Classification-based Static Collection Selection for Java: Effectiveness and Adaptability

Noric Couderc
Lund University
Sweden
noric.couderc@cs.lth.se

Christoph Reichenbach
Lund University
Sweden
christoph.reichenbach@cs.lth.se

Emma Söderberg
Lund University
Sweden
emma.soderberg@cs.lth.se

ABSTRACT

Carefully selecting the right collection datastructure can significantly improve the performance of a Java program. Unfortunately, the performance impact of a certain collection selection can be hard to estimate. To assist developers there are tools that recommend collections to use based on static and/or dynamic information about a program. The majority of existing collection selection tools for Java (e.g., CoCo, CollectionSwitch) pick their selections dynamically, which means that they must trade off sophistication in their selection algorithm against its run time overhead. For static collection selection, the Brainy tool has demonstrated that complex, machine-dependent models can produce substantial performance improvements, albeit only for C++ so far.

In this paper, we port Brainy from C++ to Java, and evaluate its effectiveness for 5 benchmarks from the DaCapo benchmark suite. We compare it against the original program, but also to a variant of a brute-force approach to collection selection, which serves as our ground truth for optimal performance. Our results show that in four benchmarks out of five, our ground truth and the original program are similar. In one case, the ground truth shows an optimization yielding 20% speedup was available, but our port did not find this substantial optimization. We find that the port is more efficient but less effective than the ground truth, can easily adapt to new hardware architectures, and incorporate new datastructures with at most a few hours of human effort. We detail challenges that we encountered porting the Brainy approach to Java, and list a number of insights and directions for future research.

ACM Reference Format:
Noric Couderc, Christoph Reichenbach, and Emma Söderberg. 2023. Classification-based Static Collection Selection for Java: Effectiveness and Adaptability. In *Proceedings of The 24th International Conference on Evaluation and Assessment in Software Engineering (EASE 2023).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Most recent programming languages include a collection framework as part of their standard library (or runtime). For example, Java, C#, Python and Ruby all provide a collection framework. Collections typically implement an Abstract Data Type (ADT), which defines operations that the collection supports, and their semantics. In Java, the List, Map, and Set interfaces describe the associated ADTs, which are implemented in several classes. Collections that implement the same ADT are largely interchangeable.

However, even when collections are *semantically* interchangeable, their respective *performance* characteristics may vary. Liu and Rus [13] found that changing a single collection initialization parameter yielded a 17% run time reduction, while Jung et al. [9] reduced one benchmark's run time by 77% by swapping one collection datastructure for another.

The performance impact of *collection selection*, i.e., the choice between equivalent collection datastructures, depends on a complex interplay of factors. Developers may be aware of some factors, e.g., data structure size, but others (e.g., cache size) can be hard to grasp.

There are other factors beyond collection type and usage that may influence execution time. Jung et al. [9] showed that CPU architecture had a strong influence on running time for C++ programs, so that the optimal collection(s) for a task would be different for different machines. For Java programs, two additional factors could matter (in blue): the JVM implementation [16], and Just-In-Time (JIT) compilation and its "JVM warmup" [4] effect in Java.

Several existing approaches assist developers with collection selection, either statically, i.e., before execution [1, 9, 20], or dynamically, i.e., at run time [5, 26]. Tools that make static replacements can take as much time and gather as much data as needed to select which collection to instantiate. Dynamic approaches pay for profiling and decision-making at run time, so they must rely on simpler models and limited data. In exchange, they gain the ability to alter the collection selection during run time.

Existing approaches for Java use different models of collection performance for decision-making. CoCo [26] and Chameleon [20] use expert specifications as their models, and observe collection usage (e.g., method call counts) to take decisions. CollectionSwitch [5] uses micro-benchmarking to learn the cost of each operation in relation to collection size. However, if CPU architecture has a strong influence on which collection is best, as Jung et al. report [9], our tools should take the environment into account when making suggestions. Their Brainy approach is then promising, as it builds a model that factors in CPU behavior. However, the Brainy approach has so far only been realized for C++, and not for any languages whose implementations utilize just-in-time compilation, like Java.

In this paper, we report on our experience of using the Brainy approach for Java, focusing on the following research questions:

RQ$_1$ What are the technical challenges in porting the Brainy approach to Java?

EASE 2023, 14–16 June, 2023, Oulu, Finland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

	Manual	Automatic	Other
Static	Chameleon [20]	Brainy [9]	Artemis/NSGA-II [1]
Dynamic	CoCo [26]	CollectionSwitch [5]	

Table 1: Overview of collection selection assistance tools.

RQ$_2$ How effective is the Brainy approach at optimizing Java programs?

RQ$_3$ How adaptive is the Brainy approach when applied to Java?

RQ$_4$ What obstacles impair the effectiveness of the Brainy approach in Java?

To find answers to these questions, we have ported the Brainy approach to Java in an implementation we call Brainy4J. In this work, we have attempted to replicate the Brainy approach as far as possible, based on details available in the original paper [9] and from correspondence with the authors. In some cases we had to fill in missing details ourselves, or to make larger adjustments to adapt the approach to the needs of the Java environment.

We evaluate our system against a ground truth constructed from a brute-force selection of collections for 5 benchmarks from the DaCapo benchmark suite [3]. We find Brainy4J to be less effective at selecting collections than our ground truth. Brainy4J runs faster, but misses the most impactful optimization available. We note weaknesses in the Brainy approach that manifest in our setting, we highlight key challenges to overcoming them. We find Brainy4J highly adaptive to different environments, but find no strong evidence for the significance of CPU architecture details to Java collection selection.

The contributions of this paper are the following:

- Brainy4J, a port of the Brainy approach to Java.
- an evaluation of Brainy4J's effectiveness and adaptability.
- a list of challenges found when porting Brainy. along with an exploration of the design space for overcoming them.

2 RELATED WORK

Table 1 lists the most closely related tools. covering static collection replacement [1, 9, 20] and dynamic replacement via adaptive collections [5, 15, 26]. We further divide approaches between those that use hand-written performance models ("Manual"), and those that construct models via machine learning ("Automatic").

Static collection replacement. Shacham et al.'s 2009 Chameleon system [20] uses traces, heap information, and hand-written rules to select the collections to use. Basios et al's 2018 Artemis [1] uses a genetic algorithm to optimize the program directly, without a cost model. Instead, it explores possible variants by executing them in the cloud. Our work is a direct port of Jung et al.'s 2011 Brainy [9], which uses a machine learning model instead of hand-written heuristics (as in Chameleon) and running purely locally, with knowledge about the underlying architecture (unlike Artemis).

Adaptive collections. Xu [26] and Österlund et al. [15] present collections that switch implementations adaptively, based on usage. Österlund et al. present lists that switch between array and hashmap representations based on a state machine that tracks method calls. CoCo [26] minimizes copies by moving elements between different collections on demand. Costa et al.'s 2018 CollectionSwitch [5] builds on these tools to introduce smart constructors that select which type to instantiate. Similarly to Brainy, CollectionSwitch learns the relationship between the size of a data structure and its cost of operations via regression on micro-benchmarks. However, Brainy uses more complex models and suggests static changes.

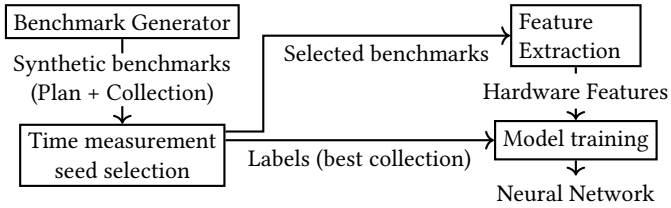


Figure 1: Overview of Brainy's classifier training process

Related approaches. The notion of re-usable software components dates to the early days of software engineering, with McIlroy proposing "catalogues of standard parts" that software engineers should be able to choose from [14]. This intuition sees components as units of functionality that should adhere to well-defined interfaces, possibly specified in a suitable formalism [7, 11, 17, 23]. For object-oriented programming, this idea matches Liskov's behavioural contracts for *substitutability* in subtype interfaces [12]. We here exploit that parts of the Java Collections Framework follow such contracts, but note that some aspects of their semantics are left to implementers, limiting substitutability.

At the language level, SETL [19] entirely hides the choice of data structure from developers to allow its compiler to effect speedups. For model-based or program refinement techniques [2, 10, 21] and for domain-specific languages like Spiral [18], prior research has shown the effectiveness of data structures selection, up to library-level auto-tuning [22]. While some modern dynamically typed languages similarly hide data structure details from their users and could use similar techniques, statically typed languages like Java or C++ require users to utilize explicit abstractions, such as the factory APIs exposed by CollectionSwitch.

3 BRAINY

We briefly review the most salient aspects of Brainy's approach to data structure selection before we discuss our adaptations (Section 4). Brainy tries to answer the question: *"what data-structure should I use, when the original data-structure behaves a certain way?"* [9]. Its premise is that developers provide Brainy with a program and source location, and it suggests the collection to use.

Brainy's centerpiece is a machine-learned classifier that takes information about (a) the current data structure, (b) collection usage, and (c) the CPU architecture, and proposes a replacement datastructure. To train this classifier (illustrated in Figure 1), Brainy collects training data from *synthetic micro-benchmarks* specific to each ADT. Brainy runs each micro-benchmark with each known datastructure for the ADT, both to determine execution time and to extract *collection usage features*. To optimize a given target program, Brainy extracts the same types of features from one or more target program executions and asks the classifier for a recommendation.

3.1 Building Brainy's training set

For each ADT, Brainy generates randomized micro-benchmarks to examine how different datastructures behave in different use cases. We identify each micro-benchmark with this ADT and with a *random seed*, from which Brainy synthesizes a **plan** — a sequence of operations to execute on one datastructure instance. Brainy fixes the length of all plans at 1000 methods. It first assigns a random weight to each ADT operation, then selects the 1000 operations one at a time with a probability proportional to their weight.

Phase I: Time measurements and benchmark selection. For each datastructure that implements the ADT, Brainy then measures how long the micro-benchmark takes to run, i.e., how well the datastructure performs for this specific plan. If one datastructure *wins* the micro-benchmark, meaning that it is at least 5% faster than all other datastructures, Brainy includes this micro-benchmark in its training set after labeling it with the winner. Brainy ensures that all datastructures win equally often: once Brainy has collected a fixed number of wins for a given datastructure (1000 in their evaluation), it discards any additional plans in which the same datastructure wins. We call this process *benchmark selection*. We hypothesize that the authors' motivation behind this step is to avoid the *class imbalance problem* [8], which in our context means that an imbalance in the number of wins will disproportionately penalize datastructures with fewer wins.

Phase II: Obtaining collection usage features. After Phase I has computed a set of benchmarks with clear "winner" datastructures, Phase II revisits each benchmark to extract collection usage features, i.e., information that Brainy can use to categorize *how* the micro-benchmark is using the datastructure. Brainy instruments the plan and re-executes it with each datastructure to collect features from four different categories: *hardware performance counter* features, specifically branch prediction and L1 cache miss rates, one *memory size* feature, the ratio between each collection element and the cache block size, *operation counts*, for each ADT operation, and three *cost* features, as we describe below.

Cost Features. Intuitively, choosing the right data-structure requires knowing what operations we want to execute, and how much time these cost: If we mainly insert elements but rarely read, we will likely prefer a datastructure for which insertion is cheap. Brainy therefore gathers the cost associated with *insertions*, *deletions*, *searches* (look-ups and iterations) during the benchmark. The authors measure this cost as follows: insertion and deletion (number of data elements moved forwards or backwards by the insertion), and search (number of data elements accessed before finding the element of interest).

Training data. At the end of Phase II, Brainy has collected training data in the form of micro-benchmarks, annotated with the winning data structure, and collection usage features for each datastructure. Brainy trains a family of artificial neural networks[1] with this data: one network per datastructure to replace.

3.2 Classification

For classification, Brainy obtains collection usage features and records the datastructure that it should replace, and queries the appropriate classifier. For example, if the user wants to replace a hash_set, Brainy will feed the features into a classifier specialized for hash_set replacement, the set ADT, and the user's current machine.

[1]We were unable to obtain details about the structure or size of these networks.

3.3 Evaluation of Brainy

The authors apply Brainy to nine datastructures (vector, list, deque, set, AVL_set, hash_set, map, AVL_map, and hash_map) and six ADTs (set, map, list, vector, and order-oblivious list and vector), and train it on two distinct Intel microarchitectures.

To test the training accuracy of Brainy, the authors generate a test set of 1000 benchmarks, one for each datastructure. They report an accuracy between 80% and 90% for the 2006 Intel Core microarchitecture, and an accuracy between 70% and 80% on the 2008 Intel Bonnell microarchitecture (Intel Atom).

The authors also compare Brainy's suggestions and their effect on four benchmark programs, each of them on three different workloads. They report that on average, Brainy reduces execution time by 27% on the Core microarchitecture, and by 33% on the Bonnell (Atom) microarchitecture. All benchmarks report at least 10% execution time reduction, with a maximum of 77% in one case.

4 PORTING BRAINY TO JAVA: BRAINY4J

In this section we address **RQ$_1$** by describing our strategy for implementing Brainy4J (Brainy for Java) and highlighting our design decisions in this process.

4.1 Selection of Collections

Following prior work [5, 20, 26], we selected collections from the Java Collection Framework's List, Set, and Map ADTs. After initial experiments, we concluded that we could not reproduce several of the optimizations from prior work purely with datastructures from the standard library, and added data structures from CollectionSwitch [5].

For lists, we selected ArrayList, LinkedList and HashArrayList from CollectionSwitch. For sets, we selected HashSet, TreeSet, and ArraySet from FastUtil [25], as used in CollectionSwitch (Chameleon [20] also reports using an ArraySet). For maps, we selected HashMap, TreeMap and ArrayMap [25].

These data structures are both the ones we consider as *targets* of a transformation, and as *sources*, meaning that we did not explore transformations e.g. from custom user-defined map implementations to standard library map datastructures. We additionally configured LinkedHashSet and LinkedHashMap as sources (only).

4.2 Datastructure Adaptability

We automatically replace data structures at the Java bytecode level. To enable replacement, we introduce suitable shared super-interfaces for all affected collections as needed, e.g. LinkedListInterface. Since two data structures may not offer the same different APIs even if they both implement the same ADT, we also added "universal" adapter subclasses for each data structure that inherit the shared super-interfaces and offer suitable adapter functionality. For example, LinkedList exposes a method `pop()` that ArrayList does not offer. The adapter subclass ArrayListUniversal, which inherits from ArrayList and implements LikedListInterface, implements this feature. Other methods we expose through default methods in

the super-interfaces. Our measurements show that these changes by themselves have no significant effect on execution time.

4.3 Feature Selection and Extraction

In their original study, Jung et al. selected one set of features for each collection using genetic algorithms. In our case, we used the features that they reported as being important for all collections.

To gather features we need to instrument benchmarks. We use our adaptability transformations as the foundation for our tracing framework. For each adapter interface, we synthesize wrapper classes that obtain call counts for each method, and the same hardware performance features used in Brainy (via JNI invocations to the PAPI C library [24]). The wrappers support concurrent operations via lock-free datastructures and discount recursive calls (e.g., calls from `addAll` to `this.add`). This instrumentation gives us two of the four feature categories used in Brainy, excluding the *memory size* and *cost* categories. We do not use the *memory size* feature, since Java's generic collections use boxed element representations. Our minimally invasive instrumentation strategy makes it challenging to obtain *cost* features. Instead, we aggregate cycle counts for four families of operations (insertions, deletions, iterations, lookups) as proxy metrics. When tracing the target program, we average features over two replications of ten runs, and discard data for the first three iterations. To measure running time and hardware performance counters needed to train our models, we followed best practices in using the Java Microbenchmarking Harness (JMH) [6].

4.4 Benchmarking and Model Training

To keep a balanced training set, Brainy rejects new benchmarks if they do not add new information. If collection C wins for benchmark b, but Brainy already has enough benchmarks for which collection C wins, b is rejected. In our case, this process was too slow to be practical, so we do not reject benchmarks (Section 7). We therefore explored a fixed number of 1000 seeds per ADT, and restricted exploration to datastructure elements of type `Integer`.

In their original study, Jung et al. used neural networks. We used random forests with 100 decision trees, which were as effective as neural networks at classification, while making it easy to measure the importance of each feature.

4.5 Allocation Site Selection

In their original study, Jung et al. manually selected an interesting allocation site for evaluating Brainy. However, typical Java programs often have hundreds of relevant allocation sites, which sites should be picked? To use a systematic, reproducible approach, we selected the ones we expected to be most significant for overall performance. We selected the 10 "busiest" allocation sites. To measure busyness, we instrumented our benchmarks to count the number of operations on each datastructure (including constructor calls), and summarized those per allocation site, counting only the ADTs that we are tracking and excluding allocations within the Java Standard Library.

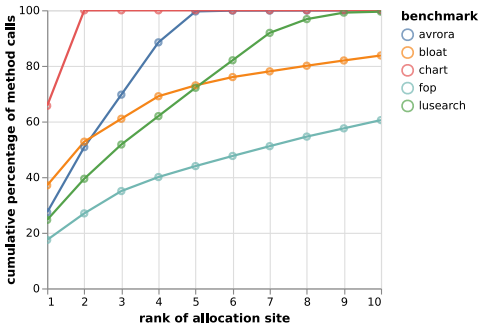


Figure 2: Cumulative collection operations and constructors aggregated for the top ten allocation site. For chart, two allocation sites account for more than 99% of the calls

5 EFFECTIVENESS OF BRAINY4J

To address $\mathbf{RQ_2}$, we examine how effective Brainy4J is at reducing the running time of Java programs. We structure this exploration around the following research questions:

$\mathbf{RQ_{2.1}}$ What are the model characteristics of Brainy4J?
$\mathbf{RQ_{2.2}}$ What are the costs in terms of time for using Brainy4J?
$\mathbf{RQ_{2.3}}$ How effective is Brainy4J compared to the ground truth?

5.1 Experimental Setup

For the experimental setup we need a selection of environments, a selection of Java benchmarks, a configuration of Brainy4J, and a ground truth to compare to.

Selection of Environments. As *environmental factors*, we considered the Java Virtual Machine and hardware configuration. We tested with various Java Virtual Machines but observed no significant differences. Therefore, all machines used OpenJDK 8.0.292 with a JVM heap size of 12 GB. To reduce noise in our measurements, we ran each system with CPU frequency scaling and hyperthreading disabled. Table 2 summarizes the systems we evaluated on.

Selection of Benchmarks and Allocation Sites. We evaluate the effectiveness of Brainy4J on the default workloads of five DaCapo [3] benchmarks, selected based on use in prior collection replacement studies (shown in Table 3). For each benchmark we considered the ten busiest allocation sites (Section 4.5) for replacement. Figure 2 shows the distribution of datastructure method calls for the top ten sites for the selected benchmarks. The top ten sites comprise 64% of all calls for fop, 73.3% for bloat, and $> 99.8\%$ for avrora, lusearch, and chart. We validated this selection mechanism by collecting the number of CPU cycles spent per allocation site and observed no substantial difference.

Brainy4J Configuration. For *training*, we generated 1000 micro-benchmarks per ADT, with 1000 operations each. For execution time, we used 3 replications (independent JVM runs), 2 warmup iterations lasting 500 ms each, and 5 measurement iterations, lasting 500 ms. Each micro-benchmark taking 5ms to run. To extract hardware features, we ran each micro-benchmark 10 times. We provide each sample to the classifier. For *classification*, we used a random forest with 100 trees. Our training data consisted of 9×1000 micro-benchmarks, iterated 10 times, for $90,000$ training samples with 35 features.

System	CPU	Cores	CPU Freq.	Microarch.	RAM	OS: Ubuntu	Kernel: Linux
Sandy	Intel i7-3820	4 × 2	3.6–3.8 GHz	Sandy Bridge	16 GiB DDR3-1600	18.04.6 LTS	5.4.0
Cypress	Intel i7-11700K	8 × 2	3.6–4.9 GHz	Cypress Cove	128 GiB DDR4-3200	22.04.01	5.15.0
Zen3	AMD EPYC 7713P	64 × 2	2.0–3.675 GHz	Zen 3	512 GiB DDR4-3200	22.04.01	5.15.0

Table 2: Our benchmarking environments.

Benchmarking. We ran 20 in-process iterations of each benchmark. For fop, this number was not sufficient to reach steady-state, so we ran 150 iterations. We replicated this measurement 20 times. The last 10 runs are used to estimate steady-state performance.

Ground Truth: Greedy Search. To understand how Brainy4J compares against the "best possible solution" given our search space, we performed a limit study, i.e., we estimate the maximum improvement we could hope to obtain.

Since the effort for exploring all possible combinations of replacements is exponential in the number of allocation sites (e.g., 59049 variants for fop just for the top ten allocation sites, each of which would take several minutes to benchmark), we selected the top ten allocation sites (sorted after number of calls) and opt for a 'greedy' strategy: we optimized each allocation site independently, and merged the results to produce an "expected best" candidate (requiring only 21 variants per benchmark). To compare two variants, we measured the 95% confidence interval of the difference in running times, in steady-state. We picked the variant for which the confidence interval shows a significant improvement. If not, we keep the original collection. This greedy search would yield the same result as a full exhaustive search (for the top ten allocation sites) if the impact of all datastructure replacements were independent of each other.

5.2 Results

We present the results structured around the research questions presented at the beginning of this section.

$RQ_{2.1}$: Model characteristics Brainy4J. To evaluate our model, we measured accuracy on 10-fold of cross-validation, while making sure the test set contains data only about benchmarks which are not in the training set. The model learns effectively, with an accuracy of more than 85% for all collections and machines. However, this is partially explained by an imbalance problem in the training data. We come back to this issue in Section 7.

$RQ_{2.2}$: Cost of Brainy4J (time). Brainy4J's end-to-end optimization time is much shorter than that of greedy search. Once the classifier is trained, building a variant of a benchmark takes between 60 and 90 seconds, including tracing, classification, and building the optimized program. Greedy search takes much longer, since optimizing a program requires building roughly twenty variants of the program, and running each variant roughly 400 times (20 replications, 20 iterations until steady-state). As a result, greedy search takes between one (fop) and 14 hours (avrora) to optimize one benchmark.

$RQ_{2.3}$: Effectiveness Brainy4J vs. Greedy Search. Figure 4 show the 95% confidence intervals of average speedup associated with different changes, as well as the speedup of the optimization by Brainy4J. We show each variant in three environments: for each of our three environments (Table 2), for steady-state performance [4].

We denote single-change variants as $\langle from \rangle \rightarrow \langle to \rangle @ \langle alloc\text{-}site \rangle$ to identify the datastructures we transform *from* and *to*. For example, AL→LL@1 switches an ArrayList to a LinkedList at the first (highest-ranked) allocation site.

We see that for 3 out of 5 benchmarks, the original program was well optimized, in that neither Brainy4J nor greedy search managed to optimize the benchmark significantly. For chart and avrora, we observed statistically significant changes, but the effects were below 1%. For lusearch, we did not observe any statistically significant improvement. For fop, greedy search found an optimization which yielded between 1 and 2% of speedup on Zen3 and Sandy, and Brainy4J finds it as well. For bloat, greedy search finds one effective optimization (HM→AM@2) that improves the running time by approximately 20% in steady-state by changing a HashMap to an ArrayMap. However, Brainy4J did not suggest this optimization, it either suggests to keep the HashMap, or suggests to use a TreeMap instead. Using a TreeMap does not work as the elements stored in the map are not comparable, so Brainy4J discards that suggestion.

Our plots show the running times of the greedily and Brainy4J-optimized variants, but these include several changes. To understand the effect of one single change on performance, we took each change suggested by either greedy search or Brainy4J, and plotted the running time of a variant making that single change.

Using an ArrayMap is effective for steady-state performance, but has a detrimental effect on performance at startup, we discuss this in Section 6.2.

Examining the individually selected transformations (Figure 3), we find no matches between the suggestions by Brainy4J and suggestions by Greedy search. None of the changes suggested by Brainy4J has a significant impact on running time. We suspect that the difference between Brainy4J and Greedy search is due to our difficulties in generating a balanced training data-set. We come back to this issue in Section 7.

		Tools			
		Artemis	Coco	CollectionSwitch	Chameleon
Benchmarks	avrora	x	x	x	
	bloat		x	x	x
	chart		x		
	fop	x	x	x	x
	lusearch		x	x	

Table 3: DaCapo benchmarks used in our work, and their use in prior studies on collection selection.

6 ADAPTABILITY OF BRAINY4J

To address $\mathbf{RQ_3}$, we examine how adaptive Brainy4J is to changes in the environment (CPU, JVM) and the configuration (e.g., adding a

new collection). We structure this exploration around the following research questions:

RQ$_{3.1}$ What is the effect of the CPU on recommendations?
RQ$_{3.2}$ What is the effect of the JVM on recommendations?
RQ$_{3.3}$ What is the cost of changing environment?
RQ$_{3.4}$ What is the cost of changing collections?

6.1 Experimental Setup

Understanding the effect of the CPU and JVM on recommendations hinges on two aspects: do the recommendations change, and if so, how does this change impact performance? In our case, we can look for recommendations in the training data, and in Brainy4J's recommendations for the DaCapo benchmarks.

To test if Brainy4J adapts to new CPU architectures, we ran the experiments from Section 5 on our three benchmarking machines. If CPU architecture matters, as in the original Brainy study, we should see different suggestions on different machines, both in the training data and in suggestions offered by greedy search.

To test if JVM implementation and warmup impacted collection suggestions, we ran greedy search on our different machines, with four different JVM implementations. To test if reaching steady state impacted the effect of changes, we measured the speedup of a greedy-search optimized variant in both startup and steady-state.

To evaluate the cost of changing environments (e.g. running Brainy4J on a new machine) and changing collections, we report on the time we spent performing such tasks.

6.2 Results

We present the results structured around the research questions.

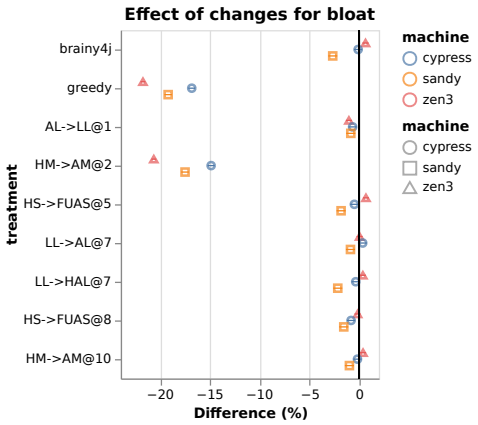
RQ$_{3.1}$: Effect of the CPU. For our DaCapo benchmarks, we observe some differences between machines, and see some variation in greedy search's suggestions too, but they concern changes that had little effect on performance.

For Brainy4J, we observe that suggestions are not the same between different machines, but in practice the changes do not have

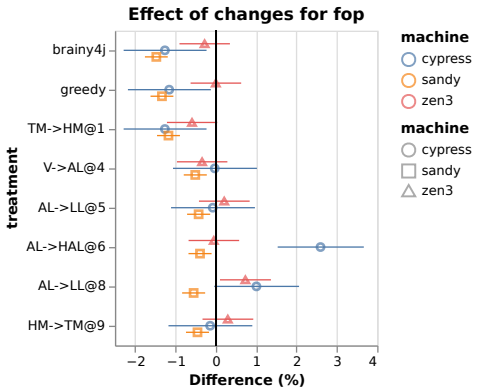
@	From	Brainy4J	M	Greedy	M
avrora					
1	HM	→ HM	S	→ TM	Z
2	LL	→ AL	C	→ AL	Z
2	LL	→ HAL	Z, S	→ AL	Z
3	LL	→ AL	C	→ AL	Z
3	LL	→ HAL	Z,S		S, C
5	LL	→ AL	all		all
6	HS		all	→ TS	Z
7	HM		all	→ AM	Z
8	HM	→ TM	S		all
9	HM		all	→ AM	Z
10	HM	→ TM	S		all
bloat					
1	AL			→ LL	all
2	HM			→ AM	all
5	HS			→ FUAS	C, S
7	LL	→ AL	C		all
7	LL	→ HAL	Z,S	→ HAL	S
8	HS			→ FUAS	C, S
10	HM			→ AM	S

@	From	Brainy4J	M	Greedy	M
chart					
3	AL		all	→ HAL	S
6	AL		all	→ LL	S
7	TM	→ HM	C	→ HM	S
8	AL		all	→ LL	S
9	TM	→ HM	C		all
fop					
1	TM	→ HM	C	→ HM	C, S
3	AL		all	→ LL	C
4	V		all	→ AL	S
5	AL		all	→ LL	S
5	AL		all	→ HAL	C
7	AL	→ LL	all	→ LL	C
8	AL		all	→ LL	S
9	HM	→ TM	S	→ TM	S
lusearch					
4	TM	→ HM	C		all
5	TM	→ HM	C		all
10	HM	→ TM	S		all

Figure 3: Transformations used by Brainy4J/greedy search on different machines (M), with Z=Zen3, C=Cypress, S=Sandy



(a)



(b)

Shorthand	Collection
AL	ArrayList
LL	LinkedList
HAL	HashArrayList
HM	HashMap
TM	TreeMap
AM	ArrayMap
HS	HashSet
TS	TreeSet
FUAS	ArraySet

Figure 4: Effect of changes for bloat (a) and fop (b). Points denote the mean speedup, error bars denote the 95% confidence intervals. Greedy search finds an important optimization for bloat, but Brainy4J misses it. For fop, Greedy search and Brainy4J find a small optimization for Sandy and Zen3 The table shows the shorthand labels we use for collection names in our notation for changes.

significant effects. As far as training and classification are concerned, we observed many similarities between different machines.

We compared the suggestions given in Brainy4J's training data for Cypress and Sandy, we see that they agree for 93.5% of benchmarks (n = 9000). This could show that CPU architecture does not

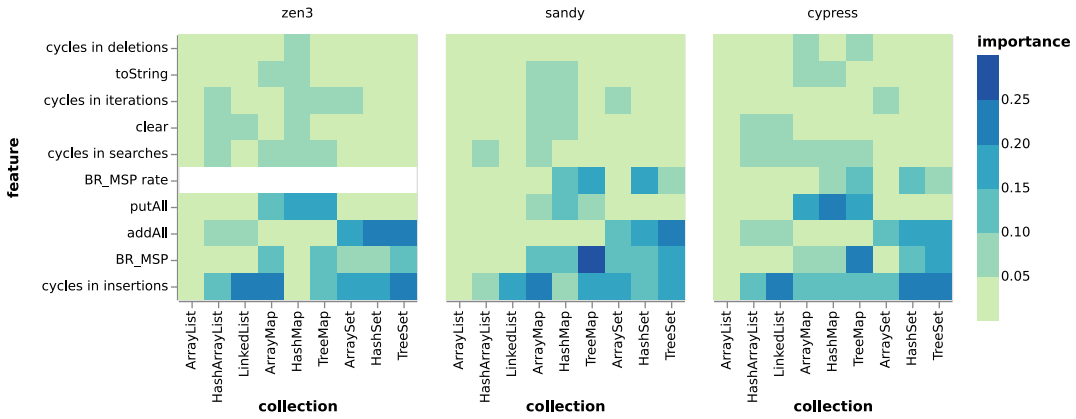


Figure 5: Importance of different features to the classifier, per machine. BR_MSP_rate is not available on Zen3.

startup	steady	interface	proportion
ArrayList	ArrayList	List	**99.4**
ArrayList	HashArrayList	List	0.6
HashMap	HashMap	Map	**94.3**
HashMap	TreeMap	Map	5.1
HashMap	ArrayMap	Map	0.6
HashSet	HashSet	Set	**79.1**
HashSet	TreeSet	Set	20.8
TreeSet	TreeSet	Set	0.01
TreeSet	ArraySet	Set	< 0.01

Table 4: Suggestions of best collection for training data, for all three machines. Startup and in steady state. When the tables agreed, the proportion is in bold.

matter, but it could also indicate that the architectures of both machines are quite similar. We compared on machines with two different CPU architectures: Cypress and Zen3 (one with an Intel CPU, one with an AMD CPU), and see that they also agree for 95.7% of benchmarks (n = 9000).

We looked at which features Brainy4J considers important for classifications, and also observed similarities between machines. Figure 5 shows the five most important features for classifiers specialized in each collection (x-axis), for all three machines. It displays what the classifier will look at when analysing the behavior of the collection. The color denotes the importance of the feature, as reported by the random forest classifier. On the y axis, `BR_MSP` refers to branch mispredictions, while `BR_MSP_rate` refers to the ratio of misprediction per branch instructions. Notably, the classifier for ArrayList does not consider any feature, because the training data always suggests to keep the ArrayList. Several sets of features are important on all three machines. Cycles spent in methods that insert in the collection ("cycles in insertions" in the figure) and number of calls to `Collection.addAll` and the corresponding Map operation `Map.putAll` are important. The number of branch mispredictions is also an important feature, for maps and sets. For lists, the number of cycles spent in insertions is the most important feature.

RQ$_{3.2}$: Effect of the JVM. We report all our results in this paper for OpenJDK 8.0.292. We validated the results on OpenJDK 18.0.1.1 and 11.0.12, and on GraalVM 22.1.0.r17 (excluding the bloat benchmark, which crashes[2] on the other JDKs), and observed no significant differences: both suggestions and running times are similar.

To study the effect of the JVM, we thus put focus on the impact of JVM warmup on the choice of collections, we examined (a) training data and (b) benchmark variants under both hot and cold JVMs.

First, we used Brainy4J's training data, comparing the suggested collections when only considering startup running time measurements (cold JVM) against steady-state measurements (hot JVM). We found that the suggestions agreed in 90% of all micro-benchmarks. Table 4 shows the suggested data structures and how often each tuple was suggested by the training data. We see that the suggestions are more diverse after warmup, but in the majority of cases, the same collection would be suggested both at startup and after warmup.

Second, we compared the effect of a single collection change, for startup and steady-state. We find that warmup can significantly affects different collections differently. For bloat, one collection change (HM→AM@2) is detrimental at startup but reduces steady-state run time by around 20%.

We find that all four instances in which we introduced HashArrayLists (two in avrora, one each in bloat and lusearch) are reliably detrimental at startup, but the negative effect vanishes after warmup. HashArrayList is our only datastructure without lazy initialization and depends on the Eclipse Collections framework, through a nontrivial chain of delegation, so that we expect it to benefit from inlining optimizations more than other datastructures.

RQ$_{3.3}$: Cost of Changing Environment. Switching to a new machine or a new JVM does not require an expert to re-write the performance model, as it might be the case for tools like CoCo and Chameleon [20, 26]. It is however necessary to re-train the classifiers. Training Brainy4J takes roughly 30-35 hours, for a training dataset of 9000 benchmarks (1000 seeds, 9 collections). Measuring the best collection for all benchmarks takes approximately 30 hours,

[2]We suspect that it fails due to a problem in a custom class loader, likely due to changes in class loader semantics introduced in Java 9 or later.

and gathering the features takes between 45 and 90 minutes. Training the classifier takes a few seconds. Measuring the running time of benchmarks takes most of the time, because we need to warm up the JVM before we can measure the running time [4].

$RQ_{3.4}$: Cost of Changing Collections. If Brainy4J can build a new model of collection performance, it should be relatively easy to add new collections to Brainy4J, comparatively to tools like CoCo, in which an expert would have to modify the performance model. We found that we need no more than a few hours to add a new collection to Brainy4J, and have partly automated this process. Re-running the micro-benchmark suite and training the model is then fully automatic and takes $\approx 30h$ for 1000 seeds. *Removing* a collection requires removing all training data from benchmarks that use this collection, and re-training the model which takes a few seconds.

7 OBSTACLES TO EFFECTIVENESS

To address $\mathbf{RQ_4}$, we examine interactions between Java and Brainy that may be obstacles to effectiveness. We structure this exploration around the following research questions:

$\mathbf{RQ_{4.1}}$ What is the effect of JIT-compilation?
$\mathbf{RQ_{4.2}}$ How effective is benchmark selection?
$\mathbf{RQ_{4.3}}$ What is the balance of the training data?
$\mathbf{RQ_{4.4}}$ Are our micro-benchmarks too short or too long?
$\mathbf{RQ_{4.5}}$ Is the type of elements important?

$RQ_{4.1}$: Effect of JIT compilation. A key challenge in adapting Brainy to Java is the increased distance between application code and machine. Brainy depends on measuring dynamic program features, some of which (hardware performance counters and running time) can vary with external factors, like machine architecture, but are also influenced by JIT compilation. The JIT introduces an additional dimension to our models — how "hot" or "cold" the code that we are optimizing is — and substantially affects micro-benchmark evaluation.

First, even if we only focus on "hot" code (which is likely to dominate performance in long-running programs), we need to iterate the same benchmark many times (3.5*s* per micro-benchmark). As a result, benchmark selection becomes prohibitively expensive. We have experimented with parallel execution but found that even on highly-parallel multicore systems with hyper-threading disabled, concurrent micro-benchmarking significantly altered our measurements. For Brainy, measuring a benchmark's running time was *fast*, but obtaining features was *slow*, for Brainy4J, both benchmarking and tracing are slow.

Second, we found evidence that our instrumentation interferes with JIT compilation. With tracing enabled, we observed an overhead of anywhere between 4% (lusearch on a cold JVM) and 1300% (fop on a hot JVM, on Zen3). The effect on warmed-up JVMs was generally more pronounced. In three cases, the overhead was greater than 400%, with more than 70% attributable to our general instrumentation overhead (JNI + Java, not counting PAPI). We speculate that these interactions are due to JIT optimizations not triggering. In almost all other cases, the cost for PAPI calls dominates the overhead. While we expect that this overhead causes a nontrivial amount of perturbation, we found that hardware performance counter measurements during benchmark execution remained stable across multiple runs of the program.

$RQ_{4.2}$: Effectiveness of Benchmark Selection. Jung et al. used benchmark selection (Section 3.1) to gather the same amount of training data for all datastructures, discarding excess "wins" for datastructures that already reached the preconfigured limit, and discarding micro-benchmarks where the winner reduced run time by less than 5% compared to the second-best datastructure. We ran benchmark selection with three different benchmark generation schemes: Uniform (choose methods with a uniform distribution), Brainy (our best-effort approximation of Brainy's selection scheme, cf. Section 3.1), and Markov (first learning a Markov chain from traces on fop, lusearch, and avrora, and using it as generator). For all generation schemes considered, we observed that ArrayList, HashMap and HashSet win far more often than the other collections. This could indicate that some collections dominate others regardless of context, or that our micro-benchmarks are biased towards a subset of the collections.

Benchmark selection did not find many benchmarks for which ArrayMap won, but greedy search chose ArrayMap as effective replacement twice, for bloat, including (HM→AM@2), the overall most impactful replacement. We suspect that benchmark selection is biased against ArrayMap, since ArrayMap works well on small maps, which we rarely create with a fixed length of 1000 method calls.

We found no synthetic benchmarks for which LinkedList won. This could suggest LinkedList is intrinsically inefficient, but Artemis [1] found 87 cases where LinkedList was more effective than ArrayList, and reports AL → LL as the most commonly proposed transformation. We assume that Brainy4J's benchmarks fail to show scenarios in which LinkedList shines and conclude that benchmark selection is ineffective in our setting.

We observe that even if benchmark selection were effective, it would be *inefficient*. If the most "unlucky" datastructure averaged one win per 100 micro-benchmarks (cf. zero wins for LinkedList), we would need 100k micro-benchmarks on average to observe 1000 wins for that datastructure. At ~3.5*s* of CPU time per datastructure and micro-benchmark, examining three datastructures would use over 11.5 days of CPU time, yet discard 97% of all measurements.

$RQ_{4.3}$: Balance of Training Data. Since we found benchmark selection ineffective in Brainy4J, we do not utilize it for benchmarking. As a consequence, Brainy4J's training data is imbalanced: our training data favors some collections over others (Table 4); for example, LinkedList does not appear at all. Table 4 shows that the same datastructure wins for the vast majority on benchmarks: ArrayList and HashMap win in more than 90% of cases, HashSet wins in more than 70% of cases. Our micro-benchmarks did not find a single case in which LinkedList wins on any of our three machines. We see the synthetic benchmark generation algorithm, the length of benchmarks, and the elements that we store in the collections as possible causes.

$RQ_{4.4}$: Number of Method Calls. We traced how many methods are called for each collection object in the five selected benchmarks. Over two million method calls, 98% o objects have less than 10 method calls, 1% 10 and 1000, and around 120 are much longer (up to more than a million calls). Both short traces and long traces are interesting for Brainy4J. Some of the sites we selected allocate

few collections that grow very big (long trace), while others allocate many collections that stay small (short trace per collection). Overall, we observe that the number of collection objects that receive 1000 or more method calls is negligible compared to the number of objects that receive ten or fewer calls. We note that this difference may reflect a difference in programming style between the C++ benchmarks that Brainy investigated and the Java benchmarks we examined.

$RQ_{4.5}$: Importance of Element Types. When training Brainy4J, our synthetic benchmarks only store integers in the collections they benchmark. Since we observed that HashArrayList's performance improved with warmup, we suspect that the cost of `hashCode` and `compare` (which might get cheaper with JIT compilation) plays an important role in the performance of sets and maps.

8 THREATS TO VALIDITY

Internal Validity. Jung et al. used genetic algorithms for *feature selection*. We used the features that they reported as being important. It is possible that another selection could improve the results of Brainy4J. The *translation of a plan into a micro-benchmark* could introduce overhead with effects on performance. To prevent this, we generated bytecode for each micro-benchmark. All *benchmarks suffer from tracing*: the program runs slower and the benefit from JIT compilation is smaller. This may have affected the readings Brainy4J used to make a decision. To make our micro-benchmarks comparable with their real-world counterparts, we normalized the hardware features by the total number of cycles, and the software features (number of calls to methods) by the total number of method invocations. However, this *normalization* does not capture the size of the collection, which showed to be important for CollectionSwitch. *Various factors can affect runtime performance when running benchmarks*. To prevent noise from disturbing the measurements, we disabled frequency scaling and hyperthreading and ran 20 replications per run. For micro-benchmarking, we relied on JMH. *Classification accuracy* depends on the split between training and test data. We used 10-fold cross validation and reported statistics on how accuracy varied on different datasets. We also ensured that samples for the training and test sets never came from the same micro-benchmarks.

External Validity. We used three different machines and five benchmarks. Still, adding more machines and benchmarks would help to further generalize the results.

9 DISCUSSION

Here, we look back at our three research questions:

What are the technical challenges in porting the Brainy approach to Java? (RQ_1) In porting the Brainy approach to Java, the main technical challenge was tracing collection usage and CPU behavior during runs without disturbing the measurements. We needed to generate collection classes that support tracing, and devise a scheme to substitute collection classes by tracing collections without breaking the program. Despite our precautions we still observe that tracing has a significant overhead.

How effective is Brainy4J at optimizing Java programs? (RQ_2) Our greedy search found statistically significant improvements for only one of our five benchmarks, driven by one high-impact replacement. Brainy4J's complex decision making mechanism failed to pick up the most important of these replacements. We expect that there is further potential for improvement: Artemis [1], another static selection tool, found statistically significant improvements (around 5% run time reduction) for both avrora and fop, though the authors do not report which changes they applied. Artemis considers three additional (concurrent) ADTs and several datastructures that we did not explore here, which may account for the differences. With Chameleon [20], Shacham et al. found even more dramatic improvements, but their baseline is a modified 2009 version of the J9 JVM, which makes it difficult to compare their findings to ours. One of their key findings was the importance of lazy datastructure initialization, which has since been added to all datastructures we considered, as of OpenJDK 8, with the exception of HashArrayList. The CoCo [26] and CollectionSwitch [5] approaches provide collections that can switch implementations at runtime. CoCo reports improvements on avrora by 11%, bloat by 4%, fop by 16%, and lusearch by 44%, albeit for Jikes RVM running on the Intel Nehalem microarchitecture. CollectionSwitch did not improve avrora's nor fop's running time, but improved bloat's running time by 22%, and lusearch's by 15%. Both CoCo and CollectionSwitch improved lusearch's running time by switching some instances of HashMap to ArrayMap, reducing memory usage for small maps. Our greedy search confirmed one such optimization for bloat, but similar changes to lusearch did not have the same effect. We speculate that the difference may come from the tools' dynamic nature, or from a different selection of allocation sites.

How adaptive is the Brainy approach when applied to Java? (RQ_3) Brainy4J can *adapt to new CPU architectures* and *different JVMs* without manual work. However, we were unable to identify cases in which switching architectures or JVMs affected the optimal decisions of our greedy search in a statistically significant way. By contrast, Brainy reported different optimizations for different architectures in their C++ benchmarks.

For *adapting Brainy4J's selection of datastructures*, we found that adding new datastructure takes at most a few hours of work to implement adapter subclasses and default operations (Section 4.2). We have partly automated this process but expect that more automation is feasible.

We expect that Brainy4J could be effective at *adapting to hot vs. cold JVM usage*, using a separate model for the latter case. We did not explore this direction but note that it may be significant for Java programs with short run times: the most effective optimization we observed (in bloat) was ineffective on cold JVMs, even incurring a significant slowdown on one machine.

What obstacles impair the effectiveness of the Brainy approach in Java? (RQ_4) We found two challenges in adapting Brainy to Java: the composition of the synthetic benchmarks, and the role of JIT compilation in the JVM. From what we see in our investigations (Section 7), the generated synthetic benchmarks struggle to exercise the strengths of some of our data-structures, like LinkedList. We have identified several possible causes for this challenge: the size of the synthetic benchmarks, how we generate plans and method

arguments, and our selection of elements to store in the datastructures. Brainy4J generated benchmarks with 1000 method calls, but we observed that real-world collection traces varied a lot in size. Brainy4J only stores integers in collections, and does not model the cost of methods like `hashCode`, `equals` or `compareTo`, though the cost of these methods may be crucial for deciding which collection to select. Lastly, we observed that JIT compilation plays a significant role in the effectiveness of collection changes, and found indications that Brainy4J's tracing instrumentation interferes with JIT optimization.

10 CONCLUSIONS

We were unable to find evidence that Brainy is effective when applied to Java. We observe two challenges caused by JIT compilation that reduce the effectiveness of the approach: First, the cost for benchmarking is higher than for C++, which prevented us from using benchmark selection, since it became too expensive to be practical. To adapt the Brainy approach to Java would require a different approach to generate the set of synthetic benchmarks. A better approach could be not to discard benchmarks (since benchmarking is expensive), and instead use the results. For example, by using regression to estimate the cost of running a benchmark.

Second, we suspect that Java JIT compilation is more sensitive to instrumentation than static compilation, especially since we must rely on JNI calls for gathering hardware performance counter data. This would reduce the accuracy of our models for reasoning about the performance of uninstrumented data structures when the JVM is hot. In summary, Brainy4J was not as effective as greedy search, and was less effective than dynamic tools, such as CoCo and CollectionSwitch.

Future Work. To make Brainy4J more effective, a possible approach would be to obtain more information from fewer benchmark runs. Currently, the benchmark generator does not use any feedback to build new benchmarks. For example, if it found a benchmark where LinkedList is very fast, it would not use this information to find other such benchmarks. One possible direction for future work would be to allow the benchmark generator to take inspiration from existing benchmarks to build new ones. In addition, our collections only contained integers, while hash maps and tree-based maps make heavy use of methods of elements, like `compare` and `hashCode`. One possible extension of this work would be to test different types of data for the elements stored in the collections, and the relationship between the cost of `hashCode` and `compare` on these elements. Finally, we do not know how tracing interacts with JIT-compilation and this could also be explored further.

ACKNOWLEDGMENTS

This work was funded by Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, Lake Buena Vista, FL, USA, 118–128. https://doi.org/10.1145/3236024.3236043

[2] F.L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. 1985. *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L*. Number 183 in Lecture Notes on Computer Science. Springer Verlag, Berlin, Heidelberg, New York, Berlin.

[3] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, and Thomas VanDrunen. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. (2006), 22.

[4] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (Aug. 2008), 83–89. https://doi.org/10.1145/1378704.1378723

[5] Diego Costa and Artur Andrzejak. 2018. CollectionSwitch: a framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. ACM Press, Vienna, Austria, 16–26. https://doi.org/10.1145/3168825

[6] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. 2019. What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* 47, 7 (2019), 1452–1467. https://doi.org/10.1109/TSE.2019.2925345 Conference Name: IEEE Transactions on Software Engineering.

[7] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. 2005. Semantic essence of AsmL. *Theor. Comput. Sci.* 343, 3 (2005), 370–412. http://research.microsoft.com/apps/pubs/default.aspx?id=77435

[8] Nathalie Japkowicz and Shaju Stephen. 2002. The class imbalance problem: A systematic study. *Intelligent data analysis* 6, 5 (2002), 429–449.

[9] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: effective selection of data structures. *ACM SIGPLAN Notices* 46, 6 (June 2011), 86–97. https://doi.org/10.1145/1993316.1993509

[10] Wolfram Kahl. 1999. The Term Graph Programming System HOPS. (March 1999), 136–149.

[11] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. *SIGPLAN Not.* 9, 4 (March 1974), 50–59. https://doi.org/10.1145/942572.807045

[12] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841. https://doi.org/10.1145/197320.197383

[13] Lixia Liu and Silvius Rus. 2009. Perflint: A Context Sensitive Performance Advisor for C++ Programs. In *2009 International Symposium on Code Generation and Optimization*. 265–274. https://doi.org/10.1109/CGO.2009.36

[14] M. D. McIlroy. 1968. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968).

[15] Erik Osterlund and Welf Lowe. 2013. Dynamically transforming data structures. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Silicon Valley, CA, USA, 410–420. https://doi.org/10.1109/ASE.2013.6693099

[16] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. 2021. Evaluating the Impact of Java Virtual Machines on Energy Consumption. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 11 pages. https://doi.org/10.1145/3475716.3475774

[17] David L Parnas. 1972. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.

[18] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232– 275.

[19] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. 1981. An Automatic Technique for Selection of Data Representations in SETL Programs. *ACM Trans. Program. Lang. Syst.* 3, 2 (1981), 126–143. https://doi.org/10.1145/357133.357135

[20] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), 11.

[21] Douglas R. Smith. 1990. KIDS: A Semi-Automatic Program Development System. In *Client Resources on the Internet, IEEE Multimedia Systems '99*. 302–307.

[22] Daniele G. Spampinato and Markus Püschel. 2014. A Basic Linear Algebra Compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. ACM, New York, NY, USA, Article 23, 10 pages. https://doi.org/10.1145/2544137.2544155

[23] Clemens Szyperski. 2003. Component Technology: What, Where, and How?. In *Proceedings of the 25th International Conference on Software Engineering* (Portland, Oregon) *(ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 684–693.

http://dl.acm.org/citation.cfm?id=776816.776916
[24] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*. Springer, 157–173.
[25] SEBASTIANO Vigna. 2006. fastutil 5.0. (2006). https://fastutil.di.unimi.it/
[26] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26.