



VRIJE  
UNIVERSITEIT  
BRUSSEL



# GUIDELINES FOR PERFORMANCE EVALUATION

Parallelism and Distribution

Lecturer: Elisa Gonzalez Boix  
Assistant: Jens Van der Plas

Sciences and Bio-Engineering Sciences

*Benchmarking is used to measure the performance of a program based on a specific indicator (e.g., time to compute, throughput of the program, memory usage, etc.) which can then be compared (e.g., across different configurations, architectures, etc.). Benchmarking is tricky, as the exact time the program needs to run may depend on a variety of factors, some of which are out of a developer's control. This document describes common pitfalls and provides some guidelines you should follow when benchmarking your application to be able to draw meaningful conclusions. It is based on best practices from literature as well as from experiences by researchers involved in this course and its precursory version, including Steven Adriaensen, Sam Van den Vonder, and Jens Van der Plas. We first describe general guidelines for conducting performance evaluation followed by specific ones for parallel performance (conducted in this course). In appendix A, we list some tools that can help you to run your experiments on a server.*

## 1. Prior to Benchmarking

The main goal of benchmarking an application is to evaluate its performance. Before benchmarking, make sure the application is correct and meets the requirements. If unit tests are provided, make sure all tests pass to avoid benchmarking a faulty implementation (which could also make it difficult to interpret the results correctly).

Before any experiment can be run, you must decide on what you are going to test. If you are going to compare multiple configurations of your program, make sure to only vary one parameter at a time. When multiple parameters are varied, drawing valid conclusions is most likely impossible. Also, think of which values you are going to use for the program's parameters. Sensible choices for these values may depend on the machine on which you are going to run the experiments, or on the dataset you are using, for example.

Once you have decided what you are going to test, you can set up the performance benchmarks. For this, you need a *benchmarking program*, i.e., a program that runs the code you want to benchmark and performs the measurements. Performance evaluations should be conducted using established *benchmarking suites* (i.e., collections of benchmarking programs) since it improves the comparability of results [1]. However, you will probably write a benchmarking program dedicated for your project, which runs your code with different values for its parameters and performs the measurements. Non-standard benchmark suites are also used in literature when it is a better choice than the established ones, e.g., a single-threaded benchmark suite is not suitable for evaluating parallel performance [1].

Benchmarking is not an easy task since the exact time a program needs to run to completion depends on a variety of factors. Therefore, multiple runs of the same program will not take exactly the same amount of time. To properly characterise the behaviour of a program, every experiment must be repeated multiple ( $n$ ) times. These  $n$  trials will allow you to get a sufficiently accurate results of the program's execution, and to use some statistics to draw conclusions from the results afterwards. Moreover, more trials may be needed when programs run on a Virtual Machine (VM) which uses a Just-In-Time (JIT) compiler.

When a program runs on a JIT compiling VM such as the Java Virtual Machine (JVM), it is first interpreted for a certain period of time until the JIT identifies *hot* parts in the program (e.g., frequently executed loops or method calls) which are then optimised (i.e., more efficient machine code is generated for these parts). Program execution is thus slow during this phase (called the *warmup* phase) as the JVM is still optimising code, and fast afterwards when the program is said to be running at *steady-state* performance [2, 3]. To avoid an influence of the warmup phase on the results, you need to take into account a warmup phase for *each* experiment, meaning that the measurements of the first  $w$  iterations during which the JVM is optimising code need to be discarded. You need to analyse every experiment individually to determine the proper value for  $w$  to ensure the program has reached steady-state<sup>1</sup>. Note that these  $w$  iterations come on top of the  $n$  trials you do measure (so in total, you have to repeat every experiment  $w + n$  trials).

There are several other considerations to take into account when preparing benchmarking experiments:

---

<sup>1</sup> Georges et al. [2] describe how steady-state performance can be determined, which is a complex process. Barret et al. [3] showed that making  $w$  a fixed number as often done (e.g. 5) may not guarantee that the program has reached a steady state. A good value for  $w$  may depend on the runtime of your program (i.e., choose a bigger  $w$  for shorter runtimes).

- When an experiment only has a very short runtime, the variety between runs will be (relatively) larger than when an experiment takes a long(er) time to run. For experiments that only take a very short time (e.g., 10–100ms),  $n$  can/should be made sufficiently large ( $n \geq 1000$ ). This is for example the case when you run an experiment locally on a computer using a small dataset. For experiments that take a (much) longer time (e.g., when using a big data set on a server), a smaller  $n$  will need to be used. When you are limited in time (e.g., because you were appointed a specific time slot on a server), you will need to take this into account when deciding on  $n$ .
- Make sure every dataset you use to benchmark the program is loaded only once (if possible). Loading a large dataset may take a long time, and therefore could lead to a lot of wasted time if this is repeated for every repetition of an experiment.
- Take into account that things may go wrong during benchmarking. In some cases, it may be perfectly fine if the benchmarks stop when there is an error. However, in other cases, your benchmarking program might just be able to continue with another experiment. Therefore pay attention to exception handling: if you run your experiments on a server, you might potentially only notice crashes after some time has passed, losing valuable time of your assigned server time slot. Also, make sure to write your results to a file as soon as possible. After all, you do not want to lose results should the benchmarking program crash.
- When performing measurements on programs running on a VM or another managed runtime, garbage collection is also a factor that affects program execution. Since the garbage collector may suspend the program execution to ensure safety of the object graph, it may introduce a slowdown. You might want to force a garbage collection before every (repetition of an) experiment. Unfortunately, most JVMs will ignore an explicit call to the garbage collector. As such, it is recommended to handle this issue with repetitions for programs running on the JVM.

When benchmarking programs that run on the JVM, it is strongly recommended to use Java Microbenchmark Harness (JMH)<sup>2</sup>, a Java library for writing, running and analysing microbenchmarks on the JVM. JMH may not prevent all the issues explained in this document, but it certainly helps in diminishing them (e.g., it avoids unwanted optimisations which may compromise the results).

## 2. Running the Experiments

Once your application is believed to be correct and you have a benchmarking program, it is time to actually run your experiments. Use the same machine for all experiments you compare to each other: different machines may lead to different runtimes and hence will make a meaningful comparison impossible.

Similarly, use the same VM version for experiments that will be compared (or e.g., the same compiler, . . .), as this can also influence the results. Make sure you use the same (VM) configuration options as well. If a VM parameter needs to be changed, use the same customized value for all experiments you perform and report it. For example, the heap size of the JVM influences the number of garbage collections that will occur during the execution of the benchmarks, and hence changing the heap size influences garbage collection. However, you should *not* tune garbage collection parameters unless you are absolutely certain that the bad performance lies in how the garbage collector works. Garbage collection tuning is a very complex task, and in many cases, bad garbage collection performance is only a symptom of a bigger problem in your code in terms of memory management; you may best put some time into refactoring your code to improve its efficiency<sup>3</sup>.

During your experiments, you want to keep your system as stable as possible to avoid influences on your measurements. Therefore make sure no other users are using the machine and avoid running other applications during the experiments. When you are using a server to benchmark your project, you will have the machine for yourself during an assigned time slot. Make sure you do not go over your assigned timeslot to avoid hampering the experiments of the other students.

<sup>2</sup><https://openjdk.java.net/projects/code-tools/jmh/>

<sup>3</sup>You may however need to adjust the heap size of the JVM to accommodate working with large datasets etc. Nonetheless, this is the only garbage collection parameter you should normally need to modify. Also note that increasing the heap size may mask errors in your code, hence use this parameter with sufficient care. You will probably be informed when a bigger heap size is needed.

The operating system and hardware of the machine can also influence your measurements. For example, PCs have a battery saving mode. Therefore, make sure such modes do not get enabled/disabled whilst running the experiments, but ensure that the same modes remain active (e.g., by plugging your laptop into AC power).

### 3. Reporting and Analysing Experimental Results

After having run the experiments and obtained the measurements, you will have to analyse and report your results appropriately. Note that your report provides a window to your experiments to other people who are interested in your experiments. Therefore, the style and quality of your report are important, as your report gives reviewers (e.g., the course assistant) a first impression of (the quality of) your research. In your report, you will have to describe a.o. the experiments you performed, the experimental setup, the results you obtained and the conclusions you draw from these results. Note that in science, reproducibility is important, so that other researchers can verify your measurements and conclusions. This means that your report should contain sufficient information and details so that others can reproduce your experiments and derive the same conclusions. This is also crucial for comparison with future ideas and systems.

#### Describing the Experiments and Experimental Setup

In order to support the interpretation of your results, your report should start with a thorough description of the experiments you performed. You should describe what you measured, the parameters you varied, how you performed the measurements, etc. Clearly describe how the experiments were executed (e.g., number of repetitions, number of runs discarded to account for VM warmup,...). When you vary a parameter of the implementation, it may make sense to explain the choice of values for this parameter.

To allow your experiments to be reproduced and results to be verified, it is also important to include all parameters used by the hardware and software platforms which are relevant for the experiments. More concretely, you should report your hardware platform in full detail. For example, does your processor support *hyperthreading*<sup>4</sup> or *turbo boost*<sup>5</sup>? You need to do this for every machine you use, even if it is a server you are required to use with given specifications (after all, someone else reading your report may not know these specifications). Finally, it is important to also report version numbers of software running on the machine (e.g., OS and JVM) and the parameters you used to configure the software (e.g., the JVM heap size and specific compiler options you enabled). In conclusion, report information w.r.t. your experiments and experimental setup in sufficient detail so that it enables the repeatability of your experiments.

#### Presenting the Experimental Results

For every experiment, you will have obtained a series of measurements, of which you will calculate statistical properties. Often, summary statistics like the average or median value of the measurements are used to characterize results [1, 2]. However, it is also important to give information about the error of your measurements, e.g., by presenting confidence intervals and the standard deviation, to understand the distribution of the data. If you use confidence intervals, make sure to mention which confidence level they represent. Information on the error of your measurements will help you to give a meaningful interpretation to your results.

To report results, graphs are preferred over big tables of numbers as they provide a visual intuition about your results. First, it is important to consider what data you want to report, that is, you should think of a visual representation of your results that enables you to easily analyse your results later. For example, do you want to report your measurements as such, such as runtimes, directly, or is it better to report derived values such as speed-ups? Depending on the experiment and the data you want to report, different types of graphs may be appropriate. Make sure it is clear what data every graph represents, by giving the graph an appropriate title and caption, and by naming the axes. Zooming into the interesting range of an axis may help to explain differences but you should be explicit about it in the report to avoid

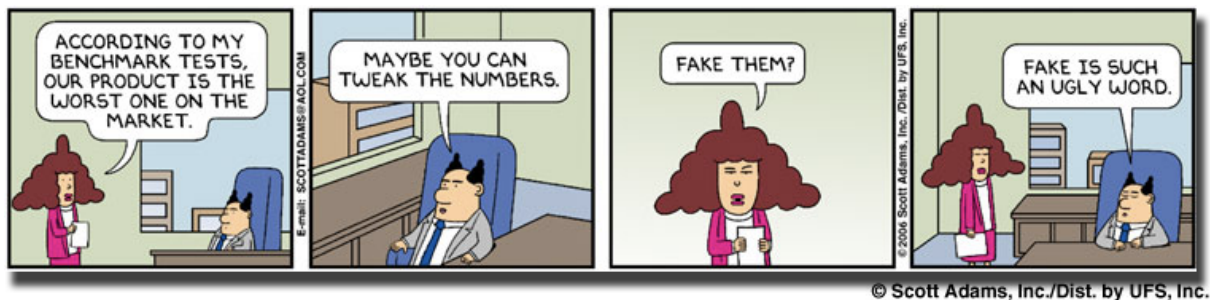
<sup>4</sup>Processors with hyperthreading have multiple hardware threads (usually 2) per physical core. Hence, it is possible that your operating system mentions there are 8 logical cores, whereas your processor only has 4 physical cores. However, even in the perfect case, you will not get a speed-up of 8.

<sup>5</sup>Turbo Boost is a technique that allows your processor to run at a higher clock frequency under certain circumstances, e.g., when the workload is high (or conversely, when it is low), depending on the power supply (AC power or battery), temperature, etc.

misleading the readers. Avoid truncated axes and incorrectly plotted ratios as they may also mislead visual intuition [1]. Finally, visually depict the error on your measurements as well.

### Analysing the Results

Given your experimental results, you will (often) be required to make certain statements about your results (i.e., draw certain conclusions) to answer the research questions you aim to solve. Depending on your results, it may (or may not) be easy to draw conclusions. When analysing the results, do not forget to point the reader to the graphs corresponding to the conclusion. Often, reports may contain multiple graphs and therefore it is important to be clear which results you used when making a specific statement. Do not forget to elaborate and explain how certain results lead to a certain conclusion. You will notice here that a clear and correct and clear graphical representation of your results is crucial for you to be able to draw conclusions based on your experimental results! Therefore, spend sufficient time on processing the results of your experiments and on creating graphs.



## 4. Course-specific Guidelines

In this section, we list specific additional guidelines that apply to this course, i.e., for parallel performance.

1. You will be measuring speed-ups of your parallel implementation. However, also pay attention to the sequential parts of your code (e.g., the joining of two partial results in a parallel Fork/Join computation). E.g., you will see poor speed-ups if at every join of partial results you employ some inefficient operations, such as the sequential traversal (or concatenation) or array(list)s. Hence, pay attention to efficiency where possible. Note that the focus of the project is on efficiently parallelising a given implementation, which therefore also includes what you do sequentially as part of your Fork/Join tasks.
2. Since we are studying parallel performance, it is important to report the number of physical and logical processors of the machines used as well as the JVM heap size (which is not equal to a machine's RAM size).
3. Even though in practice it is important to use the same JVM on your machine and on the server where you will conduct experiments (i.e., Firefly), you are not an administrator for both machines. As such, for this course it is ok if different versions are used. Nevertheless, you need to include the hardware and software details for both machines in your report.
4. To measure the parallelism of your implementation, you **have to** use a machine with at least 4 *physical* cores. If you do not have this kind of machine, please contact the course assistant to give you access to an appropriate machine. Please make sure you use all the logical cores available in your machine for the experiments.
5. The code used in the exercise session on benchmarking is written in a needlessly complex style. For your own benchmarking program, do not make your code more complex than necessary. In addition, the code used in the benchmarking program of the exercise session contains Java lambdas. As explained during the exercise sessions, we explicitly advise against using Java lambdas in your Fork/Join project, as this may (inexplicably) impact the speed-ups of the Fork/Join framework (i.e., you might obtain bad performance results).

## 5. Further Reading

The literature on performance benchmarking is ample. For more details on benchmarking software, and in particular JVM applications, check the following work:

1. The *SIGPLAN Empirical Evaluation Checklist*, which is attached to the end of this file.
2. Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 21-25, 2007, Montreal, Quebec, Canada, pages 57-76. ACM, 2007. DOI: <https://doi.org/10.1145/1297027.1297033>
3. Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proceedings of the ACM Programming Languages. OOPSLA*, Article 52 (October 2017), 27 pages. DOI: <https://doi.org/10.1145/3133876>
4. Some useful pointers on JMH:
  - Julien Ponge. Avoiding Benchmarking Pitfalls on the JVM. Oracle Technical Article, 2014, <https://www.oracle.com/technical-resources/articles/java/architect-benchmarking.html> (Captured March 2021)
  - Jakob Jenkov. Tutorial on JMH - Java Microbenchmark Harness, 2019, <http://tutorials.jenkov.com/java-performance/jmh.html> (Captured March 2021)



## A. Tools for Performance Evaluation on a Server

Often, you will have to run the performance evaluation on a powerful server, allowing you to examine e.g., the scalability of your program. We assume that you will connect to this server via the command line, using *SSH* (secure shell). In this appendix, we describe some common command-line tools that can help you to benchmark on a Unix-based server. The use of these tools is of course optional, but knowing them may aid your evaluation, allowing you to efficiently use your time on the server. For clarity, all commands in this section are written in a monospaced and underlined font. They must be used verbatim, hence, do not change the punctuation of the commands.

### A.1. Checking Server Usage

When performing a performance evaluation, you will want to be the sole user of the server to avoid your measurements being influenced by other users. Therefore, before starting your experiments, ensure that no other users are present on the server. Using the w command, you can see which users are logged in on the server<sup>6</sup>. Normally, you only want to see your username here.

### A.2. Checking CPU Usage

When performing an evaluation of a (parallel) program, it may be useful to see how well the cores of the machine are used. To this end, you can make use of a process monitor, such as the task manager on Windows or the activity monitor on macOS. On the command line, you can use htop, which allows you to inspect the load of all cores, memory usage, and lists all processes. As htop also lists all processes currently active on the machine (and the corresponding accounts), you can also use it to verify whether no other users are active on the server.

### A.3. Specifying File Paths

For your performance evaluation, you might need to use datasets located in a specific folder or you may want to write your results to a file. In both cases, you will need to write down the file path (in your program), specifying where a specific file is to be found or needs to be created. In Unix, both absolute and relative file paths exist. You can discriminate between them by looking at their prefix:

- Paths starting with / are called absolute paths. They specify where a file can be found starting from the root of the file hierarchy.
- Paths starting with ~/ are relative w.r.t. your home directory. Suppose your home directory is located at /Users/me (absolute path) and contains a folder Recipes, then the absolute path to this folder is /Users/me/Recipes and the relative path w.r.t. your home directory is ~/Recipes.
- Paths starting with ./ are relative w.r.t. your current working directory. Hence, continuing the example, when you are working in your home directory, you can specify the Recipes folder by means of the path ./Recipes.
- Paths starting with ../ are relative w.r.t. the parent directory of your current working directory. Hence, when you are in your home directory, you could also refer to your Recipes folder using the (not very useful) path ../me/Recipes. Using this command, you can easily go up several levels. For example, you could also refer to your Recipes folder using the path ../../Users/me/Recipes.

More information w.r.t. file paths in Unix can be found online, e.g., [here](#).

When possible, use absolute pathnames. This avoids issues, e.g., when executing your Java program from within a different directory. Note that when navigating directories from the command line, tab completion is enabled, facilitating moving between directories.

---

<sup>6</sup>In case terminal multiplexers such as Screen are installed on the server, more complex commands (such as ps aux | grep -E "SCREEN|tmux") may be needed as well. This example command lists all active Screen and tmux processes on the server. Normally, you don't want to see any process listed here. When benchmarking on the Firefly server, you normally do not need to use these more involved commands (hence, the command w should suffice).

## A.4. Keeping your Program Running

When running your program using SSH, it is very important to remain connected to the server. This is because when the connection breaks, e.g., due to a temporary connection issue, the server will kill the SSH session and your program will be terminated. Hence, you need to remain connected to the server continuously whilst running your benchmarks, even when they take a lot of time. As connection errors are not rare, this is quite problematic.

A solution to this issue is provided by `tmux`, a so-called *terminal multiplexer*, which allows your benchmarks to continue running even when your connection to the server drops. When using `tmux`, you can reconnect to the server after a connection break and your programs will still be running! As `tmux` is a very powerful tool, we here present its most fundamental commands<sup>7</sup>.

`tmux` allows you to create multiple so-called *virtual terminals* within your ‘physical’ terminal. To start a `tmux session`, just run the `tmux` command. The new session contains a single *window*, in which you can see a new virtual terminal. This terminal works just like a regular terminal, except some extra commands become available to interact with `tmux` itself (to manage your virtual terminals). You can create multiple windows, to allow easy multitasking. Additionally, every window can be split into multiple *panes* (which are shown next to one another in the same window). By default, a window has one pane and each pane runs a virtual terminal.

To interact with `tmux` itself, you need to press `ctrl+b`, followed by a key command. Some useful key commands are the following:

- ? Shows a list of all possible commands. You can go back to the virtual terminal by pressing `q`.
- c Creates a new virtual window containing one pane. The green status bar at the bottom of the screen lists the windows you have created (every window has a number).
- <number> Opens the window identified by the given number.
- p Navigates to the previous window.
- n Navigates to the next window.
- & Closes the current window. A virtual terminal can also be closed using the `exit` command, just like a ‘normal’ terminal.
- d *Detaches* a session.

The most interesting feature of `tmux` is the possibility to *detach* a session. This causes you to return to your original ‘physical’ terminal, but all virtual windows keep running in the background: `tmux` will keep your session active until you explicitly close it. Hence, after detaching a session, you can close your connection to the server whilst your program keeps running. The same thing happens when your SSH connection breaks: `tmux` will keep your session active and your benchmarking program running. Hence, every program running in a virtual terminal is *protected* from connection errors and will be kept active until you close it.

When you reconnect to the server, you can use `tmux ls` to see which sessions are active, and use `tmux attach -t <name>` to *attach* (reopen) the session with a given name (the name of a session is listed before the colon when using `tmux ls`, by default this is just a number).





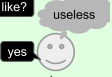

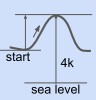






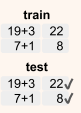








**Important:** As `tmux` keeps your sessions active, you are responsible for closing your all your `tmux` sessions when you finish working on the server. Otherwise, your program(s) will keep running, possibly prohibiting other people from using the server, e.g., to run their own performance evaluations. Use `tmux ls` to verify that no more `tmux` sessions are active when you are finishing your work on the server. If some sessions are still active, end them (e.g., by attaching and executing `exit` in each of them).

<sup>7</sup>More information can be found in the `tmux` manual, [online](#), or in the assignment of the lab session on Firefly.



# SIGPLAN Empirical Evaluation Checklist

This checklist is meant to *support* informed judgement, not *supplant* it.

Clearly Stated Claims Example Violations	 <p>Claims not explicit</p> <p>Claims must be explicit in order for the reader to assess whether the empirical evaluation supports them. Missing claims cannot possibly be assessed. Claims should also aim to state not just what is achieved but how.</p>	Relevant Metrics Example Violations	 <p>Indirect or inappropriate proxy metric</p> <p>Proxy metrics can substitute for direct ones only when the substitution is clearly, explicitly justified. For example, it would be misleading and incorrect to report a reduction in cache misses to claim actual end-to-end performance or energy consumption improvement.</p>
	 <p>Claims not appropriately scoped</p> <p>The truth of a claim should clearly follow from the evidence provided. Claims that are not fully supported mislead readers. 'Works for all Java' is over-broad when based on a subset of Java. Other examples are 'works on real hardware' when evaluating only with (unrealistic) simulation, and 'automatic process' when requiring human intervention.</p>		 <p>Fails to measure all important Effects</p> <p>All important effects should be measured to show the true cost of a system. For example, compiler optimizations may speed up programs at the cost of drastically increasing compile times of large systems, so the compile time should be measured as well as the program speedup. Failure to do so distorts the cost/benefit of the system.</p>
	 <p>Fails to acknowledge limitations</p> <p>A paper should acknowledge its limitations to place the scope of its results in context. Stating no limitations at all, or only tangential ones, while omitting the more relevant ones may mislead the reader into drawing overly-strong conclusions. This could hold back efforts to publish future improvements, and may lead researchers down wrong paths.</p>		 <p>Insufficient information to repeat</p> <p>Experiments evaluating an idea need to be described in sufficient detail to be repeatable. All parameters (including default values) should be included, as well as all version numbers of software, and full details of hardware platforms. Insufficient information impedes repeatability and comparison of future ideas and can hinder scientific progress.</p>
Suitable Comparison Example Violations	 <p>Fails to compare against appropriate baseline</p> <p>Empirical evidence for a claim that a technique/system improves upon the state-of-the-art should include a comparison against an appropriate baseline. The lack of a baseline means empirical evidence lacks context. A 'straw man' baseline that is misrepresented as state-of-the-art is also problematic, as it would inflate apparent benefit.</p>	Appropriate and Clear Experimental Design Example Violations	 <p>Unreasonable platform</p> <p>The evaluation should be on a platform that can reasonably be said to match the claims; otherwise, the results of the evaluation will not fully support the claims. For example, a claim that relates to performance on mobile platforms should not have an evaluation performed exclusively on servers.</p>
	 <p>Comparison is unfair</p> <p>Comparisons to a competing system should not unfairly disadvantage that system. Doing so would inflate the apparent advantage of the proposed system. For example, it would be unfair to compile the state-of-the-art baseline at -O0 optimization level, while using -O3 for the proposed system.</p>		 <p>Ignores key design parameters</p> <p>Key parameters should be explored over a range to evaluate sensitivity to their settings. Examples include the size of the heap when evaluating garbage collection and the size of caches when evaluating a locality optimization. All expected system configurations (e.g., from warmup to steady state) should be considered.</p>
Principled Benchmark Choice Example Violations	 <p>Inappropriate suite</p> <p>Evaluations should be conducted using appropriate established benchmarks where they exist so that claimed results are more likely to generalize. Not doing so may yield results that are not sufficiently general. Established suites should be used in context; e.g. it would be wrong to use a single-threaded suite for studying parallel performance.</p>	Appropriate Presentation of Results Example Violations	 <p>Gated workload generator</p> <p>Load generators for typical transaction-oriented systems should be 'open loop', to generate work independent of the performance of the system under test. Otherwise, results are likely to mislead because real-world transaction servers are usually open-loop.</p>
	 <p>Unjustified use of non-standard suite(s)</p> <p>The use of standard benchmark suites improves the comparability of results. However, sometimes a non-standard suite, such as one that is subsetted or homegrown, is the better choice. In that case, a rationale, and possible limitations, must be provided to demonstrate why using a standard suite would have been worse.</p>		 <p>Tested on training set</p> <p>When a system aims to be general but was developed with close consideration of specific examples, it is essential that the evaluation explicitly perform cross-validation, so that the system is evaluated on data distinct from the training set. For example, a static analysis should not be exclusively evaluated on programs used to inform its development.</p>
	 <p>Kernels instead of full applications</p> <p>Kernels can be useful and appropriate in a broader evaluation. However, a claim that a system benefits applications should be tested on such applications directly, and not only on micro-kernels, which may lack important characteristics of full applications.</p>		 <p>Misleading summary of results</p> <p>The summary of the results must reflect the full range of their character to avoid misleading the reader. For example, it is not appropriate to summarize speedups of 4%, 6%, 7%, and 49% as 'up to 49%'. Instead, the full distribution of results must be reported.</p>
Adequate Data Analysis Example Violations	 <p>Insufficient number of trials</p> <p>Modern systems with non-deterministic performance properties may require many trials (e.g., of a single time measurement) to characterize their behavior adequately. Failure to do so risks treating noise as signal. Similarly, more trials may be needed to get the system into an intended state (e.g., into a steady state that avoids warm-up effects).</p>	Appropriate Presentation of Results Example Violations	 <p>Inappropriately truncated axes</p> <p>Graphs provide a visual intuition about a result. A truncated graph (with an axis not including zero) will exaggerate the importance of a difference. 'Zooming' in to the interesting range of an axis can sometimes aid exposition, but should be pointed out explicitly to avoid being misleading.</p>
	 <p>Inappropriate summary statistics</p> <p>Summary statistics such as mean and median can usefully characterize many results. But they should be selected carefully, because each statistic presents an accurate view only under appropriate circumstances. An inappropriate summary may amplify noise or hide an important trend.</p>		 <p>Ratios plotted incorrectly</p> <p>Incorrectly plotted ratios badly mislead visual intuition. For example, 2.0 and 0.5 are reciprocals, but their linear distance from 1.0 does not reflect that, so plotting those numbers on a linear scale significantly distorts the result. This misleading effect can be avoided either by using a log scale or by normalizing to the lowest (highest) value.</p>
	 <p>No data distribution reported</p> <p>A measure of variability (e.g., variance, std. deviation, quantiles) and/or confidence intervals is needed to understand the distribution of the data. Reporting just a measure of central tendency (e.g., a mean or median) can mislead the reader, especially when the distribution is bimodal or has significant variance.</p>		 <p>Inappropriate level of precision</p> <p>Measurements reported at a proper level of precision reveal relevant information. Under-precise reports may hide such information, and over-precise ones may overstate the accuracy of a measurement and obscure what is relevant. For example, reporting '49.9%' when the experimental error is +/- 1% overstates the level of precision of the result.</p>

# Notes

**Claims not Explicit** This includes *implied* generality — implied: ‘works for all Java’, but actually only on a static subset; implied: ‘works on real hardware’, but actually only works in simulation; implied: ‘automatic process’, but in fact required non-trivial human supervision; implied: ‘only improves the systems’ performance’, but actually the approach requires breaking some of the system’s expected behavior.

**Fails to Acknowledge Limitations** One concern we have heard multiple times is that this example, previously titled *Threats to validity*, is not useful. The given reason is that *threats to validity* sections in software engineering papers often mention threats of little significance while ignoring real threats. This is unfortunate, but does not eliminate the need to clearly scope claims, highlighting important limitations. For science to progress, we need to be honest about what we have achieved. Papers often make, or imply, overly strong claims. One way this is done is to ignore important limitations. But doing so discourages or undervalues subsequent work that overcomes those limitations because that progress is not appreciated. Progress comes in steps, rarely in leaps, and we need those steps to be solid and clearly defined.

**Fails to Compare Against Appropriate Baseline** The baseline could also be an unsophisticated approach to the same problem, e.g., a fancy testing tool is usefully compared against one that is purely random, in order to see whether it does better.

**Inappropriate Suite** This includes misuse of incorrect established suite e.g. use of SPEC CINT2006 when considering parallel workloads.

**Unjustified Use of Non-Standard Suite(s)** A concern we heard was that use of standard suites may lead to work that overfits to that benchmark. While this is a problem in theory, and is well known from the machine learning community, our experience is that PL work more often has the opposite problem. Papers we looked at often subset a benchmark, or cherry-picked particular programs. Doing so calls results into question generally, and makes it hard to compare related systems across papers. We make progress more clearly when we can measure it. Good benchmark suites are important, since only with them can we make generalizable progress. Developing them is something that our community should encourage.

Note that ‘benchmark’ in this category includes what is measured and the parameters of that measurement. One example of an oft-unappreciated benchmark parameter is timeout choice.

**Inappropriate Summary Statistics** As particular best practices: The geometric mean should only be used when comparing values with different ranges, and the harmonic mean when comparing rates. When distributions have outliers, a median should be presented. There are many excellent resources available, including: [Common errors in statistics \(and how to avoid them\)](#). (Phillip I Good and James W Hardin, 2012), [What is a P-value anyway?: 34 stories to help you actually understand statistics](#). (Andrew Vickers, 2010), and [Statistical misconceptions](#). (Schuyler W Huck, 2009).

**Ratios Plotted Incorrectly** For example, if times for a and b are 4 sec and 8 sec respectively for benchmark x and 6 sec and 3 sec for benchmark y, this could be shown as a/b (0.5, 2.0) or b/a (2.0, 0.5), where 1.0 represents parity. Although the results (0.5 & 2.0) are reciprocals, their distance from 1.0 on a linear scale is different by a factor of two (0.5 & 1.0), overstating the speedup. This is why showing ratios (or percentages) greater than 1.0 (100%) and less than 1.0 (100%) on the same linear scale is visually misleading.

## FAQ

**Why a checklist?** Our goal is to help ensure that current, accepted best practices are followed. Per the [Checklist Manifesto](#), checklists help to do exactly this. Our interest is the good practices for carrying out empirical evaluations as part of PL research. While some practices are clearly wrong, many require careful consideration: Not every example under every category in the checklist applies to every evaluation — expert judgment is required. The checklist is meant to assist expert judgment, not substitute for it. ‘Failure isn’t due to ignorance. According to best-selling author Atul Gawande, it’s because we haven’t properly applied what we already know.’ We’ve kept the list to a single page to make it easier to use and refer back to.

**Why now?** When best practices are not followed, there is a greater-than-necessary risk that the benefits reported by an empirical evaluation are illusory, which

harms further progress and stunts industry adoption. The members of the committee have observed many recent cases in which practices in the present checklist are not followed. Our hope is that this effort will help focus the community on presenting the most appropriate evidence for a stated claim, where the form of this evidence is based on accepted norms.

**Is use of the checklist going to be formally integrated into SIGPLAN conference review processes?** There are no plans to do so, but in time, doing so may make sense.

**How do you see authors using this checklist?** We believe the most important use of the checklist is to assist authors in carrying out a meaningful empirical evaluation.

**How do you see reviewers using this checklist?** We also view the checklist as a way to remind reviewers of important elements of a good empirical evaluation, which they can take into account when carrying out their assessment. However, we emphasize that proper use of the checklist requires nuance. Just because a paper has every box checked doesn’t mean it should be accepted. Conversely, a paper with one or two boxes unchecked may still merit acceptance. Even whether a box is checked or not may be subject to debate. The point is to organize a reviewer’s thinking about an empirical evaluation to reduce the chances that an important aspect is overlooked. When a paper fails to check a box, it deserves some scrutiny in that category.

**How did you determine which items to include?** The committee examined a sampling of papers from the last several years of ASPLOS, ICFP, OOPSLA, PLDI, and POPL, and considered those that contained some form of empirical evaluation. We also considered past efforts examining empirical work (Gernot Heiser’s “[Systems Benchmarking Crimes](#)”, the “[Pragmatic Guide to Assessing Empirical Evaluations](#)”, and the “[Evaluate Collaboratory](#)”). Through regular discussions over several months, we identified common patterns and anti-patterns, which we grouped into the present checklist. Note that we explicitly did not intend for the checklist to be exhaustive; rather, it reflects what appears to us to be common in PL empirical evaluations.

**Why did you organize the checklist as a series of categories, each with several examples?** The larger categories represent the general breadth of evaluations we saw, and the examples are intended to be helpful in being concrete, and common. For less common empirical evaluations, other examples may be relevant, even if not presented in the checklist explicitly. For example, for work studying human factors, the Adequate Data Analysis category might involve examples focusing on the use of statistical tests to relate outcomes in a control group to those in an experimental group. More on this kind of work below.

**Why did you use checkboxes instead of something more nuanced, like a score?** The boxes next to each item are not intended to require a binary “yes/no” decision. In our own use of the list, we have often marked entries as partially filling a box (e.g., with a dash to indicate a “middle” value) or by coloring it in (e.g., red for egregious violation, green for pass, yellow for something in the middle).

**What about human factors or other areas that require empirical evaluation?** PL research sometimes involves user studies, and these are different in character than, say, work that evaluates a new compiler optimization or test generation strategy. Because user studies are currently relatively infrequent in the papers we examined, we have not included them among the category examples. It may be that new, different examples are required for such studies, or that the present checklist will evolve to contain examples drawn from user studies. Nonetheless, the seven category items are broadly applicable and should be useful to authors of any empirical evaluation for a SIGPLAN conference.

**How does the checklist relate to the artifact evaluation process?** Artifact evaluation typically occurs after reviewing a paper, to check that the claims and evidence given in the paper match reality, in the artifact. The checklist is meant to be used by reviewers while judging the paper, and by authors when carrying out their research and writing their paper.

**How will this checklist evolve over time?** Our manifesto is: Usage should determine content. We welcome feedback from users of the checklist to indicate how frequently they use certain checklist items or how often papers reviewed adhere to them. We also welcome feedback pointing to papers that motivate the inclusion of new items. As the community increasingly adheres to the guidelines present in the checklist, the need for their inclusion may diminish. We also welcome feedback on presentation: please share points of confusion about individual items, so we can improve descriptions or organization.

Feedback via: <http://www.sigplan.org/Resources/EmpiricalEvaluation/>