Appendix

1 Data Collection

1.1 Configurable Software Systems

To ensure the practicality and generality of our empirical findings, this paper considers investigating three widely used configurable software systems with diverse engineering functionalities, including compiler, Web server, and database management system. In the following paragraphs, we outline their key characteristics including the engineering narration, configuration options considered in our experiments, and the settings of workloads.

- **LLVM**[1]: The LLVM Project is a collection of modular compiler and toolchain technologies. It provides a modern, SSA-based compilation strategy that supports both static and dynamic compilation of any programming language. ▶ LLVM has more than 578 configuration options while we choose 20 of them for our empirical study. ▶ 12 test suites from the widely used `PolyBench` benchmark suite[2] are chosen to constitute the workloads. ▶ The `run_time` for the compiled program is used as the fitness function to measure the quality of a configuration of the `LLVM`.
- **APACHE**[3]: The APACHE HTTP Server Project aims to provide a robust and scalable HTTP service. ▶ It consists of multiple modules, the core of which has 89 configuration options and 21 configuration options for MPM module. Here we choose 15 options directly related to the quality of a configuration in our experiments. ▶ 9 different running environments are generated by using the Apache HTTP server benchmarking tool[4]. ▶ We use the request handled per second as the fitness function.
- **SQLITE**[5]: This is an embedded database project. Instead of maintaining a separate server process, SQLITE directly reads and writes data to disk. ▶ It has 50 compile-time and 29 run-time configuration options and we chose 18 of them in this study. ▶ We used the SQLITE Benchmark[6] to constitute 10 different running workloads. ▶ The writing speed in sequential key order in async mode (`fillseqsync`) is used as the fitness function.

The meta information of the selected configuration options (parameters) for these systems are listed in Tables 1 to 3. The settings of different workloads for each system are listed in Table 4. In the following paragraphs, we introduce the corresponding attributes for different workloads.

- For the LLVM, we adopted different compiling file to constitute different workloads, as indicated by the attribute `program_name`.
- For the APACHE, there are two attributes to setup the system, and their different combinations constitute different workloads:
  - `requests` represents the number of requests to perform for the benchmarking session. The default is to just perform a single request which usually leads to non-representative benchmarking results.
  - `concurrency` represents the number of multiple requests to perform concurrently. The default is one request at a time.
- For the `SQLITE`, the worklodas are based on two system attributes:
  - `num` indicates the number of entries.
  - `value_size` represents the value size.

1.2 Summary of our computational resrouces

All of our data collection experiments were run on a cluster with 20 nodes, each of which is equiped with Intel® Core™ i7–8700 CPU@3.10GHz and 16GB memory. Evaluating all 86M configurations from the 3 systems with 5 repetitions took about 6 months to complete, which results in a total of more than 86, 400 CPU hours. For the landscape construction and analyses, all the experiments were carried out using a single node with Intel® Xeon® Platinum 8260 CPU@2.40GHz and 256GB memory.

2 Fitness Landscape Analysis

By representing the software configuration landscape as a directed graph[7], many classic fitness landscape analysis (FLA)

[1] https://llvm.org/
[2] http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/
[3] https://httpd.apache.org/
[4] https://httpd.apache.org/docs/2.4/programs/ab.html
[5] https://www.sqlite.org/index.html
[6] https://github.com/ukontainer/sqlite-bench
[7] Implemented using `NetworkX` package: https://networkx.org/.

Table 1: Selected configuration options for LLVM

Index	Parameter	Value
1	`inline`	{on, off}
2	`openmpopt`	{on, off}
3	`mldst-motion`	{on, off}
4	`gvn`	{on, off}
5	`jump-threading`	{on, off}
6	`correlated-propagation`	{on, off}
7	`elim-avail-extern`	{on, off}
8	`tailcallelim`	{on, off}
9	`constmerge`	{on, off}
10	`dse`	{on, off}
11	`slp-vectorizer`	{on, off}
12	`callsite-splitting`	{on, off}
13	`argpromotion`	{on, off}
14	`aggressive-instcombine`	{on, off}
15	`polly-simplify`	{on, off}
16	`polly-dce`	{on, off}
17	`polly-optree`	{on, off}
18	`polly-delicm`	{on, off}
19	`polly-opt-isl`	{on, off}
20	`polly-prune-unprofitable`	{on, off}

Table 2: Selected configuration options for APACHE

Index	Parameter	Value
1	`AcceptFilter`	{nntp, http}
2	`KeepAlive`	{on, off}
3	`KeepAliveTimeout`	$\{1, \dots, 300\}$
4	`MaxKeepAliveRequests`	$\{1, \dots, 2^{10}\}$
5	`TimeOut`	$\{1, \dots, 300\}$
6	`MaxConnectionsPerChild`	$\{1, \dots, 1,000\}$
7	`MaxMemFree`	$\{2^{10}, \dots, 2^{20}\}$
8	`MaxRequestWorkers`	$\{100, \dots, 3,000\}$
9	`MaxSpareThreads`	$\{50, \dots, 500\}$
10	`MinSpareThreads`	$\{20, \dots, 250\}$
11	`SendBufferSize`	$\{2^{10}, \dots, 2^{16}\}$
12	`ServerLimit`	$\{100, \dots, 3,000\}$
13	`StartServers`	$\{1, \dots, 10\}$
14	`ThreadLimit`	$\{10, \dots, 200\}$
15	`ThreadsPerChild`	$\{10, \dots, 200\}$

methods can be implemented in straightforward graph traversal manners. Here, we delineate the essential ideas and implementations of several FLA methods used in this paper.

Local optima. A local optimum is a configuration that has no superior neighbor. Once the landscape is represented as a graph, the local optima can be easily identified by finding the nodes with no outgoing edges, i.e., the *sink* nodes.

Basin of attraction. While a rugged landscape can be difficult to optimize due to the pressence of various local optima, not all are equal in terms of the capability of trapping a solver. For a 2D minimization case, this can be envisioned by the fact that each local optimum is located at the bottom of a 'basin' in the landscape surface. Configurations in each basin would eventually fall into the corresponding basin bottom, i.e., the local optimum, when following a simplest hill-climbing local search. To determine the basin of attraction of each local optimum in the landscape, we consider two most popular local search paradigms:

Table 3: Selected configuration options for SQLITE

Index	Parameter	Value
1	`SQLITE_SECURE_DELETE`	{on, off}
2	`SQLITE_TEMP_STORE`	$\{0, 1, 2, 3\}$
3	`SQLITE_ENABLE_AUTO_WRITE`	{on, off}
4	`SQLITE_ENABLE_STAT3`	{on, off}
5	`SQLITE_DISABLE_LFS`	{on, off}
6	`SQLITE_OMIT_AUTO_INDEX`	{on, off}
7	`SQLITE_OMIT_BETWEEN_OPT`	{on, off}
8	`SQLITE_OMIT_BTREECOUNT`	{on, off}
9	`SQLITE_OMIT_LIKE_OPT`	{on, off}
10	`SQLITE_OMIT_LOOKASIDE`	{on, off}
11	`SQLITE_OMIT_OR_OPT`	{on, off}
12	`SQLITE_OMIT_QUICKBALANCE`	{on, off}
13	`SQLITE_OMIT_SHARED_CACHE`	{on, off}
14	`CacheSize`	$\{1, \dots, 10, 240\}$
15	`AutoVacuumON`	$\{0, 1, 2\}$
16	`ExclusiveLock`	{on, off}
17	`PageSize`	$\{1, \dots, 10, 240\}$
18	`Wal`	{on, off}

- **Best-improvement local search** (Algorithm 1): In each iteration, the search moves to the neighbor with the highest fitness value. It terminates when no neighbor has a higher fitness value than the current configuration (i.e., a local optimum). For a graph-based landscape, this can be achieved by iteratively selecting the best *successor* of each node until a local optimum is encountered. The *best improvement basin* of a local optimum can be then determined by exhaustively perform such search from each configuration in the landscape, and collect all the configurations that fall into the same local optimum. Note that while this sounds like a computationally expensive task, in practice for landscapes with even millions of configurations, it would take only a few seconds to determine the basin of attraction of each local optimum.
- **First-improvement local search** (Algorithm 2): Here in each iteration, instead of selecting the best neighbor, the search moves to the first neighbor that it encounters with a higher fitness value. This is implemented by iteratively random selecting a successor of each node until a local optimum is reached. Under this paradigm, identifying the basin of attraction of each local optimum is equal to finding the *ancestors* of a node.

Autocorrelation. This is a widely used metric for characterizing the ruggedness of a landscape. As briefly introduced in the main paper, it is the autocorrelation ρ_a of a consecutive series of fitness values $\{f_1, \dots, f_n\}$ obtained from a random walk on the landscape. Due to the graph representation of the landscape, performing a random walk on the landscape is equivalent to that on a graph, which can be executed in a lightning fast manner in `NetworkX`.

Graph embedding. In this paper, we adopted `GL2Vec`, an improved version of `Graph2Vec` to extract low-dimensional features from the LON of each landscape. The generated features are able to capture the topological structure of the LON, and thereby the distribution and connectivity pattern of local optima in the landscape. We employed the implementation of

Table 4: Lookup table of settings of different running environments for three configurable software systems.

Sys.	LLVM	SQLITE		APACHE	
Index	`program_name`	`num`	`value_size`	`requests`	`concurrency`
1	2mm	10	100	50	50
2	3mm	10	1,000	100	100
3	atax	10	10,000	100	100
4	correlation	10	30,000	200	200
5	covariance	100	100	250	250
6	deriche	100	100	300	300
7	doitgen	100	1,000	400	400
8	fdtd2d	100	10,000	500	500
9	gemm	100	30,000	1,000	100
10	symm	1,000	10		
11	syr2k				
12	syrk				
13	trmm				

Algorithm 1: Best-Improvement Local Search

Input: A starting configuration $\mathbf{c}$; A neighborhood function $\mathcal{N}$; A fitness function f
Output: A local optima configuration $\mathbf{c}^\ell$

while $\mathbf{c}$ *is not a local optimum* **do**
  $\mathbf{c}'^\star = \text{argmax}_{\mathbf{c}'\in\mathcal{N}(\mathbf{c})}(f(\mathbf{c}'))$;
  if $f(\mathbf{c}'^\star) > f(\mathbf{c})$ **then**
    $\mathbf{c} \leftarrow \mathbf{c}'^\star$;
  else
    $\mathbf{c}$ is a local optimum;
    break;

Algorithm 2: First-Improvement Local Search

Input: A starting configuration $\mathbf{c}$; A neighborhood function $\mathcal{N}$; A fitness function f
Output: A local optima configuration $\mathbf{c}^\ell$

while $\mathbf{c}$ *is not a local optimum* **do**
  `Improve = False`;
  for $\mathbf{c}' \in \mathcal{N}(\mathbf{c})$ **do**
    if $f(\mathbf{c}') > f(\mathbf{c})$ **then**
      $\mathbf{c} \leftarrow \mathbf{c}'$;
      `Improve = True`;
      break;
  if *not* `Improve` **then**
    $\mathbf{c}$ is a local optimum;
    break;

`GL2Vec` from the `Karateclub` package.

3 Full Results for Sections 4.1 and 4.2

The full results related to Sections 4.1 to 4.2 can be found in Table 5, Figure 1, and Figure 2.

Table 5: Full results related to Figure 1 in tabular form, including the number of local optima in each landscape, and the statistics and p-value for comparing the distribution of local optima with random configurations sampled from each landscape.

System	Workload	n peaks	Stat.	p-value
LLVM	gemm	25,974	0.100	$7.6e^{-6}$
	3mm	17,918	0.145	$4.6e^{-2}$
	syrk	43,522	0.095	$2.9e^{-4}$
	trmm	24,658	0.092	$1.7e^{-4}$
	fdtd2d	39,012	0.092	$1.7e^{-4}$
	correlation	12,782	0.145	$4.6e^{-2}$
	2mm	17,494	0.145	$4.6e^{-2}$
	covariance	20,495	0.145	$4.6e^{-2}$
	syr2k	24,619	0.139	$2.5e^{-2}$
	deriche	36,631	0.092	$1.7e^{-4}$
	doitgen	32,148	0.092	$1.7e^{-4}$
	symm	42,962	0.095	$2.9e^{-4}$
	atax	15,388	0.092	$1.7e^{-4}$
APACHE	100_100	204,372	0.067	$9.4e^{-6}$
	200_200	184,939	0.067	$9.4e^{-6}$
	50_50	186,086	0.067	$9.4e^{-6}$
	1000_100	121,153	0.067	$9.4e^{-6}$
	300_300	165,537	0.067	$9.4e^{-6}$
	150_150	179,419	0.067	$9.4e^{-6}$
	250_250	177,716	0.067	$9.4e^{-6}$
	400_400	191,534	0.067	$9.4e^{-6}$
	500_500	128,825	0.067	$9.4e^{-6}$
SQLITE	1000_10	74,261	0.056	$3.2e^{-18}$
	100_100	75,324	0.056	$3.2e^{-18}$
	100_1000	72,908	0.056	$3.2e^{-18}$
	100_10	77,017	0.056	$3.2e^{-18}$
	100_10000	71,053	0.111	$2.9e^{-5}$
	10_1000	79,035	0.111	$2.9e^{-5}$
	10_30000	75,645	0.056	$3.2e^{-18}$
	10_10000	79,438	0.056	$3.2e^{-18}$
	10_100	78,869	0.056	$3.2e^{-18}$
	100_30000	66,680	0.111	$2.9e^{-5}$

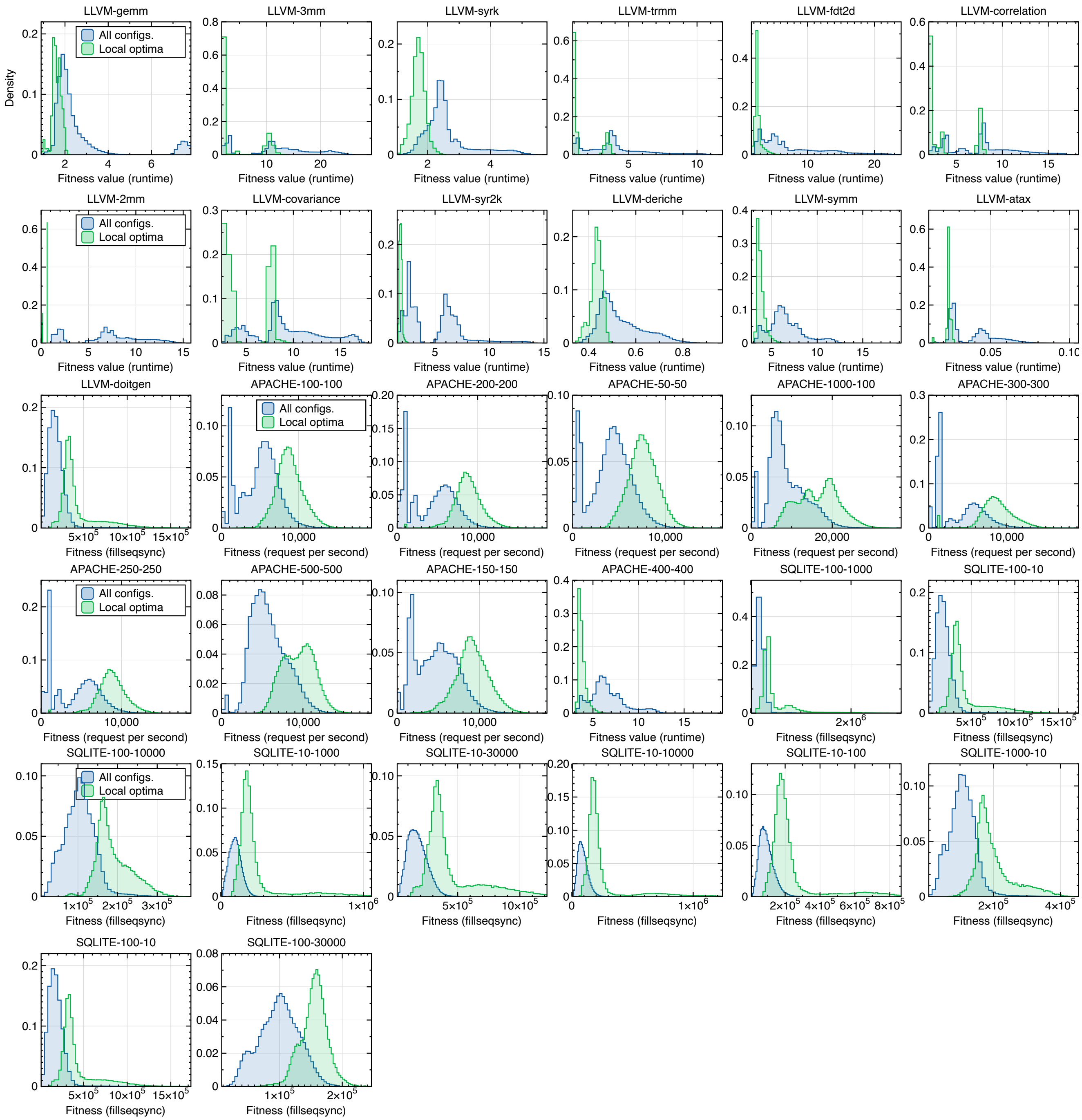

Figure 1: Comaprison of fitness distribution of all configurations (blue) versus local optima (green) for all studied landscapes. Note that while the objective function for LLVM is minimized, APACHE and SQLITE have maximized objectives.

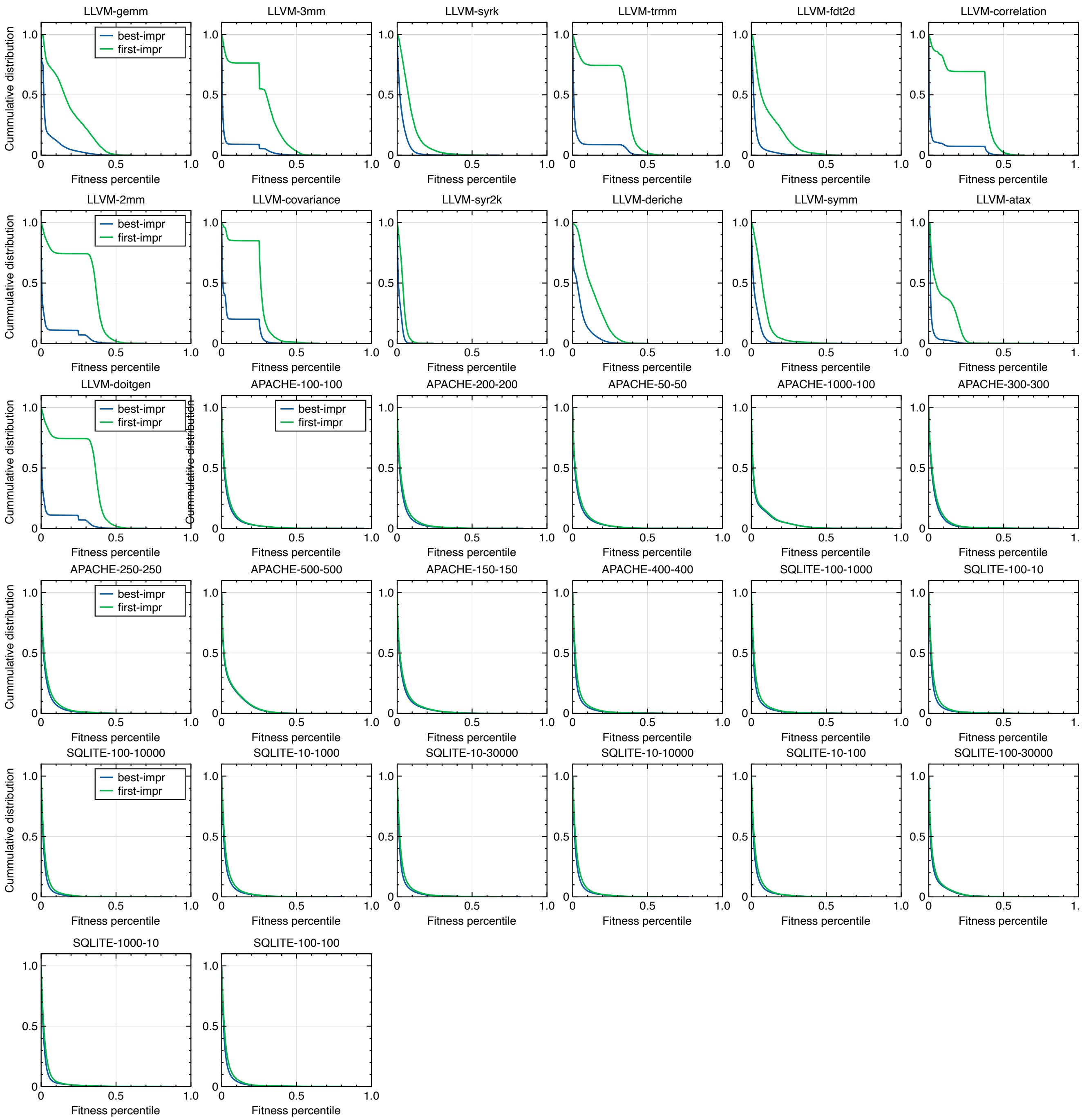

Figure 2: This plot provides the result for the Figure 3(A) and (B) (in the main text) across all scenarios, showing the cummlative distribution of basin size versus the fitness percentile of local optima under both best- and first-improvement local search. Note that for best-improvement, the basin size can be deterministically calculated, and $y = 1.0$ indicates the total sum of all basin sizes, which equals to the total number of configurations in the landscape. For first-improvment, the curves are approxiated by conducting 10^9 runs of randomized local search on the landscape, and hence $y = 1.0$ represents the total frequency of visits (i.e., 10^9).

4 Full Results for Sections 4.3 and 4.4

The full results related to Sections 4.3 to 4.4 can be found in Figures 3 to 5.

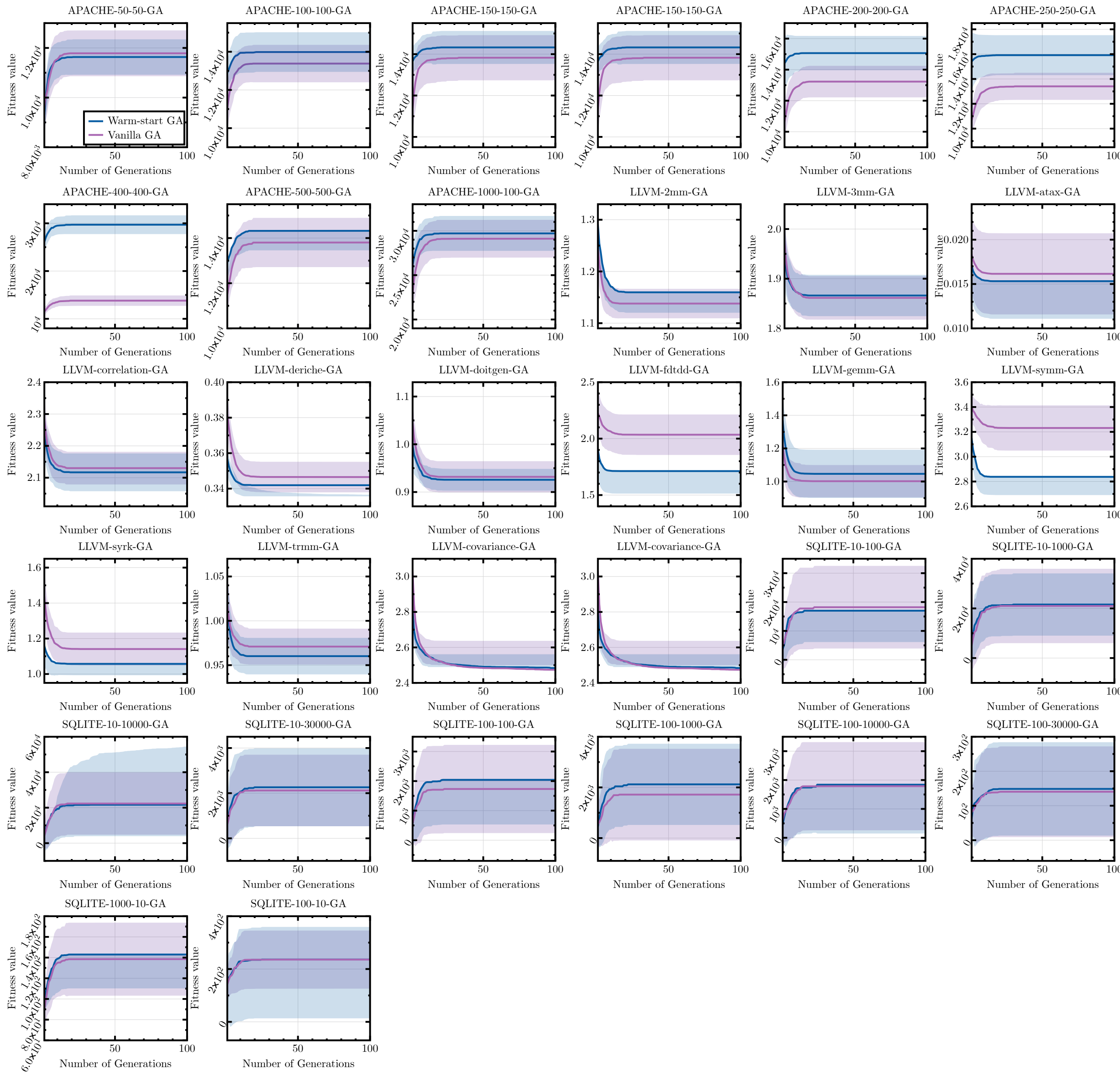

Figure 3: Evolutionary trajectories of warm-start GA (blue) against its vanilla version (purple) in 32 workloads. In particular, both algorithms are started with an initialized population of 50 and the total number of function evaluations is set to $5,000$. From these trajectories, we can see that the warm-start GA outperforms its vanilla counterpart, in terms of approximated optimal solution and the convergence rate, in over 78% cases.

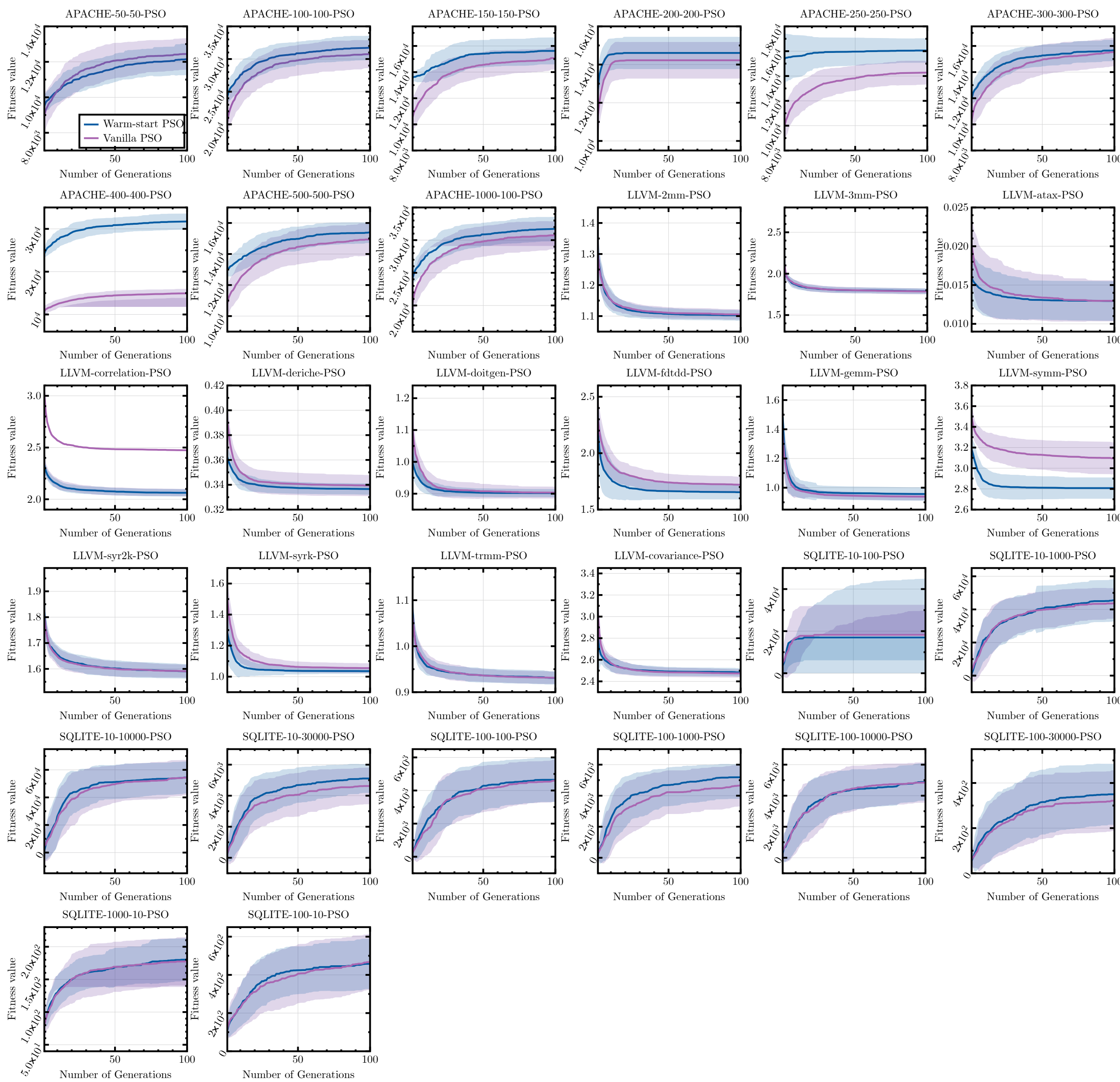

Figure 4: Evolutionary trajectories of warm-start PSO (blue) against its vanilla version (purple) in 32 workloads. In particular, both algorithms are started with an initialized population of 50 and the total number of function evaluations is set to $5,000$. From these trajectories, we can see that the warm-start PSO outperforms its vanilla counterpart, in terms of approximated optimal solution and the convergence rate, in 75% cases.

.

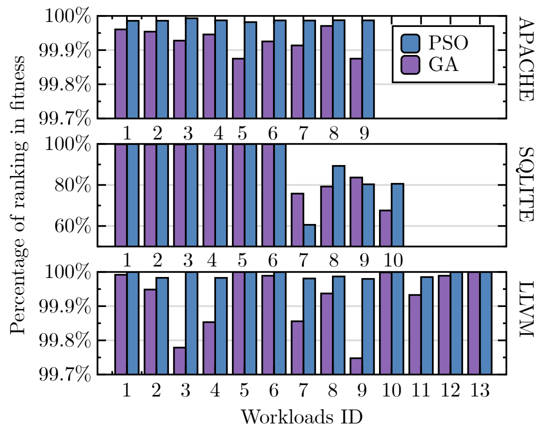

Figure 5: Bar charts of the average percentile of local optima approached by GA and PSO for LLVM, APACHE, and SQLITE on different workloads. From these comparison results, we can see that the solutions obtained by both GA and PSO are located in the top 0.1% local optima whose fitness value exceeds the 99.9% other local optima.