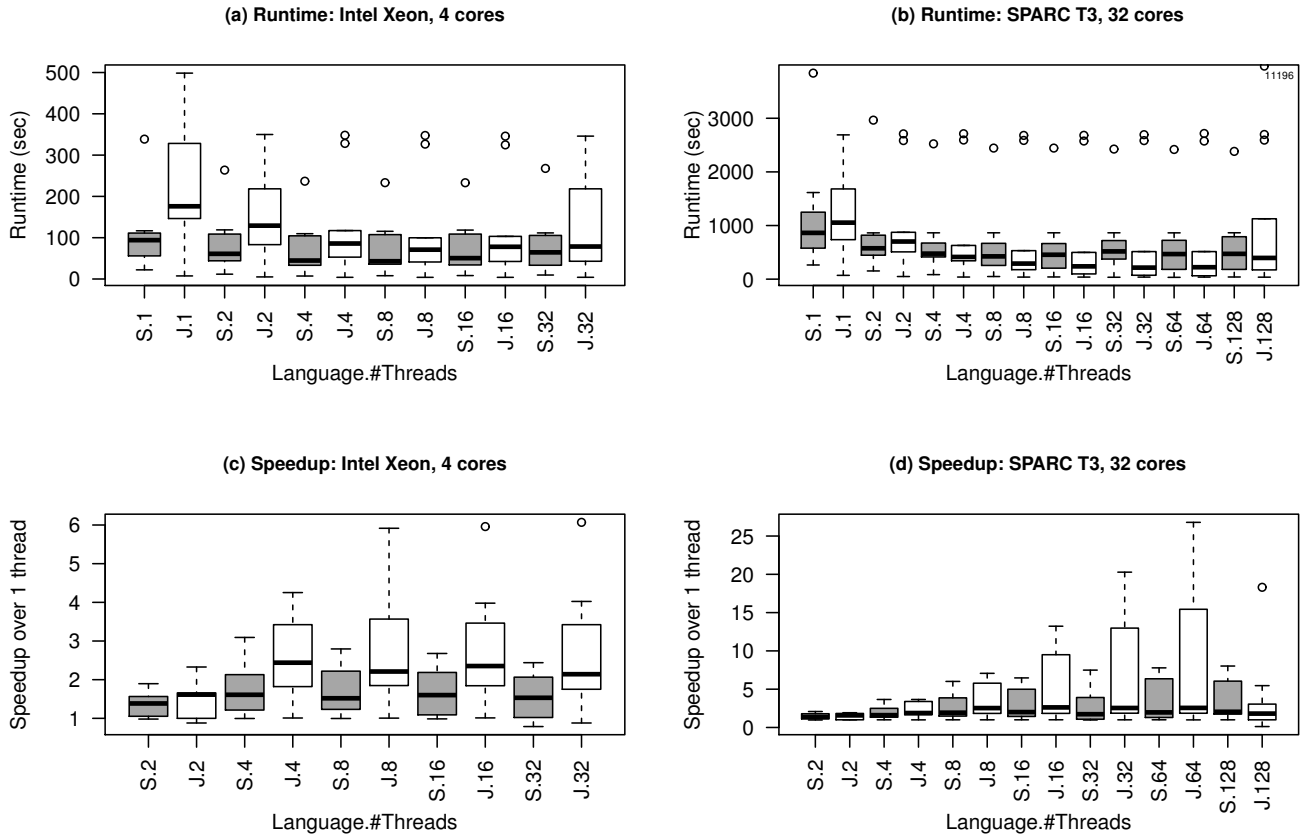


**Figure 7: Performance overview of all Scala and Java project programs for a varying number of threads.**

is easy, compared to 46% for Scala). Java syntax was perceived to be easier than Scala (92% agree that Java syntax is simple, 62% agree that Scala syntax is simple). Only 30% say that adapting to Scala's programming model was easy, compared to 100% for Java. Concerning parallelism, there is an opposite perception: 46% agree that using Java parallel constructs is easy, compared to 92% who say that using Scala parallel constructs is easy. Also, 62% say that Scala parallel construct usage was easy to remember, while 54% say that Java parallel construct usage was easy to remember. These responses suggest that Scala in general is not perceived to be easier than Java, but that the subjects felt like there are advantages for the understanding of parallel programs.

### *Tool support.*

Tool support for Scala needs further improvements. Just one subject said that tool support for Scala is good, compared to 77% for Java. Only 30% of subjects are satisfied with Scala IDE support, compared to 69% for Java.

### *Satisfaction.*

The answers show that most programmers have a positive attitude towards Scala and Java. 54% of the subjects agree that it is a pleasure to use Scala, compared to 69% for Java. There are 77% who say they will use Scala again, and 92% said they will use Java again. Concerning the programming style, 30% agree that functional programming is frustrating, whereas 46% disagree.

### *Perceived productivity.*

Programmers feel productive in both languages but complain about the Scala documentation. 92% of subjects feel productive in Scala, compared to 100% in Java. Just one programmer, however, agrees that the available Scala documentation is good, compared to 100% for Java.

### *Parallelism.*

Scala parallel programming with actors is perceived to be easier than shared-memory programming in Java, which is supported by 38% of programmers in Scala versus 23% in Java. However, most programmers said they tried to postpone parallelization work in both languages (77% in Scala versus 69% in Java). Programmers also seem to have more problems in Scala because 85% say they were afraid of breaking a working program by using additional Scala parallel constructs, compared to 69% for Java.

### *Errors.*

Race conditions are ranked the most important encountered error in both Scala and Java. In addition, debugging a multi-paradigm language such as Scala seems to be a major problem. Just 23% of programmers say that debugging Scala programs is easy, compared to 77% for Java. Consequently, we need better techniques and tools.

### *Composition.*

To handle complex programs, 69% of subjects say that Scala programs are easy to compose from simpler parts, compared to 54% for Java. Scala obviously offered advantages to

handle data structures, as 46% of programmers agree that applying operations on data structures was more flexible in Scala, compared to 15% in Java.

# 11. CODE INSPECTION AND INTERVIEW INSIGHTS

We present a summary of relevant insights that we gained from code inspections and interviews.

## 11.1 Performance

Causes for bad Scala performance often involved immutable data structures, which were typically used because they are thread-safe [30]. Subjects tended to overlook that if such data structures require updates, an implicit copying of objects was triggered under the hood.

## 11.2 Errors

Race conditions were the major cause for progress delays for both Scala and Java. Typically there was one major defect that caused the most effort to find. For example, one subject reported that he took 16 hours to find a race condition involving access to a Scala collection that concatenated two immutable data structures and 1 hour to fix it. Another Scala programmer spent 20% of his project time on debugging to track a race condition that broke a fork-join parallelization pattern.

In Java, some subjects assumed that concurrent reading of shared state does not require synchronization. Others accidentally used multiple lock objects where they should have used just one. Still others used flawed double-checked locking patterns [31]. Even though the training phase addressed exactly these issues and all subjects were able to handle them on smaller tests, it appears that humans need more tool support when projects get more complex in practice.

## 11.3 Functional style and type system issues

The functional code of some subjects was difficult to understand because they used functions with side effects that were not obvious. Implicit type conversions were powerful in saving code, but turned out to make code understanding more difficult and increased errors. When reviewing code, subjects reported that they had to spend a lot of time understanding return types of functions, which was not trivial for larger functions.

# 12. THREATS TO VALIDITY

Every empirical study, including this one, has limitations. In this study, all subjects developed a particular application that was our main object of study. To construct internal validity, we have carefully chosen the application to be representative. It is possible that results such as effort or performance differ for other applications and other hardware. Based on our feasibility studies and experience, we are confident however, that most of the issues and problems encountered in this study will also be encountered in similar parallel applications. The effects we measured were so strong that they became statistically apparent for the number of subjects participating in our study. It is possible that other subjects will obtain different results. However, the study design aimed to reduce bias by using randomized assignment of subjects to projects, training everyone in the same way, and using counter-balancing to cancel out ordering and learning effects. The skill levels measured in our own pre-tests revealed a balanced number of experts and beginners. Data collection was done in a systematically planned and consistent way. We used several sources of evidence to reduce potential bias. All data, including effort, surveys, and interviews, were reported individually at regular intervals. We cross-checked all data for plausibility and compared student reports with the delivered code and interview statements. Student statements were honest and matched the overall profiles and their history.

To validate our comparisons and create external validity, an Oracle software engineer agreed to work on the same project in Scala. As discussed throughout the paper, the results show that our data is within similar ranges as for an industry professional.

# 13. RELATED WORK

Functional and imperative languages have a long tradition. An individual comparison is beyond the scope of this paper, so we refer to [5] for a survey in the context of concurrency.

Empirical studies in multicore software engineering are scarce. Recent studies compare Transactional Memory and locks [9], Pthreads and OpenMP [10, 11], MPI and OpenMP [12]. However, empirical studies directly comparing functional versus imperative programming on today's multicore platforms have received little attention so far. The study of [8] on SML versus C++ done 15 years ago was not conducted with a multi-paradigm language on multicore and unfortunately had implementations that were difficult to compare; however, that study reports similar results to ours that subjects need more effort to test functional code as opposed to imperative code. An experience report on using the OCaml functional object-oriented language on a server application has been published by [7], however, it is not a controlled study with several subjects. The study of [32] focuses on type systems and finds a null result for the use of static type systems on development time. Experiences about how Scala has been recently used at Twitter appeared in [21]. In [33], a very small benchmark (loop recognition algorithm) is loosely compared in C++, Java, Go, and Scala in a non-controlled study. The report focuses on code and performance comparisons and concludes that Java could be about 30% faster than Scala if garbage collection inefficiencies would be fixed. Other studies such as [13] focus on the high performance computing domain where application requirements are largely different. Yet other studies such as [34] analyze the impact of team-level metrics on product-level software metrics. The work of [6] compares programming languages based on a sequential application with respect to general metrics like performance or lines of code.

# 14. CONCLUSION

Multicore hardware is ubiquitous, and software engineering has to catch up. Multi-paradigm languages such as Scala promise to alleviate the tough parallel programming problems that developers are facing today by combining functional and imperative programming styles. Our data reinforces that this direction deserves more investigation. Results show that Scala code is indeed more compact than Java code. Scala application performance is also comparable to Java. Results also show that a functional programming style does not have to lead to bad performance. The top-performing programmers wrote about half their programs in a functional style and the other half in an imperative style. As no top-performing programmer used functional style exclusively, our setting shows that there is a practical need to provide support for both styles. With respect to effort, this study refutes the claim that Scala programs are faster to develop: In comparison to Java, Scala requires more effort and especially more testing and debugging effort. Scala programmers also lagged

behind Java programmers to obtain the first working parallel applications. Programmer feedback in this study does not show that Scala programs are easier to understand than Java programs, but we track the reasons down to the more complex type system. The type system aims to speed up coding and make programs more compact but significantly complicates the reading and debugging process. We need to address these issues better in the future to make programming in the multicore era easier.

**Acknowledgements.** We thank Jochen Huck for organizational support during the study. At Oracle Labs, we thank Victor Luchangco, David Chase, Steven Rubin, Mark Moir, and Guy Steele for excellent feedback on earlier drafts of this paper. Dmitry Nadezhin provided us with the Minimum Area API in Electric and his Scala implementation that served as a comparison baseline.

## 15. REFERENCES

[1] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*, 1st ed. Artima, 2007.

[2] *C# Language Specification v. 3.0.* Microsoft Corporation, 2007.

[3] K. Davis and J. Striegnitz, "Multiparadigm programming in object-oriented languages: Current research," in *Proc. ECOOP 2007 LNCS 4906*. Springer, 2008.

[4] S. H. Fuller and L. I. Millett, "Computing performance: Game over or next level?" *IEEE Computer*, vol. 44, no. 1, 2011.

[5] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, pp. 123–169, June 1998.

[6] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23 –29, oct 2000.

[7] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy, "Using functional programming within an industrial product group: perspectives and perceptions," in *Proc. ACM ICFP*, 2010, pp. 87–92.

[8] R. Harrison, L. Smaraweera, M. Dobie, and P. Lewis, "Comparing programming paradigms: an evaluation of functional and object-oriented programs," *Softw. Eng. Journal*, vol. 11, no. 4, pp. 247 –254, jul 1996.

[9] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proc. ACM SPAA*, 2011, pp. 43–52.

[10] V. Pankratius, A. Jannesari, and W. F. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *IEEE Softw.*, vol. 26, pp. 70–77, November 2009.

[11] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy, "Software engineering for multicore systems: an experience report," in *Proc. ACM IWMSE*, 2008, pp. 53–60.

[12] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proc. ACM SC*, 2005.

[13] V. Basili, J. Carver, D. Cruzes, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, "Understanding the high-performance-computing community: A software engineer's perspective," *Software, IEEE*, vol. 25, no. 4, pp. 29 –36, july-aug. 2008.

[14] L. B. Christensen, *Experimental Methodology*, 10th ed. Allyn & Bacon, 2006.

[15] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Emp. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.

[16] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. Sage Publications, Inc, 2002.

[17] M. Odersky *et al.*, "An overview of the Scala programming language (second edition)," EPFL, Tech. Rep. LAMP-REPORT-2006-001, 2006.

[18] M. Schinz and P. Haller, "A Scala tutorial for java programmers," www.scala-lang.org, November 9 2010.

[19] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[20] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[21] M. Eriksen, "Scaling Scala at Twitter," in *Proc. ACM CUFP*, 2010, pp. 8:1–8:1.

[22] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, 2004.

[23] B. Wilkinson and M. Allen, *Parallel Programming*. Prentice Hall, 2004.

[24] N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Kluwer, 2001.

[25] F. Shull, J. Singer, and D. I. Sjoberg, Eds., *Guide to Advanced Empirical Software Engineering*. Springer, 2008.

[26] "Electric," http://www.staticfreesoft.com/index.html, 2011.

[27] G. E. Bier and A. R. Pleszkun, "An algorithm for design rule checking on a multiprocessor," in *Proc. AC DAC*, 1985.

[28] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*. Wiley, 2nd 1999.

[29] "IBM SPSS Statistics Version 19," http://www.ibm.com/software/analytics/spss/, 2010.

[30] J. Bloch, *Effective Java - Programming Language Guide*. Addison Wesley, 2001, pp. 50–56.

[31] D. Bacon *et al.*, "The "double-checked locking is broken" declaration," http://www.cs.umd.edu/∼pugh/java/memoryModel/DoubleCheckedLocking.html, Sep 2011.

[32] S. Hanenberg, "An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time," in *Proc. ACM OOPSLA*, 2010.

[33] R. Hundt, "Loop Recognition in C++/Java/Go/Scala," 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[34] A. Meneely, P. Rotella, and L. Williams, "Does adding manpower also affect quality?: an empirical, longitudinal analysis," in *Proc. ACM SIGSOFT/FSE*, 2011.