The case for the Three R’s of Systems Research:

Repeatability
Reproducibility & Rigor

Jan Vitek

Kalibera, Vitek. Repeatability, Reproducibility, and Rigor in Systems Research. EMSOFT11
In 2006, Potti & Nevins claim they can predict lung cancer.

In 2010, papers retracted, bankruptcy, resignations & investigation.

Bad science ranging from fraud, unsound methods, to off-by-one errors in Excel.

Uncovered by a repetition study conducted by Baggerly & Coombes with access to raw data and 2,000 hours of effort.
Out of 122 papers in ASPLOS, ISMM, PLDI, TACO, TOPLAS

90 evaluated execution time based on experiments

71 of these 90 papers ignored uncertainty
### Table 2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Core 2</th>
<th>Pentium 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>m5</td>
<td>O3</td>
</tr>
<tr>
<td>Operating System</td>
<td>Linux 2.6.25</td>
<td>Linux 2.4.21</td>
</tr>
<tr>
<td>Tool Chain</td>
<td>gcc 4.1.3, icc 10.1</td>
<td>gcc 4.2.1</td>
</tr>
<tr>
<td>Measurement</td>
<td>papi-3.5.1 / perfmon-2.8</td>
<td>papi-3.0.8 / perfctr-5.2.16</td>
</tr>
<tr>
<td>Micro-architecture</td>
<td>Core</td>
<td>NetBurst</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.4 GHz</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>memory</td>
<td>8G</td>
<td>2G</td>
</tr>
<tr>
<td>L1</td>
<td>32K Ins., 32K Data</td>
<td>12K Ins. 8K Data</td>
</tr>
<tr>
<td>L2</td>
<td>128K Unified</td>
<td>512K Unified</td>
</tr>
<tr>
<td>L3</td>
<td>4096K</td>
<td>NA</td>
</tr>
<tr>
<td>TLB entries</td>
<td>512</td>
<td>64</td>
</tr>
</tbody>
</table>

3.3 Following best practices

With all aspects of our measurements we attempted to be as careful as possible. In other words, the measurement bias that we demonstrate later in the paper is present despite our following best practices.

- Except in the experiments where we add environment variables, we conducted our experiments in a minimal environment (i.e., we unset all environment variables that were inessential).
- We conducted all our experiments on minimally-loaded machines, used only local disks, and repeated each experiment multiple times to ensure that our data was representative and repeatable.
- We conducted our experiments on two different sets of hardware and (when possible) one simulator. This way we ensured that our data was not an artifact of the particular machine that we were using.
- Some Linux kernels (e.g., on our Core 2) randomize the starting address of the stack (for security purposes). This feature can make experiments hard to repeat and thus we disabled it for our experiments.

4. Measurement Bias is Significant and Commonplace

This section shows that measurement bias is significant and commonplace. By significant we mean that measurement bias is large enough to lead to incorrect conclusions. By commonplace we mean that it is not an isolated phenomenon but instead occurs for all benchmarks and architectures that we tried.

We quantify measurement bias with respect to the following question: how effective are the O3 optimizations in gcc? By "O3 optimizations" we mean optimizations that O3 introduces (i.e., it does not include optimizations that carry over from O2).

4.1 Measurement bias due to link order

We first show the measurement bias due to link order for all benchmarks and then discuss one potential cause for it on one benchmark.

#### (a) Perlbench

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Link Order</th>
<th>Speedup (O2) / Speedup (O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>default</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>alphabetical</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>random</td>
<td>1.05</td>
</tr>
</tbody>
</table>

#### (b) All Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Link Order</th>
<th>Speedup (O2) / Speedup (O3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>default</td>
<td>0.95</td>
</tr>
<tr>
<td>libquantum</td>
<td>alphabetical</td>
<td>1.00</td>
</tr>
<tr>
<td>perlbench</td>
<td>random</td>
<td>1.10</td>
</tr>
<tr>
<td>bzip2</td>
<td>default</td>
<td>1.00</td>
</tr>
<tr>
<td>h264ref</td>
<td>alphabetical</td>
<td>1.05</td>
</tr>
<tr>
<td>mcf</td>
<td>random</td>
<td>1.10</td>
</tr>
<tr>
<td>gobmk</td>
<td>default</td>
<td>1.00</td>
</tr>
<tr>
<td>hammer</td>
<td>alphabetical</td>
<td>1.05</td>
</tr>
<tr>
<td>sjeng</td>
<td>random</td>
<td>1.10</td>
</tr>
<tr>
<td>sphinx</td>
<td>default</td>
<td>1.00</td>
</tr>
<tr>
<td>milc</td>
<td>alphabetical</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Figure 2. The effect of link order on Core 2.
Out of 122 papers in ASPLOS, ISMM, PLDI, TACO, TOPLAS

90 evaluated execution time based on experiments

71 of these 90 papers ignored uncertainty

This lack of rigor undermines the results

Yet, no equivalent to the Duke Scandal.

Are we better?
Is our research not worth reproducing?
Is our research too hard to reproduce?
Repetition
… re-doing the same experiments on the same system and using the same evaluation method

Reproduction
… independent researcher implements/realizes the published solution from scratch, under new conditions

Is our research hard to repeat?
Is our research hard to reproduce?
Goal

Break new ground in

hard real-time concurrent garbage collection
Aparté

GC in 3 minutes
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Garbage Collection

Phases
- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction
Incrementalizing marking

Collector marks object

Application updates reference field

Compiler inserted write barrier marks object
Incrementalizing compaction

- Forwarding pointers refer to the current version of objects
- Every access must start with a dereference
Obstacles

No real-time benchmarks for GCed languages

No clear competition, two GC algorithms claim to be best

No accepted measurement methodology

No open source experimental platform for comparison
**Step 1**

**Develop an open source experimental platform**

*Picked the Real-time Specification for Java*

**First generation system, about 15 man/years**

*Flew on a Boeing ScanEagle*

**Second generation system, about 6 man/years**

*Competitive with commercial JVMs*

*A Real-time Java Virtual Machine for Avionics. TECS, 2006*
Observations

Results on noncompetitive systems not relevant

Much of work went into a credible research platform

SpecJVM98

- HotSpot 1.6 Server
- IBM J9
- Sun Java RTS 2.1
- IBM Metronome SRT
- Fiji VM CMR
- Fiji VM Schism/cmrc
- Fiji VM Schism/cmra
- Fiji VM Schism/cmrcw

Fastest Fiji collector
Fiji hard RT collector
Worst-case simulators

throughput relative to HotSpot 1.6 server
Step 2

Develop an open source benchmark

Collision Detector Benchmark
In Java, Real-time Java, and C (Linux/RTEMS)

Measure response time, release time jitter

Simulate air traffic control
Hard RT collision detector thread
Scalable stress on garbage collector

About 1.5 man/years

A family of Real-time Java benchmarks. CC:PE 2011
Observation

Understanding what you measure is critical

Running on a real embedded platform and real-time OS, difference between Java & C small…

Good news?

No. The LEON3 lacks a FP unit, & the benchmark is FP intensive...
Step 3

Gain experience with the state of the art

Experiment with different GC techniques

- GC in uncooperative environment
- Brooks forwarding
- Object replication
- Object handles

About 2 man/years

Accurate Garbage Collection in Uncooperative Environments. CC:P&E, 2009
Hierarchical Real-time Garbage Collection. LCTES, 2007
Replicating Real-time Garbage Collector. CC:P&E, 2011
Handles Revisited: Optimising Performance and Memory... ISMM, 2011
Observation

Trust but verify, twice.

From workshop to journal, speed 30% better

Good news?

Later realized switching to GCC 4.4 slowed baseline (GCC didn’t inline a critical function)

Once accounted for this our speed up was 4%…

A correction was issued…
Step 4

Reproduce state of the art algorithms from IBM and Oracle

*Metronome, Sun Java RTS*

Choose measurement methodology

*Existing metric (MMU) inadequate*

About 3 man/years

*Scheduling Real-Time Garbage Collection on Uniprocessors. TOCS 2011*
*Scheduling Hard Real-time Garbage Collection. RTSS 2009*
Observation

Reproduction was difficult because of closed-source implementations & partial description of algorithms

Repetition was impossible because no common platform
Step 5

Develop a novel algorithm

*Fragmentation tolerant*

*Constant-time heap access*

About 0.6 man/years

Schism: Fragmentation-Tolerant Real-Time Garbage Collection. *PLDI 2011*
Schism: objects

- Avoid external fragmentation by splitting objects in 32-byte chunks
Schism: arrays

- For faster array access, array = variable sized spine + 32-byte chunk payload
In summary,
28 m/y reproduction
0.6 m/y novel work

<table>
<thead>
<tr>
<th>Task</th>
<th>Man/Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Experimental platform</td>
<td>21</td>
</tr>
<tr>
<td>Benchmark</td>
<td>2</td>
</tr>
<tr>
<td>Implementing basic techniques</td>
<td>2</td>
</tr>
<tr>
<td>Reproduction of state-of-the-art +measurement methodology</td>
<td>3</td>
</tr>
<tr>
<td>Implementing novel algorithm</td>
<td>0.6</td>
</tr>
</tbody>
</table>
Rigor

Cater for random effects, non-determinism

- Repeat experiment runs, summarize results
- Threat to validity detectable by failure to repeat

Guard against bias

- Use multiple configurations, hardware platforms
- Threat to validity detectable by failure to reproduce

Jain: The Art of Computer Systems Performance Analysis
Lilja: Measuring Computer Performance, A Practitioner’s Guide
Evaluate Collaboratory, http://evaluate.inf.usi.ch/
Repeatability

Enable repetition studies

Archival

Automate and archive

Disclosure

Share experimental details
Reproducibility

Community support for focused reproductions

*Open benchmarks and platforms*

Reward system for reproductions

*Publish reproduction studies*

*Regard them as 1st class publications*
The Correspondence Principle for Idempotent Calculus and some Computer Applications

Grigorii L. Litvinov and Victor P. Maslov

1 Introduction

There is a correspondence between important, useful and interesting constructions and results over the field of real (or complex) numbers and similar constructions and results over idempotent semirings, in the spirit of the Correspondence Principle in Quantum Mechanics. Analogies for some basic ideas, constructions and results in Functional Analysis and Mathematical Physics are discussed from this point of view. Thus the Correspondence Principle is a powerful heuristic tool to apply unexpected analogies and ideas borrowed from different areas of Mathematics and Theoretical Physics.

It is very important that some problems appearing in the traditional context of Functional Analysis can be transferred and then studied (semi-)rigorously in a suitable setting. This linearity conceptually simplifies the explicit construction of solutions. In this case we have a natural analogy of the so-called representation principle in Quantum Mechanics [1] (5).

The theory is well advanced and includes, in particular, some integration theory, some linear algebra, spectral theory and functional analysis. Application domains include, not only the recent development of the fuzzy setting, but also the optimization problems in graph theory, optimization with a large parameter uncertainty problem, optimal design of computer programs and computer media, optimal identification of parallel data processing, dynamic programmin, discrete event systems, computer networks, discrete mathematics, radiography, biology and many other applications of these ideas in mathematical physics and biology [19] [52].

In this paper we present a new approach to developing an approach to object-oriented software and hardware design for algorithms of displacement calculi and accurate calculations. In particular, there is a regular method for constructing such programs and theoretical schemes intended for an implementation of basic algorithms of displacement calculi and mathematical physics

(c) Camil Demetrescu
Key ideas

Artifact
Evaluation
Committee
PhD students
postdocs

Senior co-chairs

(c) Camil Demetrescu
Criteria
Consistent with the Paper

We can turn iron into gold

Paper

Artifact

(c) Camil Demetrescu
Complete
Easy to Reuse vs.
Well Documented
Statistics from OOPSLA²¹³

- 2 AEC co-chairs
- 24 AEC members
- 3 reviews per AEC member
- 3 reviews per artifact

18 accepted
21 artifacts submitted
50 papers accepted

(c) Camil Demetrescu
Title: TreatJS: Higher-Order Contracts for JavaScript (Artifact)

Authors: Matthias Keil and Peter Thiemann

Institute for Computer Science
University of Freiburg
Freiburg, Germany
keilr,thiemann@informatik.uni-freiburg.de

Abstract: TreatJS is a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond providing the standard abstractions for building higher-order contracts (base, function, and object contracts), TreatJS’s novel contributions are its guarantee of non-interacting contract execution, its automatic approach to blame assignment, its support for contracts in the style of unions and intersection types, and its notion of a parameterized contract scope, which is the building block for composable run-time generated contracts that generalize dependent function contracts.

Scope: TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. The library relies on JavaScript promises to guarantee full interoperability for contracts. It further exploits JavaScript’s reflexive feature to run contracts in a sandbox environment, which guarantees that the execution of contract code does not modify the application state. No source code transformation or change in the JavaScript run-time system is required.

Metadata: ACM Subject Classification: D.2.4 Software/Program Verification
Keywords and phrases: Higher-Order Contracts, JavaScript, Promises
Digital Object Identifier: 10.4230/LIPIcs.ESOP.2015.8

http://dx.doi.org/10.4230/LIPIcs.ESOP.2015.8

Related Conference: 29th European Conference on Object-Oriented Programming (ECOOP 2015), July 2-5, 2015, Prague, Czech Republic.

1 Scope
The artifact is designed to support reproducibility of all the experiments of the companion paper, allowing users to test the contract system on a variety of benchmarks. In particular, it allows to include TreatJS in existing JavaScript code, to specify contracts by plain JavaScript functions, to construct contracts by an unrestricted combination of other contracts, and to enforce contracts in all contexts of use.

2 Content
The artifact package includes:
- the main source of TreatJS;
- a set of test cases to examine the feature of the contract system;
- modified version of the Google Octane 2.0 benchmark suite;
- detailed instructions for using the artifact, provided as an index.html file.
TreatJS: Higher-Order Contracts for JavaScript

Matthias Kell and Peter Thiemann

Institute for Computer Science
University of Freiburg
Freiburg, Germany

Abstract
TreatJS is a language embedded, higher-order contract system for JavaScript which enforces contracts by run-time monitoring. Beyond providing the standard abstractions for building higher-order contracts (i.e., function, and object contractual, TreatJS’s novel contributions are its guarantee of non-interfering contract execution, its systematic approach to blame assignment, its support for contracts in the style of union and intersection types, and its notion of a parameterized contract scope, which is the building block for composable run-time generated contracts that generalize dependent function contracts.

TreatJS is implemented as a library so that all aspects of a contract can be specified using the full JavaScript language. The library relies on JavaScript proxies to guarantee full enforcement of contracts. It further exploits JavaScript’s reflective features to run contracts in a sandbox environment, which guarantees that the execution of contract code does not modify the application state. No source code transformation or change in the JavaScript run-time system is required. The impact of contracts on execution speed is evaluated using the Google Octane benchmark.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases: Higher-Order Contracts, JavaScript, TreatJS

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.28

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://doi.org/10.4230/LIPIcs.BAKTIS.1.1.1

1 Introduction
A contract specifies the interface of a software component by stating obligations and benefits for the component’s users. Customary contracts comprise invariants for objects and components as well as pre- and postconditions for individual methods. Prima facie such contracts may be specified using straightforward assertions. But further contract constructions are needed for contemporary languages with first-class functions and other advanced abstractions. These facilities require higher-order contracts as well as ways to dynamically construct contracts that depend on run-time values.

Software contracts were introduced with Meyer’s Design By Contract® methodology [39] that stipulates the specification of contracts for all components of a program and the monitoring of these contracts while the program is running. Since then, the contract idea has taken off and systems for contract monitoring are available for many languages [33, 1, 37, 12, 19, 11, 19] and with a wealth of features [30, 31, 7, 20, 40, 18, 2]. Contracts are particularly important for dynamically typed languages as these languages only provide memory safety and dynamic type safety. Hence, it does not come as a surprise that higher-order contract systems developed for Scheme and Racket [24], and TreatJS need to be implemented under the Creative Commons licence CC-BY (Creative Commons Attribution).
Artifacts as first-class citizens

2015


Conclusions

Develop open source benchmarks

Codify documentation, methodologies & reporting standards

Require executable artifacts

Publish reproduction studies