Optimised Fitness Functions for Automated Improvement of Software's Execution Time

Dimitrios Stamatios Bouras[1][0009−0001−3171−5194], Carol Hanna[2][0009−0009−7386−1622], and Justyna Petke[2][0000−0002−7833−6044]

[1] Peking University, Beijing, China
dimitris.bouras@outlook.com
[2] University College London, London, UK
{carol.hanna.21,j.petke}@ucl.ac.uk

Abstract. Precise measurement of software execution time is challenging due to environmental variability and measurement overheads, an issue critical for search-based software improvement systems that evaluate thousands of variants. While precise measurements offer precise fitness measures, they often introduce a significant time overhead. To understand which measures are most effective as fitness functions in search-based software optimisation, we conducted an empirical study of 21 approximates of execution time. These included hardware-level counters from `perf`, RAPL `energy`, and a custom measure based on weighted instruction cycles. To improve reliability, we evaluated each fitness function up to five times, using medians to reduce noise. We integrated the 13 most promising measures into a search-based software optimisation framework called `MAGPIE`. We evaluated these fitness functions plus `Time`, already present in `MAGPIE`, on 7 benchmarks using both code-level and parameter-level mutations. To assess generalizability, we tested the best performing measures with the parameter tuning tool `ParamILS` and analyzed how tool and search strategy affect outcomes. Our results show that `perf`'s `cycles` measure yields the best overall performance, outperforming `Time` by 5.1%. Sampling three times balances reliability and exploration. Energy and the weight-based measure excel in specific scenarios, with `weights` being the best for parameter optimization on `MAGPIE`, but are better suited to longer searches due to their overhead. We highlight a trade-off: low-overhead measures like `Time` work well for short runs, while robust measures such as `cycles` and `weights` benefit longer ones.

Keywords: Software Performance · Search-Based Software Engineering · Genetic Improvement

1 Introduction

Optimisation of non-functional properties of software, such as runtime or its energy use, have been pursued through a variety of strategies at the code level [7]: static methods compare variants to the original without re-evaluation (e.g., [1]);

sampling methods test selected variants via random or systematic sampling (e.g., [13]); exploratory methods iteratively refine variants using Bayesian tuning (e.g., [2]) or fuzzing (e.g., [4]); evolutionary approaches, notably genetic improvement [30], that mutates code, have achieved success in domains like bioinformatics (e.g., [23]), while manual methods remain in use when automation is impractical (e.g., [18]). More recently, large language models have been utilised to find efficiency improvements in code [32]. To be useful, they require a large set of training data with reliable measures of the non-functional property of interest.

A recurring challenge in optimisation of naturally noisy measures is selection of reliable fitness functions. Here, we consider the most common non-functional optimisation objective — program's running time [7]. Execution time as a fitness is common but noisy [10,25]; stable proxies like instruction count or executed lines (e.g., [23]) are more repeatable but may not reflect real runtime impact. Search-based methods employed in sampling (e.g., via random search), exploratory (e.g., genetic improvement), and evolutionary approaches (e.g., genetic algorithms) for runtime improvements [7], require evaluation of even thousands of software variants; even more would have to be evaluated to provide a training set for currently popular LLMs. This motivates exploring measures balancing cost to evaluate software variants, and their correlation with the objective of interest, as well as the ability to guide a given search strategy towards improved code. With this work we aim to close this gap in the literature.

First, we evaluated 21 runtime-approximating measures, from `perf` hardware counters to Intel RAPL `energy` readings and a custom instruction-weighted measure, to assess their stability and correlation with execution time. Although the selected measures are generally widely used, we have not found work that compares them all with respect to proxy for execution time.

Next, we integrated the top-performing measures into `MAGPIE` [6], a search-based improvement framework, and tested on seven benchmarks using genetic improvement and parameter tuning. Moreover, we examined multi-run retry strategies to reduce measurement noise and their trade-offs with search breadth. As far as we know, we are the first to compare these measures with the aim of finding the best trade-off between measurement accuracy and its ability to guide search towards improving software variants. Overall, our experiments cover 70 configurations (14 measures $\times$ 5 retry levels) across 22 scenarios (Section 3), totalling 1540 software improvement runs. Results show that the `cycles`, `Time`, `branches`, and `cache_references` measures provide strong search guidance, while retry of 3 balances robustness and efficiency. The `energy` and `weights` measures excel for long-running, repeatedly measured tasks despite higher cost.

Furthermore, we integrated the best measures into a parameter tuning tool called `ParamILS` to explore our findings in a different context. We observed that lightweight fitness measures dominate under short, multi-instance searches, while robust measures remain competitive in cost-tolerant contexts.

To sum up, our contributions are:

1. A comparative study of consistency and runtime correlation of 21 measures.
2. Integration of all measures as fitness functions into the search-based improvement framework `MAGPIE`.

3. Analysis of 70 measure–retry setups for optimal cost–effectiveness trade-offs within genetic improvement and parameter tuning settings with both local search and genetic programming searches in `MAGPIE`.
4. A case study evaluating our best performing measures within a different search-based context, a parameter tuning framework `ParamILS`.

2 Research Questions

In order to assess the trade-offs between accuracy of candidate fitness functions as proxies for runtime vs. effectiveness in leading search towards improved variants in search-based software optimisation frameworks, we pose the following research questions:

RQ1: *Which measures are most consistent and which most closely correlate with execution time, potentially serving as its surrogates?*
Knowing which execution time surrogates are most accurrate and at the same time most consistent could save valuable resource time wasted on repetitions. Answer to this question goes beyond applicability in the search-based context.

RQ2: *Which performance measures are most effective as fitness functions, in terms of guiding search towards finding the most time-saving software variants within a constrained time budget?*
Within the search-based context we want to know which measures are lightweight, yet accurate enough, to guide search towards improved software variants.

RQ3: *How many repeated measurements (retries) provide the best trade-off between measurement precision and computational overhead?*
Here we are interested in establishing the needed measurement overhead, per measure. Some proxies for time might require less repetitions to provide enough accuracy to guide search towards improved software variants.

RQ4: *Which performance measures and retry parameters lead to the most consistent search, ensuring stable and progressive optimization?*
Answer to this research question will provide recommendations for best use of each measure of interest.

3 Methodology

Performance Measure Collection We evaluated 21 execution time surrogates (Table 1). We selected these primarily from `perf` hardware counters (e.g., cycles, cache, branches, instructions) due to their established correlation with execution time: hardware performance counters and energy have long been used as reliable surrogates for runtime prediction across architectures and workloads [11,31]. Energy usage was recorded using Intel's RAPL interface. We also used the `weights` metric [8], which applies throughput-based weights [16] to assembly instructions. This involves first profiling with `perf-record`, computing per-function runtime shares via `perf-report`, then identifying per-command shares within functions using `perf-annotate` and lastly multiplying command throughput by its function share and command share, and aggregating. This yields a weighted sum approximating runtime more accurately than raw instruction counts.

Measure Consistency and Correlation with Execution Time To answer RQ1, we assessed which measures are both consistent and proportional to execution time. We repeated runs on each benchmark 21 times (based on the calculations in the online Appendix D). For each measure–benchmark pair we computed coefficient of variation $CV = (\sigma/\mu) \times 100\%$, then took the median CV across benchmarks. Lower CVs indicate higher reliability, with measures beating `Time` considered stronger guides. Rather than Pearson's r (insensitive to systematic scaling), we examined proportionality: for each measure(M)–benchmark(b) pair we computed the median ratio $\text{Factor}_{M,b} = \frac{\text{Median}(M,b)}{\text{Median}(T_b)}$, and the CV of these factors across benchmarks. Low Factor CV indicates consistent scaling with execution time.

Search-Based Optimisation Frameworks We selected two frameworks for our evaluation: `MAGPIE` (to answer RQ2–RQ4) and `ParamILS` to show generalisation potential and explore shorter budgets.

`MAGPIE` [6] is a Python-based, language-agnostic framework for search-based software improvement, representing edits as XML sequences to decouple search from specific modification techniques. This way searches across both parameters and modification directly to code are possible. `MAGPIE` supports multiple local search (LS) and genetic programming (GP) search strategies. We adopt the best variants from [5]: First-Improvement local search and GP with uniform concatenation crossover, both at statement and parameter level — i.e., genetic improvement and parameter tuning settings. For GP we used a population of 10, running as many generations as the time budget allowed for.

We integrated 18 `perf` measures as fitness functions, Intel RAPL `energy`, and the weighted-instruction `weights` measure [8]; standard Unix `Time` was already present. We introduced a *retries* parameter into `MAGPIE`, re-executing each variant 1 to 5 times and taking the median to suppress noise. While this increases runtime, stability markedly improves, but early experiments showed no benefit beyond 5 retries. Recall, we need to balance the measurement accuracy against time for the search process. In that sense repetitions of 20-30 tries commonly recommended in the literature *outside of search-based context*, and confirmed in our experiments on measure consistency (Appendix D), are simply impractical.

To test measure effectiveness in a different search context, we used `ParamILS` (v2.3.8) [20]— a mature, model-free parameter optimizer with features like adaptive capping and anytime behavior. Its proven performance across diverse domains and differing search dynamics from `MAGPIE` make it well-suited for evaluating our measures and retry strategies in alternative search contexts, particularly shorter and more localized searches. We modified `ParamILS`'s evaluation script to capture `perf`, `energy`, and `weights` outputs. Experiments focused on parameter tuning over the same benchmarks as `MAGPIE`, using eight measures: the six best from experiments with `MAGPIE` plus `weights` and `energy` (see answer to RQ2 in Section 4.2 for justification).

Benchmarks We used benchmarks common in related research [7], selected for popularity, maintainability, and computational intensity.

For **RQ1** we used 8 popular controlled micro-benchmarks from standard suites: `bzip2`, `gzip` from SPEC [33]; `fftp`, `lu` from SPLASH-2 [34]; `blackscholes` from PARSEC [28]; and `bitcount`, `sha`, `jpeg` from MiBench [27].

For **RQ2–RQ4** and the `ParamILS` case study 7 larger, optimization-rich benchmarks were employed: `MiniSAT`, `MiniSAT_hack`, `SAT4J`, `Weka`, `LPG` (these were used in the original `MAGPIE` paper and we thus know that optimizations can be found), plus 2 parameter-heavy real-world tasks (`scipy.optimize.minimize`, `zlib.compress`). All were parameter-optimized in both `MAGPIE` and `ParamILS`; for `MAGPIE`, code optimization was also applied to `MiniSAT`, `MiniSAT_hack`, `SAT4J`, and `Weka`. `LPG` was limited to parameter tuning (legacy library issues), while `scipy` and `zlib` lack meaningful code-level variants. We used both the best local search (LS) and genetic programming (GP) variants with `MAGPIE`, totalling 22 scenarios (4 x 2 code optimisation tasks plus 7 x 2 parameter tuning tasks).

Thus, RQ1 used small, stable workloads for consistency/correlation, while RQ2–RQ4 employed benchmarks with substantial optimization potential, spanning both parameter and code optimization contexts.

Experimental Setup We ran our experiments on a Rocky Linux 9.4 Desktop with an 8-core Intel(R) Xeon(R) E5-1620 CPU with 16 GBs RAM. For `MAGPIE`, we recorded the number of variants examined under each measure–retry combination. We selected a 30-minute fixed time budget in order to balance practical resource constraints with established practices in the field. This duration aligns with prior studies in search-based software engineering, where similar time budgets have been effectively employed [3],[14]. Each configuration of `ParamILS` was given 5 minutes per benchmark; for each benchmark–measure pair. Since `ParamILS` specializes in parameter tuning, exploratory trials showed that runtime can already find near-optimal solutions within this budget. Allowing longer runs would mainly advantage high-overhead measures, while runtime (the baseline) already converges quickly, so 5 minutes provides a fair comparison of surrogates against execution time. Three optimization runs were executed (as 3 samples were found sufficient to minimize randomness as a factor), and the median of the best configuration of each of the three runs determined ranking.

4 Results and Discussion

Next, we present our experimental results and answer our research questions. Data in each table that answers RQs 2–4 reflects results from one of two setups: `MAGPIE` with local search (LS) or genetic programming (GP). To recap, each setup has been run across 7 benchmarks, 4 run in genetic improvemenet mode, making code-level mutations, and all 7 run in parameter tuning mode, thus data from 11 scenarios is aggregated per LS or GP. The number of runtime surrogates and retries considered for each sceanrio are outlined in each subsection.

4.1 RQ1: Measure Consistency & Correlation with Execution Time

First, we present the results of running our initial experiments on measure consistency and correlation to execution time in Table 1.

Table 1: **RQ1** Variability and correlation to execution time for 8 standard benchmarks. MCV: Median Coefficient of Variation across benchmarks; MCVR: Median Ranking for CV across benchmarks; FCV: CV of Factor across benchmarks. Lower is better for all measures.

Measure	**MCV**	**MCVR**	**FCV**	**Measure**	**MCV**	**MCVR**	**FCV**
major-faults	0.000	1.0	108.49	perf_time	0.852	13.5	0.58
minor-faults	0.046	5.0	231.22	dTLB-loads	0.840	12.0	146.79
total-faults	0.040	4.5	231.21	weights	0.523	11.5	24.82
branches	0.004	3.0	90.88	cache-misses	1.972	17.0	113.60
L1-dcache-loads	0.096	6.0	53.82	L1-dcache-load	1.170	15.5	100.05
instructions	0.001	2.0	43.09	-misses			
cycles	0.221	8.0	0.81	cache-references	0.881	13.0	88.76
branch-misses	0.150	7.0	112.40	Time	0.814	16.5	0.00
energy	0.452	10.0	14.96	dTLB-load-misses	3.920	17.5	243.43
task-clock	0.593	13.5	0.30	cs	30.201	20.0	53.23
cpu-clock	0.754	15.0	0.63	migrations	142.980	21.0	106.42

Among traditional runtime proxies, `branches, cycles, branch_misses, L1_dcache_loads`, and `instructions` showed very low CV (0.0013–0.22%), confirming their stability. Fault-related measures such as `total-faults`, `minor_faults`, and `major_faults` also had extremely low CV (<0.05%), likely due to their rare occurrence. In contrast, raw `Time` and `perf_time` were less reliable (CV ∼0.8–0.85%), sensitive to system load. Clock-based metrics such as `cpu_clock, task_clock` performed slightly better (<0.75%), while `context_switches` and `mitigations` were unusable. `Energy` (0.45% CV) and the custom `weights` metric (0.52%) occupied a mid-range. The `weights` CV could be reduced by higher `perf` sampling rates, but experimentation showed little benefit for benchmark runs over ∼1s, while overhead increased sharply. A sampling rate at ∼2000 Hz is a practical trade-off. Correlation analysis (Table 1) showed that time-based measures (`perf_time, cpu_clock, task_clock`) best tracked execution time (FCV <0.63%). `Cycles` performed almost as well (0.81%) but with much lower CV, making it a robust surrogate. Surprisingly, `energy` (14.96%) and `weights` (24.8%) correlated more strongly with execution time than instructions or branches, despite moderate CV, highlighting their promise. This is particularly valuable when direct time measurement is impractical or unreliable, such as in parallel or distributed environments. Other low-CV measures (`branch_misses, branches, L1_dcache_loads, instructions`) showed weaker correlates (FCV 43–112%).

Answer to RQ1: `cycles` is the most reliable surrogate for execution time. `energy` and `weights` also show potential, thus we recommend these especially for contexts where direct time measurement is infeasible.

4.2 RQ2: Execution-time Surrogates as Fitness Functions

To evaluate performance measures for software optimization, we evaluated the best 14 as fitness functions within the `MAGPIE` framework. Original is the program

Table 2: **RQ2** Mean Median Execution Times (MET), Median Ranks (MR), and Number of Variants (Nov) for Each Measure. Columns 2–4: experiments with Local Search. Columns 5–7: experiments with Genetic Programming. Lower is better for MET and Rank, for NoV higher is better.

Measure	**Mean MET(s)**	**Median Rank**	**NoV**	**Mean MET(s)**	**Median Rank**	**NoV**
	local search			genetic programming		
Original	10.28	12	-	9.65	13	-
perf-measures						
L1_dcache_loads	5.42	7	240	5.99	7	397
branch_misses	5.46	8	222	5.75	5	431
branches	5.17	6	252	6.22	7	393
cache_misses	5.63	4	255	7.02	8	419
cache_references	5.40	8	234	4.64	5	451
cpu_clock	10.28	14	157	8.85	14	376
cycles	5.02	4	252	6.15	4	426
faults	8.92	11	176	7.28	12	381
instructions	5.59	7	270	6.48	7	384
task_clock	10.45	13	164	9.87	14	387
perf_time	5.43	6	257	5.66	7	409
other						
energy	6.53	6	156	6.21	11	257
Time	5.29	4	260	5.17	4	414
weights	7.02	10	106	6.96	12	275

before optimization. We first discuss results obtained with first-improvement local search (LS), followed by genetic programming (GP).

The 7 measures omitted from Table **??** were excluded from later experiments due to excessive variability (MCV>2) or lack of correlation with execution time(FCV>120). With faults we refer to *major_faults* from now on.

For each benchmark, we used one of the 14 measures as the fitness function to optimize the original program and obtain an improved variant with lower execution time. We then ranked each measure by the execution time of its best variant per benchmark and summarized performance by median execution time (MET), average rank across benchmarks, and the mean number of explored variants (Table **??**).

Regarding the LS approach, `cycles` emerged as the most effective fitness function (5.02s), closely followed by `branches` (5.17s) and `Time` (5.29s).
Other strong performers included `perf_time`, `cache_references`, and `L1_dcache_loads`, confirming `cycles`, `branches`, and cache-related metrics as robust alternatives to raw execution time. `Energy` ranked well (median rank 6) despite a middling mean MET (6.53s), excelling in three benchmarks.

The custom `weights` measure, though hindered by high overhead ($\approx$ 40% of variants explored vs. `perf`), achieved leading results in `LPG` (18% faster than the runner-up) and `SAT4J`, especially where execution traces were simple and dominated by few options (it was the best for parameter optimization). `Energy` also incurred overhead ($\approx$ 60% of variants explored) but still showed competi-

Measure	Mean MET(s)	Median Rank	NoV
Original	9.65	13	-
perf-measures			
L1_dcache_loads	5.99	7	397
branch_misses	5.75	5	431
branches	6.22	7	393
cache_misses	7.02	8	419
cache_references	4.64	5	451
cpu_clock	8.85	14	376
cycles	6.15	4	426
faults	7.28	12	381
instructions	6.48	7	384
task_clock	9.87	14	387
perf_time	5.66	7	409
other			
energy	6.21	11	257
Time	5.17	4	414
weights	6.96	12	275

Table 3: Mean Median Execution Times (MET), Median Ranks, and Number of Variants for Each Measure under Genetic Programming.

tive effectiveness. Most `perf` measures achieved comparable search depth since they rely on `perf-stat`. By contrast, measures like `faults`, `cpu_clock`, and `task_clock` underperformed not due to measurement overhead but because they poorly reflect runtime improvements, failing to accelerate patch discovery.

Regarding the GP approach, `cycles` consistently ranked as a top performer, excelling in three cases despite lower performance in two scenarios, which affected its mean MET. The measure of `cache_references` notably achieved the best mean MET (4.635 s), underscoring the significant impact of cache operations on execution time. `L1_dcache_loads`, `branch_misses`, `branches`, `instructions`, `Time`, and `perf_time` performed well as they did for local search, with mean METs ranging from 5.17 to 6.48. Intel's RAPL energy consumption measure was a viable surrogate with a mean MET of 6.2s, indicating its potential for representing execution time. Conversely, `faults`, `task_clock_time`, and `cpu_clock_time` were again the least effective measures. The custom `weights` measure, also underperformed (6.96s) hindered by the number of variants, but demonstrated high consistency and can be very effective given sufficient exploration time.

Answer to RQ2: `cycles`, `Time`, `branches`, `cache_references`, `perf-time`, and `L1_dcache_loads` achieved the lowest MET (5.02–5.43s vs. baseline 10.28s in LS setup). `Energy` ranked above average (median rank 6), while `weights` was the best measure in two benchmarks despite high overhead, but underperformed when used with GP.

4.3 RQ3: Optimal Number of Retries for Measures

We ranked retry counts for each benchmark–measure pair median execution time of the optimized variant, and aggregated these rankings across benchmarks (Ta-

Table 4: **RQ3** Recommended Retry Numbers for Each Measure from LS experiments. MaRN: Mean Retry Number; MRN: Median Retry Number.

Measure	MaRN	MRN	IQR	STD
energy	3.273	4.000	1.000	1.009
L1_dcache_loads	2.636	2.000	2.500	1.433
branch_misses	2.273	2.000	2.000	1.348
branches	2.636	2.000	2.000	1.502
cache_misses	2.273	2.000	2.500	1.272
cache_references	2.545	2.000	2.500	1.572
cpu_clock	2.636	3.000	2.500	1.567
cycles	2.636	2.000	3.000	1.629
faults	2.909	2.000	3.000	1.700
instructions	3.273	3.000	2.500	1.348
task_clock	3.636	4.000	1.500	1.121
perf_time	2.636	1.000	4.000	1.963
Time	2.273	2.000	2.500	1.272
weights	3.091	3.000	1.000	0.944

Table 5: **RQ3** Percentage Decrease in Variants examined by Number of Retries in LS experiments. RN: Retry No.; MaDP: Mean Decrease %; MDP: Median Decrease %.

RN	MaDP	MDP	STD	IQR
1	0.0	0.0	0.0	0.0
2	30.71	30.65	19.25	28.47
3	47.54	48.92	20.63	29.11
4	59.37	60.96	18.58	23.52
5	67.49	68.87	16.24	20.73

Table 6: **RQ3** Statistical Analysis of Retry Rankings Across All Benchmarks from LS experiments. MaR: Mean Rank; MR: Median Rank.

Retries	MR	MaR	STD	IQR
1	3.143	3.078	0.389	0.429
2	3.000	2.974	0.288	0.250
3	2.929	2.974	0.298	0.464
4	2.989	3.005	0.325	0.357
5	3.071	2.992	0.389	0.500

Table 7: **RQ3** Statistical Analysis of Retry Rankings Across All Benchmarks from GP experiments. MaR: Mean Rank; MR: Median Rank.

Retries	MR	MaR	STD	IQR
1	2.36	2.47	0.27	0.39
2	2.93	2.95	0.40	0.50
3	3.00	3.07	0.46	0.71
4	3.29	3.18	0.41	0.39
5	3.36	3.33	0.41	0.32

ble 6 for Local Search). Mean (MaR), median (MR), with standard deviation (STD), and interquartile range (IQR) of ranks show that retry = 3 is optimal: it yields the most consistent improvements while exploring enough variants. Beyond 3, retries reduce the number of variants without notable consistency gains, Conversely, 1 or 2 retries explore more variants but less efficiently.

Table 4 provides measure-specific recommendations. Measures with low MCV (e.g., `instructions`) require fewer retries, reflecting inherent stability. Measures with higher MCVs (e.g., `energy`, `weights`) benefit from larger retry counts to mitigate run to run variability. For all results in Tables 4 to 7, lower is better.

Retries directly reduce the number of variants explored: compared to 1, retry = 3 reduces steps by $\approx 48\%$ (Table 5). Despite this cost, retries markedly increase reliability of optimization trajectories, enabling clearer convergence toward improved variants. Thus, retry = 3 offered the best balance between search breadth and stability, though the optimum varies by measure

For GP, retries ranked from 1 (best) to 5 (worst), showing no benefit from higher values (Table 7). Unlike local search, where consistency is vital to not

Table 8: RQ4: Median and Mean MAD, STD, IQR (lower is better), and Search Performance (TD, ADPS, POD) (higher is better) for Each Measure and Retry Number. TD: Total decreases; ADPS: Percentage of decrease that are achieved per step on average; POD: Percentage of Steps that are overall decreases.

Measure	Med. MAD	Mean MAD	STD	IQR	TD	ADPS	POD
energy	0.157	0.155	0.267	0.266	6.0	0.047	0.098
L1_dcache_loads	0.174	0.166	0.198	0.200	7.0	0.040	0.074
branch_misses	0.145	0.143	0.301	0.324	9.0	0.041	0.079
branches	0.188	0.198	0.307	0.329	8.0	0.040	0.081
cache_misses	0.165	0.167	0.270	0.261	8.0	0.043	0.068
cache_refs	0.163	0.166	0.220	0.219	8.0	0.048	0.077
cpu_clock	0.390	0.329	0.134	0.124	4.0	0.171	0.053
cycles	0.155	0.147	0.288	0.271	10.0	0.043	0.085
faults	0.030	0.031	0.055	0.042	9.0	0.011	0.090
instructions	0.199	0.211	0.306	0.318	9.0	0.038	0.093
task_clock	0.415	0.314	0.175	0.155	4.0	0.200	0.051
perf_time	0.194	0.203	0.320	0.314	8.0	0.045	0.077
Time	0.164	0.166	0.311	0.341	9.0	0.047	0.087
weights	0.187	0.174	0.297	0.276	5.0	0.089	0.077

Retries	Med. MAD	Mean MAD	STD	IQR	TD	ADPS	POD
1	0.245	0.179	0.278	0.291	9.0	0.047	0.040
2	0.214	0.186	0.265	0.266	7.5	0.046	0.062
3	0.177	0.167	0.235	0.265	8.0	0.047	0.079
4	0.180	0.170	0.239	0.258	7.0	0.050	0.095
5	0.157	0.156	0.215	0.233	7.0	0.051	0.110

divert the search, GP's population diversity offsets noise, making a single retry most effective, since it increases the number of variants.

Regarding recommendations, we observed no significant difference in results between GP and LS experiments, thus to save space, we report on the LS experiments only, and provide results for GP in the available online Appendix B.

Answer to RQ3: Retry = 3 balances exploration and reliability, achieving the best overall ranks despite halving the number of explored variants. The precise optimum varies with the stability of the measure (as indicated by MCV). For GP retry of 1 is often surprisingly sufficient.

4.4 RQ4: Consistency Analysis Across Measures and Retries

When looking at the data to answer RQ4, we observed no significant difference in results between experiments with LS and GP. Given that local search variants proved very effective both in GI [5] and parameter tuning work [20], we report on results from experiments with LS only. We provide result tables obtained from GP runs in the online Appendix B.

Consistency through MAD. We assessed consistency via Mean Absolute Deviation (MAD) from a best-fit line across retry–measure combinations, a standard robustness metric [35,21]. Table 8 shows median and mean MAD grouped by measure and by retry count. The measure `faults` (3% MAD) was by far

the most stable, while `branch_misses`, `cycles`, `cache_misses`, and `Time` also ranked highly consistent. Even `energy` and `weights` maintained relatively low variance (<19%). Higher retry values systematically reduced MAD, confirming that retries dampen stochastic noise.

Consistency through decreases. We further analyzed the search trajectory by: (i) total decreases (TD) in fitness, (ii) average decrease percentage per step (ADPS), and (iii) proportion of decreases (POD). While retries sharply cut the number of evaluated variants (Table 5), the number of decreases fell only mildly, so POD actually rose. ADPS also improved slightly with retries. Thus, retries trade off exploration depth for efficiency: fewer steps, but each more likely and impactful. A retry value of 3 offered the best balance, aligning with Table 6.

Measure-specific effects. Across measures (irrespective of retries), `energy`, `faults`, `instructions`, `Time`, and `cycles` achieved high POD (> 8.5%), reinforcing their role as reliable guides. Notably, `cycles` combined middle-range ADPS (4.3%) with the highest TD (10), showing how steady, incremental progress can accumulate. In contrast, `weights` produced the largest ADPS (8.9%), but low TD (5) due to fewer opportunities, illustrating the cost of sparse sampling despite steep improvements. For `Time`, high variance meant some "decreases" were likely noise, particularly with low retries.

Example search space comparisons. Figure 1 in the onlineAppendix A compares `task_clock` (retry=1) vs. `weights` (retry=3) on `LPG`. The former yields chaotic, untargeted exploration with near-baseline variants and only one late cluster of modest improvements. In contrast, `weights` exhibits better guidance with clear "staircase" of optimization phases producing distinct improved groups.

Answer to RQ4: Larger retries consistently stabilize search (lower MAD), reduce steps but increase per-step efficiency (higher POD and ADPS). Stable measures like `branch_misses`, `cycles`, `cache_misses`, and especially `faults` guided the search reliably. `weights` demonstrated the steepest per-step improvements, though constrained by limited sampling opportunities.

4.5 Cross-Tool Validation with Short Search Budgets

Table 9 reports the results of our `ParamILS` experiments, where each run was limited to 5 minutes. To ensure stability under this shorter budget, each benchmark configuration was executed 21 times per run, with medians averaged across three searches. We report both the average Median Execution Time (MET) and the average rank of each measure across benchmarks.

`ParamILS` uses adaptive instance-level evaluation, sampling, and racing strategies that preclude meaningful tracking of intermediate variants. Consequently, we focused on the final best configuration as the primary performance indicator. Comparing `MAGPIE`'s interpretable variant counts against `ParamILS`'s end results, offers insights into the trade-offs between search depth, consistency, and efficiency across different search-based optimization paradigms.

Results show a clear trade-off between overhead and guidance. The lightweight metric `time` consistently ranked best, confirming that minimal-overhead mea-

sures are most effective when thousands of fast evaluations are required. In contrast, `weights`, which incurs sampling overhead, performed worst. We note that the sampling rate for `weights` was selected after experimentation, as higher rates offered limited benefit for benchmarks exceeding one second per run, while adding substantial cost. Intermediate hardware measures (`L1_dcache_loads`, `branches`, `cycles`) performed competitively with only modest overhead, while `perf_time` showed high variability across benchmarks. The `energy` measure also underperformed, ranking second worst.

These findings suggest that short-budget optimization favors measures that minimize overhead, while more detailed or resource-intensive measures become valuable primarily in longer searches or when exploring fewer variants.

Case Study Insight: Under limited search time, lightweight metrics such as `time` clearly dominate by balancing low overhead with sufficient guidance. Hardware-based counters remain competitive but add cost, while complex or high-overhead measures provide little advantage in this setting.

Table 9: ParamILS Experimental Results Across Measures.

Metric	Average MET (s)	Average Rank
`time`	3.894	3.000
`L1_dcache_loads`	3.976	3.714
`branches`	3.983	3.857
`cycles`	4.148	4.714
`cache_references`	4.215	4.857
`perf_time`	4.649	3.714
`energy`	4.787	6.000
`weights`	6.200	6.143

5 Threats to Validity

The generalizability of our findings may be limited due to the use of 22 benchmark-search type combinations in this study. While the benchmarks cover a variety of tasks and the search type includes both parameter tuning and source code optimization to broaden applicability, they may not fully represent all potential use cases. In our experiments we selected real-world benchmarks commonly used in the literature to ensure relevance. Our study was conducted on a single computer system due to the need for uniform platform for all measures. This means that certain measures might be influenced by the underlying architecture. However, we are interested in finding measures that could serve as *fitness functions*, thus consistently differentiating between two software variants. Therefore, we believe that our results will generalise, although future studies could enhance the robustness of our findings by replicating the experiments on other hardware configurations. We employed a consistent seed across all runs to ensure a fair

starting point and designated a substantial time budget (30 minutes) to explore hundreds of variants, randomness may still influence outcomes. However, considering the extensive number of combinations (1540 in total) tested in this study, we expect that randomness would average out, mitigating its impact on the overall results. To further address generalisability concerns, we have established an infrastructure that allows for easy inclusion of additional benchmarks, which future researchers can leverage to expand our work. All code, documentation, and results are available at: `https://anonymous.4open.science/r/Optimised-fitness-functions-ssbse/README.md`.

6 Related Work

Bloat and Petke in their survey of benchmarks for automated improvement of software's non-functional properties [7] conclude that the most targeted non-functional property is execution time, with energy usage and software size being the 2nd and 3rd most improved property in the literature. Various strategies have been adopted to improve code performance at the software level. The most common are compiler optimizations, algorithm configurations, and source code modifications. Among these techniques, the application of metaheuristics for software improvement have emerged as particularly potent methods due to their ability to explore complex search spaces efficiently [7]. Most search-based empirical studies that focus on runtime improvements at the code level use the Unix `Time` command or the `perf-time` commands as the main part of the fitness function, e.g., [20] and [9]. Liou et al. employed GPU kernel runtime [24]. Langdon et al. based their fitness function on the number of task units performed in a specific time window [22], while Garciarena and Santana, [17] used both compilation and execution time. Other works, such as the Cole framework [19], try to achieve multi-objective optimization, which include the execution time as one of their primary objectives, alongside other criteria of memory usage, power consumption, and code size. However, the reliance of all those tools on accurate and consistent time measurements is fundamentally flawed since research has shown the limitations of simple runtime measurements due to their susceptibility to external variations such as background CPU processes, varying CPU loads, and multi-core contention. Carothers and Fujimoto [10] and Meyer et al.[26] have shown that execution time is a very volatile measure that is affected by background processes and CPU load, Mazous et al.[25] studied how time is affected by multi-core or parallel executions. These studies underscore the necessity for incorporating a broader set of performance indicators that can more accurately reflect the true efficiency of software under various conditions. Another potential measure is the weighted sum of the assembly commands in the binary program, based on the commands' CPU cycle throughput. Patterson's and Hennessy's book on computer organization and design [29], proves that execution time and CPU cycles for each command are deeply connected. However, the use of pipelining in modern CPUs makes the two measures no longer analogous. Despite these challenges, the work by Bouras et al. [8], along with additional research on the

correlation between cycles-per-instruction and total execution cycles with runtime [15], supports the reasonable assumption of such a correlation.

7 Conclusion

Precision of execution time measurement strongly affects the efficiency of search-based software optimisation. Our study showed that multiple hardware-level measures, notably `cycles`, `branches`, and RAPL's `energy`, can serve as reliable surrogates for execution time, often providing more consistency under noisy conditions. Recomputing fitness values three times and taking the median further improved stability, albeit at the cost of exploring fewer variants. These findings highlight practical trade-offs between precision, overhead, and search efficiency, and point to broader applications where robust performance proxies are valuable. The insights gained from exploring new performance measures as fitness functions underscore the potential for broader applications, inspiring further research to refine these techniques and extend their use in more diverse computational environments. It would be interesting to explore the potential of our weighted measure in the context of recent advances such as Google's AlphaDev [12]. The `weight` measure could provide a more precise measure and thus enhance the robustness of such systems. Furthermore, such measures could be used to efficiently evaluate large corpora of data to provide training bed for LLMs.

References

1. Abdulsalam, S., Lakomski, D., Gu, Q., Jin, T., Zong, Z.: Program energy efficiency: The impact of language, compiler and implementation choices. In: International Green Computing Conference. pp. 1–6 (2014)
2. Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J., Silvano, C.: Cobayn: Compiler autotuning framework using bayesian networks. ACM Trans. Archit. Code Optim. **13**(2) (2016)
3. Ayerdi, J., Terragni, V., Jahangirova, G., Arrieta, A., Tonella, P.: Genmorph: Automatically generating metamorphic relations via genetic programming. IEEE Transactions on Software Engineering **50**(7), 1888–1900 (2024)
4. Biswas, P., Burow, N., Payer, M.: Code specialization through dynamic feature observation. In: Proceedings of CODASPY. p. 257–268. CODASPY, Association for Computing Machinery, New York, NY, USA (2021)
5. Blot, A., Petke, J.: Empirical comparison of search heuristics for genetic improvement of software. IEEE Trans. Evol. Comput. **25**(5), 1001–1011 (2021)
6. Blot, A., Petke, J.: Magpie: Machine automated general performance improvement via evolution of software (2022)
7. Blot, A., Petke, J.: A comprehensive survey of benchmarks for improvement of software's non-functional properties. ACM Comput. Surv. **57**(7) (Feb 2025)
8. Bouras, D.S., Lamprakos, C.P., Catthoor, F., Soudris, D.: Reliable basic block energy accounting. In: Embedded Computer Systems: Architectures, Modeling, and Simulation: 23rd International Conference, SAMOS 2023, Samos, Greece, July 2–6, 2023, Proceedings. p. 193–208. Springer-Verlag, Berlin, Heidelberg (2023)
9. Brownlee, A.E.I., Petke, J., Alexander, B., Barr, E.T., Wagner, M., White, D.R.: Gin: genetic improvement research made easy. In: Proceedings of the Genetic and

Evolutionary Computation Conference. p. 985–993. GECCO '19, Association for Computing Machinery, New York, NY, USA (2019)
10. Carothers, C., Fujimoto, R.: Background execution of time warp programs. Proceedings of Symposium on Parallel and Distributed Tools pp. 12–19 (1996)
11. Conte, T.M., Menezes, K.N., Mills, P.M., Patel, B.A.: Optimization of instruction fetch mechanisms for high issue rates. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA). pp. 333–344 (1996)
12. Daniel J. Mankowitz, A.M.e.a.: Faster sorting algorithms discovered using deep reinforcement learning. Nature **618**(7964), 257–263 (2023)
13. Daud, S., Ahmad, R.B., Murthy, N.S.: The effects of compiler optimisations on embedded system power consumption. Int. J. Inf. Com. Tech. **2**(1/2), 73–82 (2009)
14. Eriksson, C., Wiidh, F.: The impact on time efficiency of varying mutation rate and population size in jgenprog (2024)
15. Eyerman, S., Eeckhout, L.: Per-thread cycle accounting. IEEE Micro **30**(1), 71–80 (2010)
16. Fog, A.: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd, and via cpus. Tech. rep., Tech. Univ. of Denmark (2022)
17. Garciarena, U., Santana, R.: Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion. p. 1159–1166. GECCO '16 Companion, Association for Computing Machinery, New York, NY, USA (2016)
18. Hamza, H., Counsell, S.: Exploiting slicing and patterns for rtsj immortal memory optimization. In: Proc. of the 2013 Internat. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. p. 159–164. PPPJ '13, Association for Computing Machinery (2013)
19. Hoste, K., Eeckhout, L.: Cole: compiler optimization level exploration. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. p. 165–174. CGO '08, Association for Computing Machinery, New York, NY, USA (2008)
20. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stuetzle, T.: Paramils: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009)
21. King, A.P., Eckersley, R.J.: Chapter 2 - descriptive statistics ii: Bivariate and multivariate statistics. In: King, A.P., Eckersley, R.J. (eds.) Statistics for Biomedical Engineers and Scientists, pp. 23–56. Academic Press (2019)
22. Langdon, W.B.: Performance of genetic programming optimised bowtie2 on genome comparison and analytic testing (gcat) benchmarks. BioData Mining **8**(1), 1 (2015)
23. Langdon, W.B., Harman, M.: Optimizing existing software with genetic programming. IEEE Trans. Evol. Comput. **19**(1), 118–135 (2015)
24. Liou, J.Y., Forrest, S., Wu, C.J.: Genetic improvement of gpu code. In: GI Workshop. pp. 20–27 (2019)
25. Mazouz, A., Touati, S., Barthou, D.: Study of variations of native program execution times on multi-core architectures. 2010 International Conference on Complex, Intelligent and Software Intensive Systems pp. 919–924 (2010)
26. Meyer, T., Davis, J., Davidson, J.: Analysis of load average and its relationship to program run time on networks of workstations. J. Parallel Distributed Comput. **44**, 141–146 (1997)
27. MiBench: MiBench performance benchmark. `https://github.com/embecosm/mibench` (2024), accessed: 09-09-2024

28. PARSEC: PARSEC performance benchmark. https://github.com/bamos/parsec-benchmark (2024), accessed: 09-09-2024
29. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 5th edn. (2013)
30. Petke, J., Haraldsson, S., Harman, M., Langdon, W., White, D., Woodward, J.: Genetic improvement of software: A comprehensive survey. IEEE Transactions on Evolutionary Computation **PP**, 1–1 (2017)
31. Shao, Y., Brooks, D.: Energy characterization and instruction-level energy model of intel's xeon phi processor. In: Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED). pp. 389–394 (2013)
32. Shypula, A., Madaan, A., Zeng, Y., Alon, U., Gardner, J.R., Yang, Y., Hashemi, M., Neubig, G., Ranganathan, P., Bastani, O., Yazdanbakhsh, A.: Learning performance-improving code edits. In: The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net (2024)
33. SPEC: SPEC performance benchmark (2024), accessed: 09-09-2024
34. SPLASH-2: SPLASH-2 performance benchmark. https://www.capsl.udel.edu/splash/index.html (2024), accessed: 09-09-2024
35. Springer: Mean Absolute Deviation, pp. 336–337. Springer New York, New York, NY (2008)