# A Literate Experimentation Manifesto

Jeremy Singer

University of Glasgow, UK

jeremy.singer@glasgow.ac.uk

## Abstract

This paper proposes a new approach to experimental computer systems research, which we call *Literate Experimentation*. Conventionally, experimental procedure and writeup are divided into distinct phases: i.e. setup (the method), data collection (the results) and analysis (the evaluation of the results). Our concept of a *literate* experiment is to have a single, rich, human-generated, text-based description of a particular experiment, from which can be automatically derived: (1) a summary of the experimental setup to include in the paper; (2) a sequence of executable commands to setup a computer platform ready to perform the actual experiment; (3) the experiment itself, executed on this appropriately configured platform; and, (4) a means of generating results tables and graphs from the experimental output, ready for inclusion in the paper.

Our Literate Experimentation style has largely been inspired by Knuth's Literate Programming philosophy. Effectively, a literate experiment is a small step towards the *executable paper* panacea. In this work, we argue that a literate experimentation approach makes it easier to produce rigorous experimental evaluation papers. We suggest that such papers are more likely to be accepted for publication, due to (a) the imposed uniformity of structure, and (b) the assurance that experimental results are easily reproducible. We present a case study of a prototype literate experiment involving memory management in Jikes RVM.

***Categories and Subject Descriptors*** I.7.2 [*Document and Text Processing*]: Document Preparation—Scripting languages

***General Terms*** Documentation, Experimentation

***Keywords*** literate programming, experimental write-up

[Copyright notice will appear here once 'preprint' option is removed.]

## 1. Introduction

Most Computer Science papers follow one of a standard set of publication design patterns. For example, consider the INCREMENTAL-IMPROVEMENT pattern. The generic layout of such a paper is as follows: Given an existing system Foo, the paper authors present a small (but generally ingenious) modification, Foo+, that improves upon some aspect of the original performance of Foo. In order to convince both the paper reviewers and the eventual readers that Foo+ is indeed a significant advance worthy of publication, the authors provide rigorous experimental evidence to demonstrate the quantitative performance improvements gained by their modification.

This paper proposes that the experimental evaluation sections of INCREMENTAL-IMPROVEMENT papers could be standardized to such an extent that they might be automatically generated, following a script-based approach. We have been inspired by Knuth's philosophy of *literate programming* [15] where the program and its documentation are inter-twined, yet the executable program and the human-readable description can be extracted separately using automated scripting tools. To quote Knuth:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

In the same way, we suggest a new approach to empirical evaluation sections of INCREMENTAL-IMPROVEMENT research, which we call *literate experimentation*. We parallel Knuth's message below:

> Let us change our traditional attitude to the construction of systems software experiments: Rather than creating an experiment to demonstrate system-level improvement, let us concentrate on explaining to human beings how we intend the experiment to operate.

In an instance of a literate experiment, the experimental method (which is executed to generate the results) and the prose description of the experiment (which is included in the final paper) are inter-twined as a single textual source

```
1
2 export N=30
3 #@# We perform the individual experiment \N times
  .
4 for I in `seq 1 $N`
5 do
6     ./run_expt.sh
7 done
```

**Figure 2.** A simple literate experimentation script

file. Scripting tools can automatically configure and execute the experiments to generate a set of results. Subsequently, they can perform data analysis to present the results in an appropriate manner. Additionally, the scripting tools extract the relevant textual descriptions of the experiment to insert into appropriate sections of a paper. Figure 1 shows this workflow in a schematic diagram.

In summary, our prototype approach is to write human-readable descriptions of experiments as special comments within shell scripts. We execute the shell script to run the experiment. We process the same shell script with a simple macro-processor (perl script) to extract the LaTeX fragments for inclusion in the paper. Section 3 gives further technical details about this literate experimentation approach. For now, Figure 2 gives a trivial example script.

### 1.1 Contributions of This Paper

This paper does not follow the INCREMENTAL-IMPROVEMENT design pattern. Instead, it is more of a STEP-CHANGE style of paper. We are proposing a radical methodological shift in the manner in which systems research is published. The advantages of our literate experimentation technique presented in this paper are:

1. The structure of experimental computer systems papers will become more uniform, making them easier to write, review, and assimilate.
2. The experimental method of such systems papers will be clarified, in such a way as to make the results generation process reproducible.
3. Given these two benefits, we feel that the evaluation practice of research performed by the systems community will become more standardized and rigorous.

## 2. Motivation

This section presents the reasons why we consider our proposed literate experimentation approach will be useful for the computer systems research community. In Section 2.1 we give statistics to show that empirical experimentation papers in general (of which INCREMENTAL-IMPROVEMENT papers are a special case) are becoming increasingly common. In Section 2.2 we show that there is a fairly standard structure for such papers. In Section 2.3 we discuss the need (and current lack of provision) for accurately and efficiently reproducing results from such papers.

The rest of this paper demonstrates that these requirements can all be met by the literate experimentation method.

### 2.1 A Popular Paper Genre

A high proportion of computer systems papers that appear in major international conferences and reputable journals are based around the reporting of empirical experimental results. For instance, Shaw [23, 24] gives statistics concerning submitted papers to the 2002 International Conference on Software Engineering (ICSE). She reports that 48% of submissions (42% of accepted papers) were of the class ‘method or means of development’. Further, 14% of submissions (12% of accepted papers) were of the class ‘design, evaluation, or analysis of a particular instance’. Many of these papers fit one of the experimental paper design patterns, such as the INCREMENTAL-IMPROVEMENT pattern we describe in Section 1.

The US-based Committee on Academic Careers for Experimental Computer Scientists reports that experimental computer science has been a growing sub-discipline [7] since the 1970s. Indeed, in 2007 an ACM workshop was entirely devoted to experimental papers [1].

Tichy [26] argues that experimental style papers presenting small-step results are well worth publishing, since they improve understanding and raise new questions. He presents the importance of meaningful benchmarks and validation.

Georges et al [11] analyse 50 papers on Java performance evaluation at major international conferences between 2000 and 2007. Most of these fit into the INCREMENTAL-IMPROVEMENT pattern. Georges et al review and critique the various statistical strategies for performance evaluation used in this corpus of papers. In many cases, they conclude that the strategies are not statistically sound, or rigorous. They offer some high-level advice and ideas for how to evaluate performance in a statistically rigorous way. They make mention of experimental automation, although they are not concerned with automatic paper generation, which is the focus of our work.

The *Evaluate Collaboratory* group[1] campaign for better experimental methods in systems research [9]. Their open letter to conference program chairs states that:

> We believe it is time for Computer Science to become a better empirical science by promoting both observation and reproduction studies. We hope you agree with us. If you disagree, please let us know. If you agree, we request that you actively encourage observation and reproduction papers in your “call for papers”.

[1] `http://evaluate.inf.usi.ch`

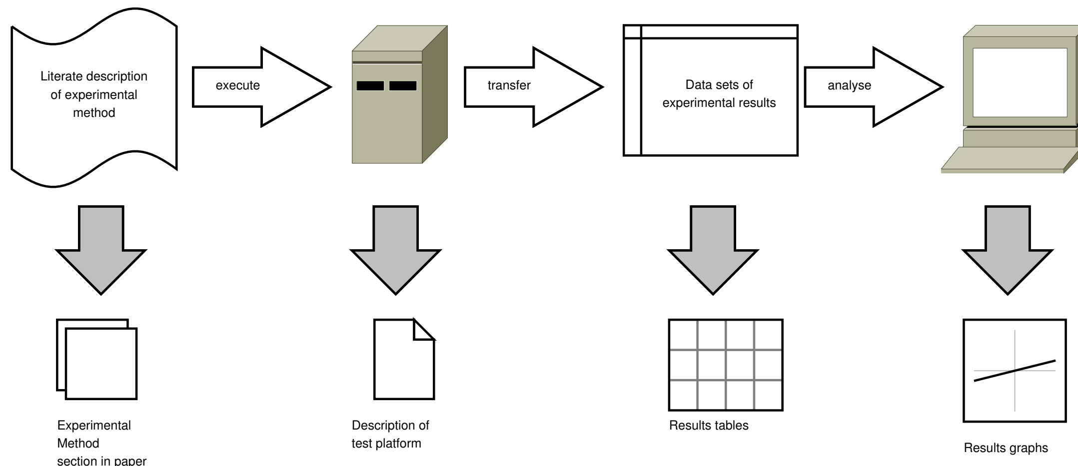


**Figure 1.** Schematic diagram showing how literate experimentation works in practice

Thus it appears that there is a growing interest in experimental computer systems research, backed by a greater community adoption of rigorous scientific techniques.

Thus we infer from the Computer Systems meta-literature, together with our personal experiences, that experimental papers are relatively common, and there is widespread support for improving community practice.

### 2.2 A Standard Paper Structure

A paper that presents experimental results generally follows a fairly standard structure:

1. It initially outlines a well-known and widely available system, either in the **Introduction** or **Background** section.
2. Then it highlights a potential inefficiency in this system, either anecdotally or with some motivating scenarios. This information is generally presented in the **Background** or **Problem Statement** section.
3. Then it outlines a potential solution to remove the inefficiency. This section may be entitled **Solution** or **Proposed Technique**.
4. Next it evaluates this solution empirically using a community-accepted benchmark suite to demonstrate that the presented modification does improve the overall performance of the system. The experimental description is written in a **Methodology** section and the results are reported in an **Evaluation** or **Results** section.

As with all published literature in this field, a certain amount of boilerplate material is required in the **Introduction** and the **Conclusion**. For academic rigour there should also be comprehensive **Discussion** and **Related Work** sections. We accept that human nature means that there will be some natural variety in presentation style and ordering of material. However we feel that for the highest quality papers, the core of an experimental paper will follow the above template, in general.

> We propose that, for a literate experiment, the whole of the **Methodology** section and a large part of the **Evaluation** section could be automatically generated from the literate experiment description.

Although this *de facto* structure is familiar to experienced systems researchers, new people (e.g. PhD students) trying to establish themselves in the field may not have the same awareness of the convention. In other scientific disciplines, the paper structure and experimental evaluation technique is tightly prescribed (e.g. biology [19], medicine [22]). However in Computer Systems research there is generally no such standard structure specification.

Thus, we argue that a key motivation for literate experiments is the *Structural* one. Papers using the literate experimentation technique will follow an explicit standard template, particularly in the **Methodology** and **Evaluation** sections.

### 2.3 A Requirement for Reproducibility

Another recent concern in the community has been the reproducibility of experiments [8]. Elsevier [2] has recently organized a grand challenge for *executable papers*. Here, reproducibility of experimental results is one of the major concerns. In addition, more popular elements of the press give coverage to alternative models of research publication, dissemination and evaluation, supported by the idea of shared experimental data and meta-data in open-access online repositories [6, 14].

In the field of Computer Systems research, more so than in any other scientific discipline, we ought to be able to *manage reproducibility*, since the entire system is under our control. However, often the experiments are so complex [20]

[2] `http://www.executablepapers.com`

that to encapsulate them accurately, completely and engagingly within the publishable paper limits is near-impossible. Paper reviewers, the gate-keepers of academic respectability, are generally unable to verify experiments empirically. Instead they resort to a cursory inspection of the presented results, and have to rely on their judgment as to whether these results are believable. Further down the line, eventual readers of the published papers may attempt to reproduce the results in replica experiments, but are often unable to do so, perhaps because vital experimental details were inadvertently omitted from the paper. Indeed, a whole sub-community[3] has built up around the idea of duplicating or debunking systems results.

Our proposed approach would make it easier to record experimental methodology, and replay it at a later date. This would aid both paper reviewers and general readers in their attempts to reproduce experimental results. Thus another key motivation of literate experimentation is the need to support *experimental authenticity and reproducibility*.

## 3. Proposed Solution

In this section, we outline:

1. the basic components of literate experimentation
2. how to compose components to form a full experimental description
3. how this description must processed in such a way as to:
   (a) perform the experiment,
   (b) recover the experimental artifacts required in the paper.

### 3.1 System Requirements

We make several platform-specific assumptions for our prototype implementation, since our literate experiment components rely on Unix-style shell scripts and LaTeX paper authoring. The shell scripts depend on some standard Unix tools, e.g. bash, perl, wget, sed, cat. Thus our literate experimentation implementation should be compatible with any common Unix variant, e.g. Linux, Mac OS X. Conceivably, the components we have developed could be ported to other platforms, either with alternative implementations, or with advanced scripting conditional tests for platform-dependent options.

We advocate that, when a paper is published online, the literate experiment description should be transmitted, along with the actual paper[4]. This will serve to improve authenticity and reproducibility of published experimental computer systems research.

[3] Workshop on Duplicating, Deconstructing, and Debunking. `http://www.ece.wisc.edu/~wddd`

[4] For this paper, we incorporate the literate experiment script code as Appendix A. Further, a tarball containing the script and supporting files is available from: `http://www.dcs.gla.ac.uk/~jsinger/casestudy.tar.gz`. We hope that more elegant, integrated, mechanisms may emerge for associating literate experiment information with published literature.

| Architecture | x86_64 |
|---|---|
| Processor | Intel(R) Core(TM) i7 CPU 920 |
| Clock frequency | 1600 MHz |
| No. cores | 8 |
| L1 Data | 32K |
| L1 Instruction | 32K |
| L2 Unified | 256K |
| L3 Unified | 8192K |
| RAM | 5973 MB |
| OS | Linux v2.6.31.14-0.4-default |

**Table 1.** Platform description, automatically generated by the InfoScript

### 3.2 Literate Experiment Scripts

In this section, we introduce various kinds of scriptlets, which have different roles in the literate experiment process. These scriptlets combine to make a full literate experiment. Initially we identify three kinds of scriptlets. We suppose that further types of scriptlet may emerge as interest grows in literate experimentation.

#### 3.2.1 InfoScripts

Some low-level script components will be executed to produce textual output that is incorporated into the LaTeX paper directly, using the `\input` command. We call these components *InfoScripts*. Figure 3 gives a simple InfoScript that, when executed, produces a LaTeX table describing the platform specification on which the experiment is to be executed. This information is obligatory in all meaningful quantitative evaluation studies. Table 1 shows the resulting table, generated on a standard Linux box which we use for experimental evaluation in Section 4.

#### 3.2.2 PipeScripts

Some low-level script components will be executed to produce output that will become the input for another script. We call these *PipeScripts*. For instance, suppose we run an experiment $n$ times to obtain $n$ measurements. These may be written to a data file. Then we apply a PipeScript to this data file to compute the arithmetic mean and standard deviation of the $n$ measurements.

We anticipate providing a library of standard PipeScripts to handle these kinds of common data processing operations.

#### 3.2.3 ExeScripts

Other, higher-level script components will be executed to perform some part of the experiment, and they will also be processed to extract a textual description of the experiment, which can be dropped into the LaTeX paper. These scripts are known as *ExeScripts*. An ExeScript is constructed by

```
#!/bin/sh

# get_sys_info.sh
# Jeremy Singer
# 17 Mar 2011

ARCH=`uname -p`
CPU=`cat /proc/cpuinfo | grep "model name" | head -n 1 | cut -d':' -f 2 | cut -d'@' -f 1`
CLOCK=`cat /proc/cpuinfo | grep "cpu MHz" | head -n 1 | cut -d":" -f 2`
CLOCK=`echo "scale=0; $CLOCK/1" | bc`
CORES=`cat /proc/cpuinfo | grep "processor" | wc -l`


echo "Architecture & $ARCH \\\\";
echo "Processor & $CPU \\\\";
echo "Clock frequency & $CLOCK MHz\\\\";
echo "No.\ cores & $CORES \\\\";

# cache info
for INDEX in {0..5}
do

  if [ -d /sys/devices/system/cpu/cpu0/cache/index${INDEX} ];
     then
     LEVEL=`cat /sys/devices/system/cpu/cpu0/cache/index${INDEX}/level`;
     TYPE=`cat /sys/devices/system/cpu/cpu0/cache/index${INDEX}/type`;
     SIZE=`cat /sys/devices/system/cpu/cpu0/cache/index${INDEX}/size`;
     echo "L${LEVEL} $TYPE & $SIZE \\\\";
  fi
done


RAM_KB=`cat /proc/meminfo | grep MemTotal | cut -d":" -f 2 | awk '{print $1;}'`
RAM_MB=`echo "scale=0; $RAM_KB/1024" | bc`;

echo "RAM & $RAM_MB MB\\\\";


OS=`uname -s`
KERNEL=`uname -r`

echo "OS & $OS v${KERNEL} \\\\";
```

**Figure 3.** InfoScript to gather experimental platform parameters

```
#######
# FETCH
#######
export JikesRvmVersion=3.1.1
#@#
#@# We conduct our experiments on the Jikes RVM platform
#@# \cite{jikesrvm} which is an open-source research based
#@# runtime environment for executing Java bytecode programs.
#@# We use version \JikesRvmVersion of Jikes RVM.
#@#
echo "fetching jikes rvm"
wget http://sourceforge.net/projects/jikesrvm/files/jikesrvm/${JikesRvmVersion}/jikesrvm-${JikesRvmVersion}.tar.bz2/download
```

**Figure 4.** ExeScript fragment to download source code tarball

weaving together commands to execute part of the experiment, and meta-data to describe that experimental process in a human-readable way. (Some of these commands may themselves be InfoScripts or PipeScripts.)

In our prototype implementation, an ExeScript is a well-formed bash script, which has meta-data hidden in special comments. A postprocessing pass over the script can extract the special comments. These are snippets of LaTeX, describing the behaviour and results of the associated command executions in a way that is suitable for inclusion into the paper. Dependencies between the shell script commands and the LaTeX, e.g. version numbers of programs, can be handled cleanly by allowing shared variables across bash and LaTeX, using the `export` directive in bash and the `define` macro command in LaTeX.

For instance, as part of the experimental setup, it may be necessary to download a specific version of program source code from an internet repository. Figure 4 gives an ExeScript snippet that performs this task, including the literate description hidden behind the special `#@#` comment delimiters. A simple perl script can process this ExeScript to generate the appropriate lines of LaTeX for inclusion in the eventual paper.

### 3.3 Experiment as Composition of Tasks

The ExeScript performs a sequence of related tasks, in an experimental pipeline. Again, we anticipate the provision of a set of template experimental pipelines, to handle the most common cases. For instance, the pipeline of tasks might be arranged as follows:

1. FETCH software system (download tarball from website, Maven repository or similar)
2. INSTALL software system (compile, or configure, as necessary)

3. FETCH system mods (patches against original system, stored in an online archive)
4. PATCH system (apply patches to original system)
5. INSTALL modified system (compile, or configure, as necessary)
6. FETCH benchmarks (again, from online repository)
7. INSTALL benchmarks (compile, or configure, as necessary)
8. RUN benchmarks on unmodified system and record results
9. RUN benchmarks on modified system and record results
10. ANALYSE results (to generate graphs, tables, etc)

This kind of workflow is a common experimental approach in computer software systems papers. Indeed, this is the approach we adopted in a recent paper [25] which we endeavour to reproduce in Section 4.

Different sections of the resulting paper will be derived from literate markup in different sections of the script. The script might be divided into logical sections to facilitate this division. For instance, the details about software sources, versions, configurations, and benchmarks will belong in the experimental setup subsection. On the other hand, the actual details of the experiments performed belong in the experimental method subsection. Again, the discussion about data analysis techniques and procedures is most appropriate in the evaluation section.

We recognise that different experiments will require alternative pipeline structures and components. Perhaps we want to compare modified benchmarks on the same underlying system, or to compare a greater number of variants of the original software system against each other. However we feel that many of our standard primitive components, e.g. FETCH, INSTALL, PATCH, RUN, ANALYSE, can be reused. Thus we are happy to provide example templates for these components. This is similar to the *project object model* for software build processes as advocated by the Apache Maven community[5].

Certain specialized scripts will be created entirely by the experiment designer. In fact, these are the scripts s/he would have probably written anyway, to automate the experiment execution. In the new literate approach, the designer simply has to incorporate these scripts, with appropriate literate markup, into the main ExeScript.

### 3.4 Generating the Paper

For now, we presume that papers are produced using the LaTeX software. The literate experiment scripts are post-processed to give fragments of LaTeX that can be incorporated in the paper source. We recommend the use of the LaTeX `\input` directive to weave together auto-generated sections and the manually authored sections of the paper. There are some issues regarding dependencies between these separate files. For instance, the literate description may refer to a label defined in a manually authored section. However these are the kinds of challenges that face authors including multiple LaTeX files at present. We expect that such authors will have no problem adapting to use the literate approach. In addition, we hope to provide basic text-editor support for literate scripts. This should help with resolving inter-file dependencies, pointing out dangling references, etc.

This paper is an initial example of a report on a literate experiment. Certain parts of this paper have been auto-generated from a literate script. These are:

1. Table 1
2. Section 4
3. Figure 5

As mentioned already, Appendix A contains the literate script source code and a tarball is available from `http://www.dcs.gla.ac.uk/~jsinger/casestudy.tar.gz`.

## 4. Case Study

This section of the paper describes a prototype case study in literate experimentation. **The whole of this section is automatically generated from a literate experiment description file**. In an ideal world, the description file would be published and archived alongside the paper. For now, we make the script and its support files available on our web site[6]. In addition, we give the entire script source in an appendix to this paper.

The case study involves reproducing a selection of experimental results from our earlier work on the interaction between micro-economic theory and garbage collection [25]. In this earlier research, we showed that a metric called *allocation elasticity* can be used to control the rate of heap growth in a virtual machine, and hence the overall execution time of a managed program. We stress that the published description of our original experiments [25] was not literate, although we do have access to the shell scripts that we used for that work. Now, in this case study, we are attempting to re-engineer these shell scripts into literate scripts, incorporating this narrative material inlined directly into the ExeScript.

### 4.1 Experimental System Details

We conduct our experiments on the Jikes RVM platform [2] which is an open-source research based runtime environment for executing Java bytecode programs. We use version 3.1.1 of Jikes RVM.

We test our modified heap growth strategy against the default Jikes RVM heap growth policy. Our strategy is avail-

[5] `http://maven.apache.org/what-is-maven.html`

[6] `http://www.dcs.gla.ac.uk/~jsinger/casestudy.tar.gz`

able[7] as a source code patch against Jikes RVM v3.1.1. Note that we also have to apply a small update patch to handle a minor checkstyle compatibility issue, which is Jikes RVM JIRA issue number RVM917.

All our experiments are conducted with the `FastAdaptiveMarkSweep` configuration, which has all internal runtime assertions disabled for production-level performance.

We evaluate our Jikes RVM modifications using the DaCapo benchmark suite [5], version 2006-10-MR2. These are standard Java benchmarks based on real-world open-source programs. DaCapo is considered to be a standard and reliable workload for virtual machine implementation and garbage collection research [4].

All experiments are performed on a stock Linux box, whose specification is given in Table 1.

### 4.2 Experimental Method

We compare the default heap growth policy in Jikes RVM with our alternative, elasticity-based expansion policy by measuring the execution times for DaCapo benchmarks with variable sized heaps.

For each benchmark test, we perform 10 trial executions. We gather timing data from the second iteration of each benchmark execution, with replay compilation. In all cases, Jikes RVM is configured to ignore explicit `System.GC()` requests from the application. We report arithmetic means and standard deviations of times, over the 10 runs.

The *default* heap growth policy we employ commences benchmark execution with a 25 MB heap, and expands the heap by increasing its size according to some factor extracted from a lookup table, which encodes a heap growth heuristic based on GC load and current live ratio. The heap size never grows beyond the maximum specified size of 1000 MB.

The *alternative* elasticity-based heap growth policy we employ commences benchmark execution with the same heap size of 25 MB, and has the same maximum limit of 1000 MB. The heap size grows after each GC for which the current elasticity exceeds the target elasticity, $E$, until the heap reaches the maximum size. At each individual growth step, the heap size is multiplied by a constant *growth ratio* value . We test various elasticity values $E$ between 0.1 and 10. We test growth ratios 1.1, 1.3 and 1.5. This leads to an *exponential* heap growth model. Alternative growth models, such as linear, are possible within the same framework.

### 4.3 Experimental Results

Figure 5 shows execution time results for selected DaCapo benchmarks, with variable sized heaps controlled using the elasticity heuristic outlined above. The $E$ value is varied along the $x$-axis, note the log scale. The execution times are measured on the $y$-axis. Confidence intervals are one standard deviation either side of the arithmetic mean, for each result. We evaluate three different heap growth ratios: 1.1, 1.3, and 1.5. (The lookup table in the default Jikes RVM `HeapGrowthManager` implementation has ratios in the same range for heap expansion.) A larger heap growth ratio will cause the heap to expand more rapidly.

We compare the performance of our new elasticity heuristic for heap growth with the default Jikes RVM policy. In Figure 5, the horizontal line in each benchmark's graph shows the execution time with the default policy.

**The following paragraph, which engages in analysis of the results presented in the graphs, does not logically belong in a literate experiment script, since such analysis can only happen after the results are auto-generated and manually inspected.**

For most benchmarks, the graphs are fairly flat when $E < 1$. Above unit elasticity, the execution time increases sharply. This is because the heap growth is restricted, which causes more GCs to occur, which degrades the application execution time. For some benchmarks, such as xalan, the default policy gives comparable performance with our new elasticity heuristic when $E < 1$. For other benchmarks, such as antlr, elasticity-based heap growth out-performs the default policy when $E < 1$, particularly for higher growth ratios.

**The final paragraph in this section, which engages in comparison of the latest results with those from our earlier work [25], does not logically belong in the ExeScript.**

We only ran experiments for three DaCapo benchmarks, namely antlr, bloat and xalan. This was merely due to time and space constraints, rather than any inherent difficulty in the experimental process. However we are encouraged to note that the final graphs in Figure 5 are very similar to those in our original paper on this topic. Indeed, we appear to have reproduced, and repeated, our earlier results.

## 5. How to Encourage Adoption

As with any major change in community behaviour, people will need time and incentives before they pick up the practice of literate experimentation.

Given that most code is recycled and adapted, we expect that a simple 'hello world' literate experiment might be helpful for other researchers to make an initial evaluation. Eventually, we hope to have a library of template literate experiment scripts / scriptlets. It would be useful to set up a website (e.g. `literateexperiments.com`) to host these examples, along with tutorials for beginners, and a community forum.

We anticipate that a major uptake would require meaningful engagement with program committees and journal editorial boards. It might be possible to have some high-profile workshops and journal special issues, for which all paper submissions are required to include literate experimental descriptions. We feel that as soon as researchers have used the

[7] `http://sourceforge.net/tracker/download.php?group_id=128805&atid=723235&file_id=379409&aid=3026328`

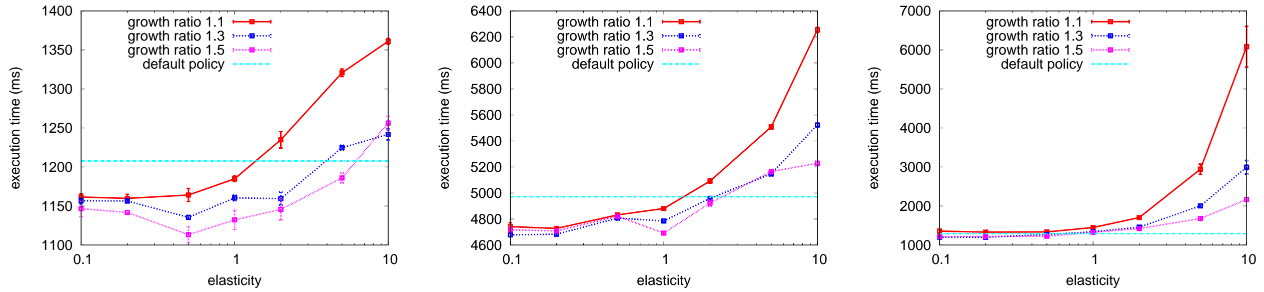


**Figure 5.** Execution times for selected DaCapo benchmarks using a full-heap collector, with variable sized heap based on elasticity heuristic, for several elasticity values and growth ratios

literate approach, they should become adherents, and perhaps even advocates of literate experimentation.

A fundamental step required to enable serious literate experimental practice is the provision of permanent and open repositories for result publication. Furthermore, the research systems development community will need to encourage and support the distribution and archival of *all* experimental artifacts, such as source code, benchmarks, patches, data sets. The Jikes RVM team [2] is a shining example of this level of care. They have an open-source infrastructure [3] available on sourceforge[8], with older versions archived as tarballs. They also provide a publicly accessible research archive[9] where people can post patches to versioned releases of the trunk source code, with links to research papers.

Other projects vary in their levels of support for such a research eco-system. Hall et al [13] advocate the use of open repositories for experimental data for research-based software systems, however they acknowledge that such practice is not yet widespread in the community.

## 6. Related Work

The classic text on proper experimental practice is Lilja [18]. However he does not offer advice about documenting experimental practice, only about conducting the actual experiments. Georges et al [11] give more recent and focused advice on software systems experiments. Again they are concerned about interpretation of results, rather than documenting practice.

The Elsevier Executable Paper Challenge[10] took place while this paper was in submission. The challenge shares many aims with our literate experimentation philosophy. The winners were recently announced, and we consider two of the solutions below.

Nowakowski et al [21]. present a web based publication repository. Each paper consists of static regions of text and interactive *assets* which allow readers to explore experimental parameters or repeat an experiment on a remote server. Their executable assets correspond to experimental scripts in our system. They are restricted by what can be executed safely on a remote server; reportedly arbitrary Perl, Python or Ruby code. The domain of the experiments will determine whether this support is sufficient for meaningful experiments. Further, it is not clear how closely their executable assets are linked with static text, i.e. how literate is their scripting approach? This is the chief priority in our literate experimental method.

Van Gorp and Mazanek [12] advocate an approach in which paper authors package up their document, experimental scripts and data as files in a virtual machine (VM) image. This VM image is then curated and distributed by the scientific publisher. Subscribers can instantiate the VM image remotely in order to read the paper and re-execute the experiments. The motivation is that all experiments should be perfectly reproducible since the original authors and later readers are running on an identical (virtual) platform. Our work differs from theirs in two main areas. (1) We only distribute the experimental scripts with the paper, rather than an entire platform image. Thus our scripts are responsible for configuring the evaluation platform appropriately, i.e. downloading and installing appropriate software packages, etc. (2) Their approach is not literate, since the paper and the experimental script are two separate entities stored in the VM image. We use a literate experimental description, which weaves together the experimental execution commands and a high-level description of the experiment.

Leisch [17] introduces the concept of literate data analysis to generate statistical reports. His system allows authors to write a single script file that includes LaTeX code for text processing and R code for data processing. This file is processed to execute the R fragments, incorporate the appropriate output into the LaTeX document and then compile this to generate a final PDF. Leisch's approach is used to generate the documentation for R, and for statistical analysis lecture notes. The main difference between his approach and ours is that Leisch has no built-in support for generating new raw data. His package can only analyse existing data files. Our script-based system can execute arbitrary software experiments to generate fresh data sets.

[8] `http://sourceforge.net/projects/jikesrvm/`

[9] `http://jikesrvm.org/Research+Archive`

[10] `http://www.executablepapers.com`

# 7. Conclusions

## 7.1 Summary

In this paper, we have introduced a *literate* approach to the construction, execution and reporting of computer systems experiments. We feel that literate experimentation may initially involve greater investigator effort. However this burden should be amply repaid with two clear benefits:

1. Literate experimentation provides a templated approach to enable researchers to devise well-constructed experiments, and subsequently better structured papers.
2. Literate experimentation supports more straightforward reproduction of results. This benefits both reviewers and readers, along with future generations of researchers attempting to improve upon the work.[11]

## 7.2 Limitations

We acknowledge that our approach as presented in this paper is prototypical. Currently it lacks many useful features. The implementation depends on Linux for experiment execution and LaTeX for paper authoring. The scripting mechanism does not give support for any level of sophisticated error handling or recovery. Further, it is not possible to select parts of the experiment to be executed in isolation, without resorting to commenting out appropriate regions of the script.

We intend to implement and distribute a more mature, platform neutral, fully featured scripting language and toolset for literate experimentation in the near future.

Another criticism is that our approach is currently limited to human interaction with the experiments. Future extensions may incorporate rich meta-data with a literate experimental write-up, using semantic web technology to enable machine understanding and processing. An example of development in this direction is the provenance information included in the Verifiable Computational Results project [10].

## 7.3 Potential Impact

Knuth himself concedes that the practice of literate programming has not become as widespread as he had initially hoped[12]. Nevertheless, inline code commenting (e.g. using Javadoc [16]) is now standard behaviour for most developers. In a sense, this is a diluted form of Knuth's literate programming. We dare to imagine that our *literate experimentation* techniques could at least have the same kind of positive effect.

We feel that any improvement in experimental Computer Science has to be a useful contribution to the field. Thus we offer our literate experimentation manifesto to the world. We conclude with a couple of drum-banging catch-phrases that encapsulate the literate experiment concept:

(i) Write up your experiments at design time!
(ii) Weave together the description with the meta-description of your experimental process!

# A. ExeScript Source for Case Study

This appendix gives the source code listing for the literate experiment ExeScript. This ExeScript was used to generate Section 4 in this paper.

```
1  #!/bin/sh
2
3  # case_study.sh
4  # Jeremy Singer
5  # 12 Apr 11
6
7  #@#
8  #@# This section of the paper describes a
9  #@# prototype case
10 #@# study in literate experimentation.
11 #@# \textbf{The whole
12 #@# of this section is automatically generated
13 #@# from a literate experiment description file}.
14 #@# In an ideal world, the description file would
15 #@# be published and archived alongside the paper.
16 #@# For now, we make the script and its support files
17 #@# available on our
18 #@# web site\footnote{\url{http://www.dcs.gla.ac.uk/~
       jsinger/casestudy.tar.gz}}.
19 #@# In addition, we give the entire script source in an
20 #@# appendix to this paper.
21 #@#
22 #@# The case study involves reproducing a selection of
23 #@# experimental results from our earlier work
24 #@# on the interaction between micro-economic
25 #@# theory and garbage collection
26 #@# \cite{singer10economics}.
27 #@# In this earlier research, we showed that a metric
28 #@# called \textit{allocation
29 #@# elasticity} can be used to control the rate
30 #@# of heap growth in a virtual machine, and hence
31 #@# the overall execution time of a managed program.
32 #@# We stress that the published description of our
33 #@# original experiments \cite{singer10economics}
34 #@# was not
35 #@# literate, although we do have access to the shell
36 #@# scripts that we used for that work.
37 #@# Now, in this case study, we
38 #@# are attempting to re-engineer these shell scripts
39 #@# into literate scripts, incorporating this
40 #@# narrative material inlined directly
41 #@# into the ExeScript.
42 #@#
43 #@#
44
45 #@# \subsection{Experimental System Details}
46
47 #######
48 # FETCH
49 #######
50 export JikesRvmVersion=3.1.1
51 #@#
52 #@# We conduct our experiments on the Jikes RVM platform
53 #@# \cite{alpern00jalapeno}
54 #@# which is an open-source research based
55 #@# runtime environment for executing
56 #@# Java bytecode programs.
57 #@# We use version \JikesRvmVersion of Jikes RVM.
58 #@#
59 echo "fetching jikes rvm"
60 wget http://sourceforge.net/projects/jikesrvm/files/
       jikesrvm/${JikesRvmVersion}/jikesrvm-${
       JikesRvmVersion}.tar.bz2/download
61 mkdir orig
62 cp jikesrvm-${JikesRvmVersion}.tar.bz2 orig
```

[11] We include the original investigator among 'future generations', since s/he might return to her/his work in several years time, when s/he has forgotten how to run the experiments, and the relevant postgraduate students have moved on.

[12] Knuth stated this at the BCS/IET Turing Lecture in Glasgow, 2011. http://www.bcs.org/content/ConWebDoc/38050

```
63  cd orig
64  echo "unpacking orig RVM"
65  tar xjf jikesrvm-${JikesRvmVersion}.tar.bz2
66  cd ..
67  #
68  mkdir patched
69  cp jikesrvm-${JikesRvmVersion}.tar.bz2 patched
70  cd patched
71  echo " unpacking patch RVM"
72  tar xjf jikesrvm-${JikesRvmVersion}.tar.bz2
73  cd ..
74
75  ########
76  # PATCH
77  ########
78  cd patched
79  export PatchFile=http://sourceforge.net/tracker/download.
        php?group_id=128805&atid=723235&file_id=379409&aid
        =3026328
80  #
81  #@# We test our modified heap growth strategy
82  #@# against the default
83  #@# Jikes RVM heap growth policy. Our strategy is
84  #@# available\footnote{\url{\PatchFile}}
85  #@# as a source code patch
86  #@# against Jikes RVM v\JikesRvmVersion.
87  #
88  wget "$PatchFile"
89  cp download.php* patch.tar.gz
90  tar xzf patch.tar.gz
91  cp ./elast_patch/elast-1.0.patch ./jikesrvm-3.1.1
92  cd jikesrvm-3.1.1
93  echo "patching patch RVM with elast patch"
94  patch -p0 < elast-1.0.patch
95  cd ../..
96
97  # Another patch for checkstyle
98  export RvmIssue=RVM917
99  #@# Note that we also have to apply a small
100 #@# update patch to handle
101 #@# a minor checkstyle compatibility issue, which is
102 #@# Jikes RVM JIRA issue number \RvmIssue.
103 wget http://jira.codehaus.org/secure/attachment/54615/
        bandaid-fix-for-${RvmIssue}.diff
104 cd orig/jikesrvm-3.1.1
105 patch -p0 <../../bandaid-fix-for-RVM917.diff
106 cd ../../patched/jikesrvm-3.1.1
107 patch -p0 <../../bandaid-fix-for-RVM917.diff
108 cd ../..
109
110 #########
111 # INSTALL
112 #########
113 export JAVAHOME=/usr/java/jdk1.6.0_18/
114 export JikesRvmConfig=FastAdaptiveMarkSweep
115 #@#
116 #@# All our experiments are conducted with the
117 #@# \texttt{\JikesRvmConfig} configuration, which has
118 #@# all internal runtime assertions
119 #@# disabled for production-level performance.
120 #@#
121 echo "building orig jikes rvm"
122 cd orig/jikesrvm-3.1.1
123 ant -Dhost.name=x86_64-linux -Dconfig.name=
        $JikesRvmConfig
124 cd ../..
125
126 echo "building patched jikes rvm"
127 cd patched/jikesrvm-3.1.1
128 ant -Dhost.name=x86_64-linux -Dconfig.name=
        $JikesRvmConfig
129 cd ../..
130
131
132 #######
133 # FETCH Benchmarks
134 #######
135 export BmVersion=2006-10-MR2
136
137 #@# We evaluate our Jikes RVM modifications using the
138 #@# DaCapo benchmark suite \cite{blackburn06dacapo},
139 #@# version \BmVersion.
140 #@# These are standard Java benchmarks based on
141 #@# real-world open-source programs.
142 #@# DaCapo is considered to be a standard and
143 #@# reliable workload
144 #@# for virtual machine implementation and
145 #@# garbage collection research
146 #@# \cite{blackburn08wake}.
147 #@#
148 wget "http://dacapo.anu.edu.au/regression/perf/dacapo-${
        BmVersion}.jar"
149
150 ###############
151 # INFOSCRIPT
152 ###############
153 ./get_sys_info.sh > sys_info.table
154 #
155 #@# All experiments are performed on a stock
156 #@# Linux box, whose specification is given
157 #@# in Table \ref{tab-prop-scripts-infoscript}.
158
159 #
160 #@# \subsection{Experimental Method}
161 #
162
163 ######
164 # RUN timing expts on orig RVM
165 ######
166
167 #@# We compare the default heap growth
168 #@# policy in Jikes RVM
169 #@# with our alternative,
170 #@# elasticity-based expansion policy
171 #@# by measuring the execution times for DaCapo
172 #@# benchmarks with variable sized heaps.
173 #@#
174
175 export NumTrials=10
176
177 #@#
178 #@# For each benchmark test,
179 #@# we perform \NumTrials trial executions.
180 #@# We gather timing data from the second iteration
181 #@# of each benchmark execution, with replay compilation.
182 #
183 #@# In all cases, Jikes RVM is configured to ignore
184 #@# explicit \texttt{System.GC()} requests from the
185 #@# application.
186 #
187 #@# We report arithmetic means and standard deviations
188 #@# of times,
189 #@# over the \NumTrials runs.
190 #
191 export HeapStart=25
192 export HeapMax=1000
193 #
194 #@#
195 #@# The \emph{default} heap growth policy we employ
196 #@# commences benchmark execution
197 #@# with a \HeapStart MB heap,
198 #@# and expands the heap by
199 #@# increasing its size according to some factor
200 #@# extracted from a lookup table, which encodes a heap
201 #@# growth heuristic based on
202 #@# GC load and current live ratio.
203 #@# The heap size never grows beyond the
204 #@# maximum specified size
205 #@# of \HeapMax MB.
206
207 RVM=./orig/jikesrvm-3.1.1/dist/
        FastAdaptiveMarkSweep_x86_64-linux/rvm
208 mkdir orig_results
209
210 for BM in `cat bms.txt`
211 do
212     REPLAY_FLAGS="-X:aos:enable_replay_compile=true   \
213                 -X:vm:edgeCounterFile=$BM.edges        \
214                 -X:aos:cafi=$BM.advice                 \
215                 -X:aos:dcfi=$BM.callgraph"
216
```

```
217     echo "" | cat > orig_results/$BM.timings.gcs;  #
            blank file
218
219     echo "bm is $BM";
220
221     for I in `seq 1 $NumTrials`;
222     do
223   echo "starting heapsize is $HeapStart";
224   echo "max heapsize is $HeapMax";
225   $RVM -X:gc:verbose=0 -X:gc:ignoreSystemGC=true -Xms${
          HeapStart}M -Xmx${HeapMax}M $REPLAY_FLAGS -
          classpath ./dacapo-2006-10-MR2.jar Harness -n 2
          $BM
226
227     done 2>&1 | cat >> orig_results/$BM.timings.gcs
228 done
229
230
231 ######
232 # RUN timing expts on modified RVM
233 ######
234 #
235 #@#
236 #@# The \emph{alternative} elasticity-based
237 #@# heap growth policy we employ
238 #@# commences benchmark execution
239 #@# with the same heap size of
240 #@# \HeapStart MB, and has the same
241 #@# maximum limit of \HeapMax MB.
242 #@# The heap size grows
243 #@# after each GC for which the current elasticity
244 #@# exceeds the target elasticity, $E$,
245 #@# until the heap reaches the maximum size.
246 #@# At each individual growth step, the heap size is
247 #@# multiplied
248 #@# by a constant \textit{growth ratio} value.
249 #@# We test various elasticity values
250 #@# $E$ between 0.1 and 10.
251 #@# We test growth ratios 1.1, 1.3 and 1.5.
252 #@# This leads to an \textit{exponential}
253 #@# heap growth model.
254 #@# Alternative growth models, such as linear,
255 #@# are possible within the same framework.
256
257 RVM=./patched/jikesrvm-3.1.1/dist/
        FastAdaptiveMarkSweep_x86_64-linux/rvm
258
259 mkdir patched_results
260
261 for BM in `cat bms.txt`
262 do
263     REPLAY_FLAGS="-X:aos:enable_replay_compile=true   \
264                   -X:vm:edgeCounterFile=$BM.edges     \
265                   -X:aos:cafi=$BM.advice              \
266                   -X:aos:dcfi=$BM.callgraph"
267
268     for GR in 1.1 1.3 1.5
269     do
270   echo "" | cat > patched_results/$BM.timings-${GR}.gcs;
          # blank file
271
272   echo "bm is $BM";
273
274   for ELASTICITY in 0.1 0.2 0.5 1 2 5 10
275   do
276       echo "elasticity is $ELASTICITY";
277       for I in `seq 1 $NumTrials`;
278       do
279     echo "starting heapsize is $HeapStart";
280     echo "max heapsize is $HeapMax";
281     $RVM -X:gc:verbose=0 -X:gc:elasticity=${ELASTICITY} -
            X:gc:growthratio=${GR} -X:gc:ignoreSystemGC=true
            -Xms${HeapStart}M -Xmx${HeapMax}M $REPLAY_FLAGS
            -classpath ./dacapo-2006-10-MR2.jar Harness -n
            2 $BM
282
283       done
284   done 2>&1 | cat >> patched_results/$BM.timings-${GR}.
          gcs
285     done
286 done
287
288 #@# \subsection{Experimental Results}
289
290 ############
291 # ANALYSE data to produce results summary files
292 ############
293
294 echo "generating graphs"
295 mkdir graphs
296
297 # Use PipeScripts taken from original experiment,
298 # to calculate means and stdevs from all dump files
299 for BM in `cat bms.txt`
300 do
301     for GR in 1.1 1.3 1.5
302     do
303   ./pipescripts/process_patched_timing_dumps.pl ./
          patched_results/${BM}.timings-${GR}.gcs > ./graphs
          /${BM}-${GR}.dat
304     done
305 # Now handle default case (standard heapgrowth model)
306     ./pipescripts/process_orig_timing_dumps.pl ./
          orig_results/${BM}.timings.gcs > ./graphs/${BM}-
          default.dat
307 done
308
309 ##############
310 # GENERATE graphs from results summaries
311 ##############
312 cd graphs
313 for BM in `cat ../bms.txt`
314 do
315     gnuplot gnuplot.commands.${BM}.times
316     epstopdf ${BM}-elast-times.eps
317 done
318 cd ..
319
320 #@# Figure \ref{fig-casestudy-graphs} shows
321 #@# execution time results for selected DaCapo
322 #@# benchmarks, with variable sized heaps controlled
323 #@# using the elasticity heuristic outlined above. The
324 #@# $E$ value is varied along the $x$-axis, note the
325 #@# log scale. The execution times are measured on
326 #@# the $y$-axis.  Confidence intervals are one
327 #@# standard deviation either side of the arithmetic
328 #@# mean, for each result.  We evaluate three
329 #@# different heap growth ratios: 1.1, 1.3, and
330 #@# 1.5. (The lookup table in the default Jikes RVM
331 #@# \texttt{HeapGrowthManager} implementation has
332 #@# ratios in the same range for heap expansion.)  A
333 #@# larger heap growth ratio will cause the heap to
334 #@# expand more rapidly.
335 #@#
336 #@# We compare the performance of our new elasticity
337 #@# heuristic for heap growth with the default Jikes
338 #@# RVM policy.  In Figure
339 #@# \ref{fig-casestudy-graphs}, the horizontal
340 #@# line in each benchmark's graph shows the execution
341 #@# time with the default policy.
342 #@#
343 #@# \textbf{The following paragraph, which
344 #@# engages in analysis of the results presented in
345 #@# the graphs, does not logically belong in a literate
346 #@# experiment script, since such analysis
347 #@# can only happen
348 #@# after the results are auto-generated and manually
349 #@# inspected.}
350 #@#
351 #@# For most benchmarks, the graphs are
352 #@# fairly flat when $E<1$.Above unit elasticity, the
353 #@# execution time increases sharply.  This is because
354 #@# the heap growth is restricted, which causes more
355 #@# GCs to occur, which degrades the application
356 #@# execution time.
357 #@# For some benchmarks,
358 #@# such as xalan, the default policy gives
359 #@# comparable performance with our new elasticity
360 #@# heuristic when $E<1$.  For other benchmarks, such
361 #@# as antlr, elasticity-based heap growth
362 #@# out-performs the default policy when $E<1$,
363 #@# particularly for higher growth ratios.
```

```
364 #@#
365 #@# \textbf{The final paragraph in this section, which
366 #@# engages in comparison of the
367 #@# latest results with those
368 #@# from our earlier work \cite{singer10economics},
369 #@# does not logically belong in the ExeScript.}
370 #@#
371 #@# We only ran experiments for three
372 #@# DaCapo benchmarks, namely
373 #@# antlr, bloat and xalan.
374 #@# This was merely due to time and space
375 #@# constraints, rather than any
376 #@# inherent difficulty in the
377 #@# experimental process. However we
378 #@# are encouraged to note that
379 #@# the final graphs are very similar
380 #@# to those in our original
381 #@# paper on this topic. Indeed, we appear
382 #@# to have reproduced, and repeated,
383 #@# our earlier results.
```

## References


[1] *Proceedings of the 2007 Workshop on Experimental Computer Science*. ACM, 2007.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb 2000.

[3] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, Feb 2005.

[4] S. Blackburn, K. McKinley, R. Garner, C. Hoffmann, A. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, et al. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, October 2006.

[6] F. Casati, F. Giunchiglia, and M. Marchese. Publish and perish: why the current publication and review model is killing research and wasting your money. *Ubiquity*, January 2007.

[7] Committee on Academic Careers for Experimental Computer Scientists, National Research Council. *Academic Careers for Experimental Computer Scientists and Engineers*. National Academies Press, 1994.

[8] A. Diwan and R. Hundt. Repeatable, reproducible, and useful. In *Proceedings of the NSF Workshop on Archiving Experiments to Raise Scientific Standards*, 2010.

[9] L. Eeckhout. Position statement at Evaluate 2010, 2010.

[10] M. Gavish and D. Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4:637–647, 2011.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 57–76, 2007.

[12] P. V. Gorp and S. Mazanek. Share: a web portal for creating and sharing executable research papers. *Procedia Computer Science*, 4:589 – 597, 2011.

[13] M. Hall, D. Padua, and K. Pingali. Compiler research: the next 50 years. *Communications of the ACM*, 52:60–67, February 2009.

[14] P. Jump. Research intelligence—rip it up and start again. *Times Higher Education*, Dec. 2010.

[15] D. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[16] D. Kramer. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, pages 147–153, 1999.

[17] F. Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. Technical Report Report No. 69, SFB Adaptive Information Systems and Modelling in Economics and Management Science, Mar 2002.

[18] D. J. Lilja. *Measuring Computer Performance A Practitioner's Guide*. Cambridge University Press, 2000.

[19] V. E. McMillan. *Writing papers in the biological sciences*. Bedford Books, 1997.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2009.

[21] P. Nowakowski, E. Ciepiela, D. Harezlak, J. Kocot, M. Kasztelnik, T. Bartynski, J. Meizner, G. Dyk, and M. Malawski. The collage authoring environment. *Procedia Computer Science*, 4:608 – 617, 2011.

[22] K. N. Nwogu. The medical research paper: Structure and functions. *English for Specific Purposes*, 16(2):119 – 138, 1997.

[23] M. Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[24] M. Shaw. Writing good software engineering research papers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 726–736, 2003.

[25] J. Singer, R. E. Jones, G. Brown, and M. Luján. The economics of garbage collection. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 103–112, 2010.

[26] W. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.