RESEARCH ARTICLE OPEN ACCESS

Profiler-Guided Detection and Refactoring of Runtime Hotspots in Java Applications

Sumit Kumar[1], Jasvir Singh[2]

[1,2]Department of Computer Science and Engineering, Punjabi University, Patiala, Punjab, India.

Abstract - *Runtime hotspots in Java applications are usually chased down by intuition rather than by measurement. This study profiles three applications chosen to represent three structurally different hotspot categories: a Sudoku backtracking solver that is algorithmic and CPU-bound, a headless Snake simulation that is allocation-bound, and an invoice generator dominated by string building and file input/output. JDK Flight Recorder is the only profiling instrument used, and each hotspot receives a refactoring matched to its category. Execution time is measured independently across ten iterations per configuration and improves significantly in every application, by 38.7 percent for the invoice generator, 52.1 percent for the solver and 67.4 percent for the simulation. Peak heap, allocation rate and garbage-collection pause time are drawn from a single aggregated recording per configuration, so they are presented as descriptive point estimates and carry no significance test, and CPU time is reported as a proxy equal to execution time because no independent instrumentation was implemented. The invoice generator improves on all five metrics and rests on the largest volume of underlying events. The other two applications each yield one result opposing the direction their refactoring targeted, which the present recording design cannot separate from sampling artefact. Every refactoring was validated against byte-identical output before measurement began. The work shows that a profiler-first workflow reliably identifies and removes hotspots, while also demonstrating how measurement design determines which performance claims a study can legitimately support.*

Keywords - *allocation reduction, garbage collection, JDK Flight Recorder, refactoring, runtime profiling*

1. Introduction

A method that looks harmless during code review can dominate a profiler trace the moment an application runs against a realistic workload. Developers without a profiling habit tend to guess at what needs optimising, and the instruction to refactor the slow part is often treated as self-evidently correct. What usually gets skipped is the check afterwards. Did the change move the metric it was aimed at, did it move a different metric instead, or did it do neither of those things?

JDK Flight Recorder ships inside every modern JDK and records processor, allocation and garbage-collection events at low overhead, which makes it a sensible default instrument for a profiler-first workflow. [1] VisualVM was the tool originally chosen for this study, with its Monitor and Visual GC tabs read live during each run. [2] That plan was abandoned partway through data collection. Live-attach timing proved unreliable for applications whose measured work finishes in tens of milliseconds, and manually reading four values per configuration off a graphical interface, repeated across six configurations, did not fit the time available. The workflow reported here replaces the manual step with a Flight Recorder session and a short command-line pipeline that extracts the same figures automatically. This is a practical substitution and not a methodological improvement, and its consequences for statistical validity are set out openly in Section 3 and Section 5.

The study covers three Java applications, each standing in for a structurally different hotspot type: algorithmic and processor-bound, allocation-bound, and string-building with file input/output. Each application was profiled, refactored using a technique matched to its category, and measured again across five metrics over ten iterations per configuration. The three refactoring techniques, namely removal of algorithmic redundancy, reduction of object allocation, and buffered input/output combined with accumulation through a string builder, are standard patterns drawn from the refactoring catalogue and from the green software engineering literature. [3], [14] In that literature, allocation-focused and input/output-focused changes are studied more often for their energy effect than for their effect on wall-clock time. [4] This paper measures runtime and resource metrics only. Energy is treated strictly as future work and is not implied anywhere in the title or the body.

Two contributions follow. The first is empirical: three hotspot categories are treated inside one controlled design, whereas most comparable case studies apply a single technique to a single application. The second is methodological. The study reports honestly on which of its own five metrics can support an inferential claim and which cannot, and it traces that division back to a specific and repairable choice in the recording design instead of leaving the weaker metrics to sit unmarked beside the stronger ones.

Section 2 reviews related work on profiling instruments, refactoring categories and benchmarking method. Section 3 describes the materials and methods actually used, including the points where they depart from the original plan. Section 4 reports and discusses

results. Section 5 examines threats to validity, and Section 6 concludes.

2. Related Work

2.1. Profiling Instrumentation

JDK Flight Recorder has grown from a commercial add-on into a standard low-overhead profiling facility built into the JDK. It exposes processor, allocation and garbage-collection events through both a graphical front end, JDK Mission Control, and a command-line interface. [1] The present study uses the command-line interface exclusively, since the objective was an automatable pipeline rather than interactive exploration. VisualVM remains a capable general-purpose tool and was the instrument originally intended here, but it suited neither short-running workloads nor scripted repeated-measures collection, and it was replaced during the project for the reasons given in Section 3.3. [2]

2.2. Refactoring Categories and Resource Behaviour

The techniques applied in this study come from the established refactoring catalogue and from work on energy-aware software engineering. [14], [15] Sahin, Pollock and Clause examined how code refactorings affect energy usage and found that refactorings can move consumption in either direction, which argues against assuming that a transformation known to improve one property leaves the others untouched. [4] The same group later showed that code transformations applied for entirely unrelated reasons still register measurable effects on resource behaviour. [5] Pereira and colleagues reported large efficiency differences across programming languages and traced much of the variation to memory-management behaviour, which supports treating allocation reduction as a category in its own right rather than folding it into general processor optimisation. [3]

Allocation behaviour under generational collection has a long empirical record. Dieckmann and Holzle characterised object lifetime and size distributions in Java workloads and found that objects are typically small and short-lived, so that churn imposes cost beyond the allocation site itself. [6] Blackburn, Cheng and McKinley quantified how collector policy interacts with program behaviour and demonstrated that collection cost is not a fixed overhead but a function of the allocation profile the application presents. [7] The Garbage-First collector, which is the default under the JDK release used here, partitions the heap into regions and schedules evacuation against a pause-time goal, so an individual young-generation pause reflects heap occupancy and scheduling state at the instant it fires. [8] That property matters directly for how the pause-time results in Section 4 can be read.

Inefficiencies in string building and file input/output are covered less often as a distinct category, although repeated concatenation inside a loop and unbuffered stream access are both flagged as cheap to fix and expensive to leave in place. [4], [15]

2.3. Benchmarking Methodology

Guidance on managed-language benchmarking is consistent on two points. Georges, Buytaert and Eeckhout showed that single-run comparisons of Java programs are unreliable because just-in-time compilation, thread scheduling and collection introduce run-to-run variation, and they recommended repeated measurement with paired statistical treatment. [10] Kalibera and Jones extended this by showing how many repetitions are actually needed at each level of an experiment and by arguing that a reported difference without an accompanying estimate of variation cannot be assessed by a reader at all. [11] Mytkowicz and colleagues demonstrated that measurement bias arising from apparently innocuous features of an experimental setup is both commonplace and large enough to reverse a conclusion. [12] Runeson and Host, together with Wohlin and colleagues, supply the case-study and experimentation frameworks used here for scoping and for the validity discussion. [9], [13]

This paper follows that guidance for execution time, where ten independent measurements exist per configuration. It does not follow it for peak heap, allocation rate or pause time, because the recording design adopted here yields a single aggregate value per configuration for those three. Section 5 treats that as a limitation of the design and not as an oversight in the analysis.

2.4. Research Gap

Empirical work that applies more than one refactoring technique across more than one application inside a single study is comparatively rare. Most published case studies pair one technique with one program, which makes it difficult to say whether an observed gain reflects the technique, the workload or the interaction between them. [13], [15] The three-application design used here is intended to address that gap while remaining small enough to control.

3. Materials and Methods

3.1. Application Selection

Three applications were selected to span three structurally different hotspot categories rather than three variations on a single category, following case-study guidance on purposive rather than convenience selection. [9] Their workloads and the refactoring applied to each are summarised in **Table 1**. Each application is small enough to be understood in full, which matters because the functional-equivalence check described in Section 3.6 has to be exhaustive rather than sampled.

Table 1. Applications, hotspot types and refactorings applied

Application	Workload	Hotspot type	Refactoring applied
Sudoku Solver	Backtracking solve of 50 randomly generated 9x9 puzzles per run	Algorithmic, recursive, CPU-bound	Per-candidate row, column and box re-scan replaced with incrementally maintained bitmasks
Snake (headless)	5,000-tick AI-controlled simulation per run, no rendering	Allocation-bound, GC-heavy	Per-tick list and coordinate-object allocation replaced with primitive integer arrays and double-buffered state
Invoice Generator	Format and write invoices for 40,000 orders read from a CSV order file	String-building and I/O-bound	Per-line concatenation replaced with one StringBuilder per invoice; unbuffered reader and writer replaced with buffered equivalents

3.2. Experimental Environment

All measurements were taken on a single machine under the configuration listed in **Table 2**. No explicit heap-size or collector flags were set, so the Garbage-First collector operated under default ergonomics for the JDK release in use. [8] Holding the environment fixed across all six configurations removes hardware and collector policy as sources of between-configuration difference, though it also narrows the range over which the results can be generalised.

Table 2. Experimental environment

Component	Specification
CPU	Intel Core i5-8250U at 1.60 GHz, 4 cores, 8 logical processors
RAM	7.9 GB
Operating system	Microsoft Windows 11 Home Single Language, version 10.0.26200 (build 26200), 64-bit
JDK	Oracle JDK 21.0.9 LTS, HotSpot 64-Bit Server VM (build 21.0.9+7-LTS-338)
Profiler	JDK Flight Recorder, command-line tool, settings=profile
Metric extraction	jfr print piped to a purpose-built Python parser
JVM flags	-XX:StartFlightRecording = filename → <config>.jfr, settings → profile. No explicit heap-size or collector flags were set, so default Garbage-First ergonomics applied under JDK 21

Environment values were captured after the fact through PowerShell rather than logged during the original data-collection session.

3.3. Profiling Procedure

Each configuration was launched once with Flight Recorder enabled for the full process lifetime, spanning three discarded warm-up iterations followed by ten measured iterations inside that single JVM invocation. After the process exited, the recording summary confirmed wall-clock duration, and the event printer extracted heap-summary, garbage-collection and allocation-sample events as text. [1]

A purpose-built parser then computed three quantities. Peak heap was taken as the maximum heap-used value across all heap-summary events. Total pause time was the sum of the duration field across all collection events. Allocation rate was the sum of the weight field across all allocation-sample events, divided by the wall-clock duration of the recording.

This arrangement has a direct consequence that shapes everything reported later. Peak heap, allocation rate and pause time are each one point estimate per configuration, built from one recording that spans all ten measured iterations. Execution time, by contrast, was captured independently for each of the ten iterations from inside the harness. The asymmetry is not incidental; it follows from recording once per configuration instead of once per iteration, and it determines which metrics can support inferential statistics in Section 3.8.

VisualVM was the instrument originally planned for the three resource metrics, through either its recording-snapshot view or its live monitoring tabs. It was dropped in favour of the scripted pipeline once live-attach timing proved unreliable for applications whose measured work completes in well under a second, and once reading four values per configuration off a graphical interface, six times over, proved too slow for the project schedule. [2] VisualVM contributes no number reported anywhere in this paper.

3.4. Refactoring Procedure

Each refactoring targeted the specific hotspot the profiler surfaced, and each follows a standard pattern from the refactoring catalogue. [14] The solver's candidate-validity check originally re-scanned the full row, column and three-by-three box on every backtracking step. It was rewritten to maintain three integer bitmasks incrementally, updated on each placement and each removal instead of being recomputed from scratch.

The simulation's per-tick state update originally allocated a fresh list and a fresh coordinate object for every segment on every tick. It was rewritten around primitive integer coordinate arrays with a reused, double-buffered state, so that a tick swaps on write instead of allocating on write. This is the classic remedy for short-lived object churn described in the allocation literature. [6], [7]

The invoice generator originally built each invoice through repeated string concatenation and wrote it through an unbuffered reader and writer pair. It was rewritten to accumulate each invoice in a single string builder and to read and write through buffered equivalents.

3.5. Functional Validation

Before any performance number was recorded, each refactored application was checked against its pre-refactoring version for identical output under a fixed workload. The solver's output was compared byte for byte against the baseline across all fifty puzzles. The simulation's final-state summary, covering tick count, alive or dead status, snake length, score and the head and food positions, was compared byte for byte under an identical seed; this establishes that both versions reach the same terminal state, not that every intermediate tick agrees. The invoice generator's output file was compared byte for byte over the same forty-thousand-order input. None of the three comparisons produced a difference.

3.6. Measurement Protocol

Every configuration, meaning each application in each of its two phases, ran ten measured iterations preceded by three discarded warm-up iterations inside a single JVM process. The warm-up reduces just-in-time compilation noise, as recommended for managed-language benchmarking. [10] Background system load was neither logged nor controlled, and Section 5.1 treats that as a limitation rather than assuming it away. [12]

Execution time was captured from inside the harness using the high-resolution system timer. Peak heap, allocation rate and pause time were derived from each configuration's single recording as described in Section 3.3. CPU time is reported as numerically equal to execution time, because no independent processor-time instrumentation was implemented in the harness. Rather than drop the metric, it is labelled explicitly as a proxy so the absence of an independent measurement stays visible in the results table itself.

3.7. Statistical Method

For execution time, the ten before-values and ten after-values for each application were tested for normality using the Shapiro-Wilk test. A paired-samples t-test was applied where both samples were consistent with normality, which was the case only for the invoice generator. The Wilcoxon signed-rank test was applied otherwise, covering the solver and the simulation. The significance threshold was set at 0.05. Cohen's d accompanies the paired t-test result. For the two applications analysed with the Wilcoxon test, the same effect-size statistic is produced by the analysis pipeline and is reported for consistency, but it should be read as approximate rather than strictly parametric in those cases.

Peak heap, allocation rate and pause time have no within-configuration variance under this design, because one recording produced one value per application and phase. No significance test and no effect size is therefore computed for them, and they appear as percentage differences between two single values. A difference reported without any estimate of variation cannot be judged by a reader, which is why these three metrics are labelled as point estimates wherever they appear. [11]

4. Results and Discussion

Complete results appear in **Table 3**. Of the fifteen application and metric combinations, only three rest on independent per-iteration measurement and inferential statistics, namely execution time for each application, all significant at a p-value of 0.002 or below. The three CPU-time rows duplicate those results by construction. The remaining nine rows are single point-estimate comparisons, reported descriptively and without any significance claim.

Table 3. Mean (SD) before and after refactoring, percentage change and significance where applicable, n = 10 iterations per configuration

Application	Metric	Before	After	Δ (%)	Test	p	Sig.
Sudoku Solver	Execution time (ms)	19.400 (2.119)	9.300 (0.949)	52.06	Wilcoxon	0.0020	Y
Sudoku Solver	CPU time (ms), proxy	19.400 (2.119)	9.300 (0.949)	52.06	= exec.	—	—
Sudoku Solver	Peak heap (MB)	2.80	22.00	-685.71	point est.	n/a	n/a
Sudoku Solver	Allocation rate (MB/s)	7.38	1.98	73.17	point est.	n/a	n/a
Sudoku Solver	GC pause, total (ms)	29.60	21.90	26.01	point est.	n/a	n/a
Snake (headless)	Execution time (ms)	9.500 (4.353)	3.100 (2.846)	67.37	Wilcoxon	0.0020	Y
Snake (headless)	CPU time (ms), proxy	9.500 (4.353)	3.100 (2.846)	67.37	= exec.	—	—
Snake (headless)	Peak heap (MB)	23.00	22.00	4.35	point est.	n/a	n/a
Snake (headless)	Allocation rate (MB/s)	13.30	1.78	86.62	point est.	n/a	n/a
Snake (headless)	GC pause, total (ms)	18.70	29.70	-58.82	point est.	n/a	n/a
Invoice Generator	Execution time (ms)	3125.500 (479.224)	1917.300 (296.736)	38.66	paired t-test	0.0002	Y
Invoice Generator	CPU time (ms), proxy	3125.500 (479.224)	1917.300 (296.736)	38.66	= exec.	—	—
Invoice Generator	Peak heap (MB)	93.40	93.50	-0.11	point est.	n/a	n/a
Invoice Generator	Allocation rate (MB/s)	276.70	202.07	26.97	point est.	n/a	n/a
Invoice Generator	GC pause, total (ms)	430.56	227.91	47.07	point est.	n/a	n/a

Rows marked "point est." compare two single values derived from one JFR recording per configuration and carry no significance test or effect size (Section 3.7). Rows marked "= exec." are numerically identical to the execution-time row above them, because CPU time was recorded as a proxy for execution time and not measured independently. Cohen's d for the invoice generator is 1.93.

4.1. Execution Time

Execution time improved in every application, though by noticeably different margins, as shown in **Figure 1**. The simulation gained the most at 67.4 percent, followed by the solver at 52.1 percent and the invoice generator at 38.7 percent. All three improvements are statistically significant. None of the three magnitudes should be treated as more real than the others on that basis alone, since a significance test establishes that an improvement is unlikely to be noise and says nothing about whether a 67 percent gain constitutes stronger evidence than a 39 percent one.

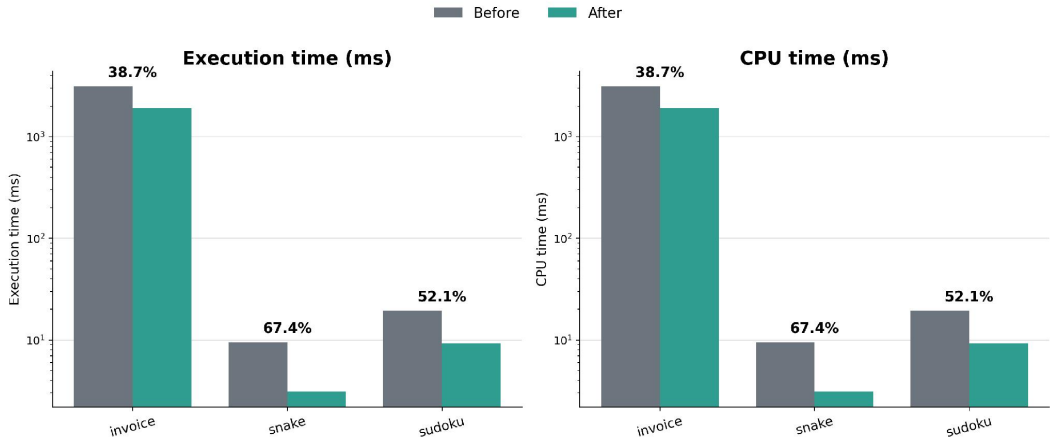

Fig. 1 Execution time and CPU time before and after refactoring, mean of 10 iterations per configuration, all three applications.

The invoice generator produced the largest absolute saving despite the smallest relative one, moving from 3125.500 milliseconds to 1917.300 milliseconds. It also produced the strongest inferential result, with a p-value of 0.0002 and a Cohen's d of 1.93, which reflects both the larger sample separation and the lower relative variance of a workload lasting seconds rather than milliseconds.

4.2. Resource-Usage Point Estimates

The point-estimate comparisons, plotted in **Figure 2**, mostly follow the direction each refactoring targeted, but not uniformly. The simulation's allocation rate fell by 86.6 percent and the invoice generator's by 27.0 percent, both consistent with allocation-focused changes. The solver's allocation rate fell by 73.2 percent even though its refactoring targeted computation rather than allocation, which is plausible enough: removing a redundant re-scan also removes whatever incidental allocation that re-scan was performing.

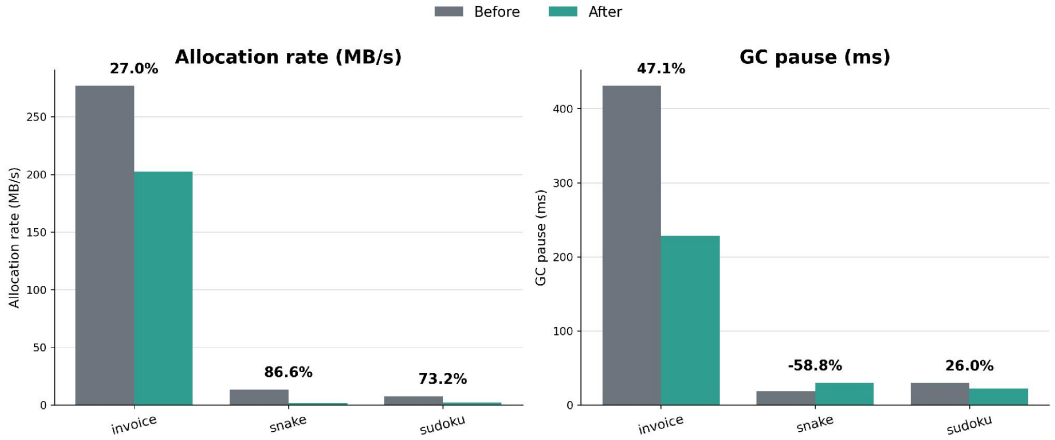

Fig. 2 Allocation rate and total GC pause time before and after refactoring, all three applications. These are single-recording point estimates and not means of 10 iterations.

Two results run against the expected direction. The solver's peak heap rose from 2.80 to 22.00 megabytes, and the simulation's total pause time rose from 18.70 to 29.70 milliseconds. Both are discussed in Section 4.3 alongside the recording-length caveat that applies specifically to them.

4.3. Within-Application Discussion

The solver's algorithmic fix improved execution and CPU time by 52.1 percent, which is consistent with replacing a full constraint re-scan by incrementally maintained bitmasks. Its peak heap moved in the opposite direction to what an algorithmic change with no intended memory effect would normally produce. Two explanations fit the available data and cannot be separated with it. Maintaining three additional integer bitmask arrays throughout the recursive solve may genuinely raise live heap relative to the unmasked baseline. Alternatively, both recordings are short enough, at three seconds and one second against roughly 190 and 90 milliseconds of actual solving work, that each captured only two heap-summary events, in which case the reported peak reflects when collection happened to sample rather than a true difference in occupancy. Allocation rate and pause time both improved in the point-estimate comparison, by 73.2 and 26.0 percent, consistent with a lighter per-candidate workload, though the same caveat about recording length applies to their reliability.

The simulation improved execution and CPU time by 67.4 percent, the largest gain of the three, consistent with removing per-tick list and object allocation in favour of primitive arrays. Allocation rate fell by 86.6 percent, in the expected direction and at a plausible magnitude for that change. Pause time, however, rose by 58.8 percent, which is the opposite of what an allocation-reduction refactoring should produce. Both recordings captured exactly one collection event, so the comparison rests on one baseline pause against one refactored pause rather than on an aggregate. A single young-generation pause under a region-based collector can vary for reasons unrelated to the code change, including heap occupancy at the moment it fires and whatever else the virtual machine was doing at that instant. [8] One event on each side cannot distinguish a real effect from noise, and this result is therefore recorded as an open question rather than absorbed into the paper's headline claims.

The invoice generator improved on every metric and by broadly comparable margins, between 27.0 and 47.1 percent for the three point-estimate metrics and 38.7 percent for execution time. That pattern is consistent with a refactoring that addresses a broad, diffuse inefficiency rather than one isolated hotspot. Peak heap barely moved, from 93.40 to 93.50 megabytes, and that near-flat result is itself informative: the refactoring changed how output was assembled and written, not how much live data the process held at any one moment, and the heap figure reflects exactly that. This application's recordings ran between 28 and 46 seconds and captured hundreds of collection events and thousands of allocation samples, two to three orders of magnitude more underlying data than the other two, so its point estimates rest on considerably more signal even though they remain single aggregate values.

4.4. Cross-Application Comparison

Execution time, the only metric measured with independent per-iteration variance in this study, improved significantly in all three applications, with gains ranging from 38.7 to 67.4 percent. CPU time mirrors those results exactly because it was recorded as equal to execution time and not instrumented separately, and it should not be read as a second line of evidence.

Among the three point-estimate metrics, the invoice generator's results are the most internally consistent and rest on much the largest volume of underlying events. The other two applications each produced one result opposing the direction their refactoring targeted, heap growth in the solver and pause growth in the simulation, and a design with one recording per configuration cannot adjudicate either. A tempting summary would treat hotspot type as a predictor of where refactoring gains concentrate. The dataset does not support a claim that strong. What it supports is narrower: each refactoring produced a significant and substantial improvement in execution time, and the resource-usage metrics moved mostly but not entirely in the expected direction, with both exceptions falling in the two applications whose recordings carried the least underlying event data.

4.5. Relation to Prior Work

The magnitude of the allocation-rate reductions in the simulation and the invoice generator sits comfortably alongside prior findings on allocation under generational collection, where cutting short-lived object churn yields gains disproportionate to the volume of code changed. [6], [7] The solver's execution-time result matches the general expectation that processor-bound recursive workloads respond well to redundancy removal. Its heap result does not fit the usual assumption that such a change is memory-neutral, and it is left open here rather than explained away.

The broader observation that a refactoring can move a resource metric in an unintended direction is itself consistent with the energy-refactoring literature, where transformations applied for one reason regularly register effects on properties the developer was not targeting. [4], [5] Given that nine of the fifteen rows in Table 3 are single point comparisons, this paper treats its results as broadly consistent with that body of work rather than as independent confirmation of any specific published magnitude. [3]

5. Threats to Validity

5.1. Internal Validity

Warm-up iterations mitigate just-in-time compilation noise for execution time, which rests on ten independent measurements per configuration. [10] Peak heap, allocation rate and pause time come from a single recording per configuration rather than from repeated independent measurements. This is a more serious constraint than a small sample, because additional runs alone would not resolve it without also recording separately for each iteration. Background process load on the test machine was not independently logged during collection, and setup features of exactly this kind have been shown to bias systems measurements in ways that are easy to miss and large enough to matter. [12]

5.2. External Validity

Three applications on one JDK release running on one machine is a deliberately narrow slice of the Java ecosystem. The design is a case study intended to compare hotspot types under controlled conditions, not a basis for generalising any specific percentage to arbitrary Java applications. [9] The applications are also small and single-threaded, so nothing here speaks to concurrent workloads or to long-running server processes where collection behaviour differs substantially.

5.3. Construct Validity

CPU time is operationalised as equal to execution time rather than measured independently, so the two rows for each application in Table 3 must not be read as separate lines of evidence for the same result. Peak heap, allocation rate and pause time are each derived from one aggregated recording per configuration. For the solver and the simulation specifically, the measured workload is short relative to the total duration of each recording, roughly 90 to 190 milliseconds of solving or simulation inside a recording of one to three seconds, which yields as few as one to six relevant events per metric. Construct validity for those three metrics is correspondingly weak in those two applications and considerably stronger in the invoice generator, whose longer recordings captured orders of magnitude more events.

5.4. Conclusion Validity

Significance testing is reported, and is valid, only for execution time and its CPU-time proxy, where ten independent measurements exist per configuration. No significance test is reported for the other three metrics, because a single point estimate per configuration provides no basis for computing one. This is stated plainly as a limitation of the measurement design rather than left as an unreported null result, since a performance difference presented without any estimate of its variation cannot be evaluated by a reader at all. [11] The effect sizes reported alongside the two Wilcoxon results should be read as approximate for the reason given in Section 3.7.

6. Conclusion and Future Work

This paper tested a profiler-guided workflow built on JDK Flight Recorder across three Java applications representing different hotspot categories. All three refactorings were validated for functional equivalence, through byte-identical output under a fixed workload or seed, before any performance measurement was taken. Execution time, the only metric measured with independent per-iteration variance, improved significantly in every application, with gains between 38.7 and 67.4 percent. Peak heap, allocation rate and pause time were measured from a single aggregated recording per configuration and are reported as descriptive point estimates. Two of those point estimates ran counter to the direction the corresponding refactoring targeted, namely an increase in the solver's peak heap and an increase in the simulation's pause time, and the present design cannot establish whether either is a real effect or an artefact of a short recording containing very few events.

The most direct extension is to move from one recording per configuration to one recording per individual iteration. That single change would put peak heap, allocation rate and pause time on the same statistical footing execution time already occupies here, and it would allow the two counterintuitive results above to be tested rather than left open. A second extension is to replace the CPU-time proxy with an independent measurement taken through the operating-system management bean around each iteration, since CPU time and execution time are currently identical by construction rather than by measurement.

Beyond those two repairs, the application set could be widened to further hotspot categories and to concurrent workloads, the same refactorings could be re-tested under a different collector to see whether the pause-time behaviour persists, and energy consumption could be instrumented directly, for instance through a source-level power monitoring agent, to test whether these runtime and allocation gains translate into energy savings. [16] The wider lesson from this study is not about any one of the three percentages reported. It is that a profiler-first workflow is only as trustworthy as the measurement design behind it, and that stating precisely which claims a design can and cannot support is part of reporting the result rather than an apology attached to it.

Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this paper.

Funding Statement

This research received no specific grant from any funding agency in the public, commercial or not-for-profit sectors.

Acknowledgments

The authors thank the reviewers for comments that improved the presentation of the measurement design and its limitations.

References

[1] Oracle Corporation, "Java Platform, Standard Edition Flight Recorder API Programmer's Guide, Release 21," Oracle Corporation, Redwood Shores, CA, USA, 2023. [Online]. Available: https://docs.oracle.com/en/java/javase/21/jfapi/

[2] VisualVM: All-in-One Java Troubleshooting Tool, 2023. [Online]. Available: https://visualvm.github.io/

[3] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking Programming Languages by Energy Efficiency," Science of Computer Programming, vol. 205, 2021, Art. no. 102609.

[4] C. Sahin, L. Pollock, and J. Clause, "How Do Code Refactorings Affect Energy Usage?," Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Torino, Italy, pp. 1-10, 2014.

[5] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W. G. J. Halfond, and J. Clause, "How Does Code Obfuscation Impact Energy Usage?," Journal of Software: Evolution and Process, vol. 28, no. 7, pp. 565-588, 2016.

[6] S. Dieckmann, and U. Hölzle, "A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks," Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 1628, Berlin, Germany: Springer, pp. 92-115, 1999.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and Realities: The Performance Impact of Garbage Collection," Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance), New York, NY, USA, pp. 25-36, 2004.

[8] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-First Garbage Collection," Proceedings of the 4th International Symposium on Memory Management (ISMM), Vancouver, BC, Canada, pp. 37-48, 2004.

[9] P. Runeson, and M. Höst, "Guidelines for Conducting and Reporting Case Study Research in Software Engineering," Empirical Software Engineering, vol. 14, no. 2, pp. 131-164, 2009.

[10] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Montreal, QC, Canada, pp. 57-76, 2007.

[11] T. Kalibera, and R. E. Jones, "Rigorous Benchmarking in Reasonable Time," Proceedings of the 2013 International Symposium on Memory Management (ISMM), Seattle, WA, USA, pp. 63-74, 2013.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing Wrong Data Without Doing Anything Obviously Wrong!," Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Washington, DC, USA, pp. 265-276, 2009.

[13] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in Software Engineering, Berlin, Germany: Springer, 2012.

[14] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed., Boston, MA, USA: Addison-Wesley, 2018.

[15] L. Cruz, and R. Abreu, "Catalog of Energy Patterns for Mobile Applications," Empirical Software Engineering, vol. 24, no. 4, pp. 2209-2235, 2019.

[16] A. Noureddine, "PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools," Proceedings of the 18th International Conference on Intelligent Environments (IE), Biarritz, France, 2022.