Engineering Java 7's Dual Pivot Quicksort Using MaLiJAn*

Adventures with Just-In-Time Compilation

Sebastian Wild† Markus Nebel† Raphael Reitzig† Ulrich Laube†

Abstract

Recent results on Java 7's dual pivot Quicksort have revealed its highly asymmetric nature. These insights suggest that asymmetric pivot choices are preferable to symmetric ones for this Quicksort variant. From a theoretical point of view, this should allow us to improve on the current implementation in Oracle's Java 7 runtime library. In this paper, we use our new tool MaLiJAn to confirm this asymptotically for combinatorial cost measures such as the total number of executed instructions. However, the observed running times show converse behavior. With the support of data provided by MaLiJAn we are able to identify the profiling capabilities of Oracle's just-in-time compiler to be responsible for this unexpected outcome.

1 Introduction

In 2009, a new Quicksort variant due to Vladimir Yaroslavskiy was chosen as standard sorting method for Oracle's Java 7 runtime library. According to the Java core library mailing list [6], the decision for the change was based on empirical studies showing that on average, the new algorithm was faster than the formerly used classic Quicksort. Surprisingly, the improvement was achieved by using a dual pivot approach, an idea that had not been considered promising because of theoretical studies [12, 4, 15]. It has remained an open problem why theory and practice do not match for Yaroslavskiy's algorithm, even though Quicksort has been assumed to be well understood. A recent closer look at the algorithm has revealed that its new partitioning scheme is able to take advantage of certain asymmetries in the outcomes of key comparisons: On average, Yaroslavskiy's algorithm needs *5 % less* comparisons than classic Quicksort to sort a random permutation [15].

In order to make this theoretical study feasible, it was based on a simplistic version of the algorithm. In this paper, we adopt the perspective of a library designer who wants to investigate whether the alleged benefit of a proposed modification carries over into practice. This involves assessing the quality of all the tricks of the trade developed during decades of experience with practical implementations of classic Quicksort.[1]

The contribution of this paper is (a) to propose a possible improvement of the implementation in Java 7's library and use it as an example to show (b) how our tool MaLiJAn can be put to good use in automatically assessing the impact a small variation has on performance.

The motivation for our modification is the aforementioned asymmetry uncovered in [15]. Whereas the JRE7 implementation chooses tertiles-of-five as pivots [7] — a natural extension of the tried and tested median-of-three strategy in classic Quicksort — the asymmetric nature of the algorithm suggests that this symmetric choice may be *sub*optimal for Yaroslavskiy's Quicksort. Therefore, we investigate an asymmetric strategy for pivot selection.

We find that our asymmetric variant can indeed slightly reduce the expected number of executed Java Bytecode instructions asymptotically; we present closed-form estimates to this effect. Interestingly, running-time measurements clearly disagree. We identify Oracle's just-in-time (JIT) compiler as cause for the seeming paradox: The inputs used for gathering profiling information dominate actual running time and details of the experimental setup decide which sorting method is faster.

The examination is driven by *Maximum Likelihood Analysis* [10], implemented as MaLiJAn, which we propose as a general-purpose experimental methodology for investigating the impact small modifications to an algorithm have on running time. It is based on decomposing the algorithm at hand along its control-flow. Having access to individual frequencies of key instructions has proven instrumental to identifying the reason for the strength of Yaroslavskiy's Quicksort. Simply counting the *overall* number of key comparisons

*This research was supported by DFG grant NE 1379/3-1.

†Computer Science Department, University of Kaiserslautern, Email: {s_wild,nebel,r_reitzi,laube}@cs.uni-kl.de

[1] "It would be horrible to put the new code into the library, and then have someone else come along and speed it up by another 20 % by using standard techniques" (Jon Bentley in [6]).

would have shown *that* there are fewer comparisons in Yaroslavskiy's algorithm than in classic Quicksort, but not *why*.

1.1 Related Work. This is not the first time that asymmetries prove useful in connection with Quicksort. For the classic algorithm, the number of comparisons is minimized by using the median instead of a random pivot for partitioning. In [8], however, Kaligosi and Sanders have shown that a skewed pivot can be used to speed up classic Quicksort on hardware with slow rollback of CPU instruction pipelines in case of branch mispredictions. Even if such a biased choice for the pivot makes it impossible to achieve an optimal number of comparisons, the outcome of key comparisons becomes less random and this makes its prediction easier. Similarly, Martínez and Roura pointed out that skewed pivots can be beneficial in applications where swaps are much more expensive than comparisons [11]. In fact, the number of swaps is *maximized* when choosing the median as pivot, but their costs are outweighed by comparisons in classic Quicksort.

In the following, we first present our results on optimizing the pivot sampling used in Java 7's dual pivot Quicksort. Then, we describe the setup of the experiments and explain the methodology of MaLiJAn used to obtain the results.

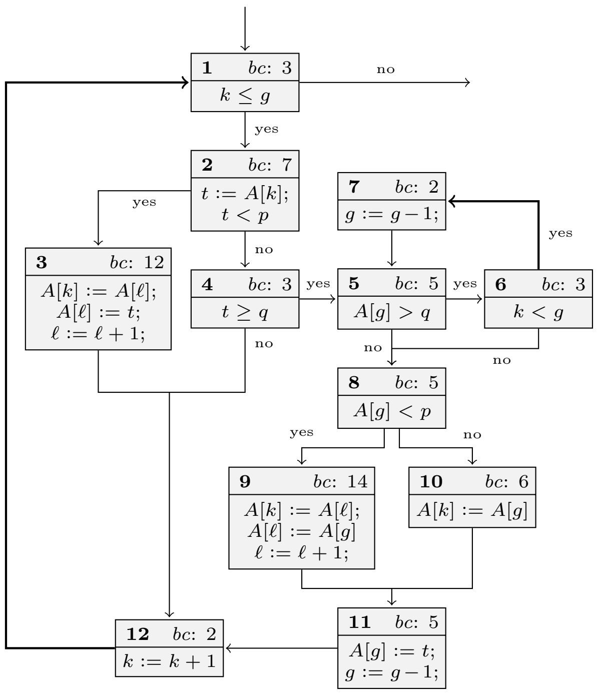

Figure 1: Control flow graph of the main partitioning loop of JRE7 (lines 20 – 34 of Listing 1 on page 13). These blocks are the only ones that are executed a *linearithmic* number of times, so they determine the leading term of costs. In the upper right corner of each block, the number of Bytecode instructions is given. Backward arcs are highlighted.

2 Java 7's Dual Pivot Quicksort

In 2009, Oracle changed the default sorting method of its Java runtime library: Since version 7, a highly-tuned implementation of Yaroslavskiy's dual pivot Quicksort is used for sorting arrays of primitive types [6, 7]. In comparison with the plain version of Yaroslavskiy's algorithm considered as Algorithm 3 in [15] (see Appendix A), the following optimizations found their way into the library implementation. We assume that we sort an array A of length n.

- **Pivot sampling:** The two pivot elements are chosen as the *tertiles of a sample of five elements*. To this end, five positions $s_1, \ldots, s_5$ of the list are selected such that they divide the list in regions with relative lengths as follows:

$A:$	3	s_1	2	s_2	2	s_3	2	s_4	2	s_5	3

  These five elements $S_1 = A[s_1], \ldots, S_5 = A[s_5]$ are sorted using Insertionsort. Denote the sorted elements by $S_{(1)} \le \cdots \le S_{(5)}$. Then, we choose the tertiles of the sample, i. e. $S_{(2)}$ and $S_{(4)}$, as pivots p and q, respectively. This is a natural generalization of the median-of-three scheme used in one-pivot Quicksort.

- **Short sublists:** For (sub-)lists that are shorter than a certain threshold, Insertionsort is used instead of Quicksort.

- **Equal elements:** Two optimizations aim at improving the handling of list with many equal elements. First of all, if the sample for pivot selection contains two equal elements, we fall back to the one-pivot Quicksort implementation of [1], which is known to be optimal for many duplicates [14]. In the dual pivot case, elements equal to p resp. q are collected in the middle partition and — with an additional scan — moved to their final positions at its respective borders, thus excluding them from recursive calls. This additional step is skipped if the middle partition is relatively short.

- **Almost sorted inputs:** Before Quicksort is called, the number of *runs*, i. e. sorted subarrays, in the input is determined. If the number of runs falls below a certain threshold, Mergesort is used instead of Quicksort, as this is more efficient on such highly-structured arrays.

In this paper, we will only consider arrays of *distinct elements*, that are *not* highly-structured, so the last two optimizations are not "active". We focus on the first optimization, pivot sampling and only use Insertionsort for sublists of size ≤ 4. The accordingly simplified JRE7 implementation can be found in Listing 1 on page 13.

3 Engineering Asymmetry

Choosing the tertiles of a sample as done in the original JRE7 implementation yields the most symmetric pivots we can infer from this sample. However, the asymmetries found in Yaroslavskiy's algorithm [15] suggest that an asymmetric choice of pivots might be better. But what kind of asymmetric choice really helps?

Asymmetric pivot choices trade lower costs in the current partitioning step for less balanced recursion trees. As it is not a priori clear how to find the optimal trade-off, we consider the general pattern behind the tertiles-of-five strategy: First, we select and sort a sample of five elements from the list. Then, we choose the pivots p and q as certain order statistics $S_{(x)}$ and $S_{(y)}$ for $1 \leq x < y \leq 5$. Denote by JRE7$_{(x,y)}$ the modified JRE7 Quicksort implementation with (x, y) pivot sampling. The original implementation is then JRE7$_{(2,4)}$ (we continue calling the original implementation JRE7 instead of JRE7$_{(2,4)}$ for short).

In order to find a good choice for x and y, we take a look at the control flow graph of JRE7. As MaLiJAn is based on the decomposition of a program along its control flow, it constructs the graph automatically. It also correctly identifies the hot spots of the algorithm, i. e. the basic blocks which are executed asymptotically most often. They are shown in Figure 1 on the preceding page. Execution of the loop is terminated once pointers k and g have crossed (exit condition of block 1). Thus, the number of iterations only depends on the length of the current sublist.

Accordingly, the overall number of loop iterations only depends on *how* balanced the recursion tree globally is, but not on the *direction* of asymmetry, i. e. whether pivots are larger or smaller than exact tertiles in expectation. The direction of asymmetry does however influence which paths through Figure 1 the iterations take: The ranks of the chosen pivots determine the odds for outcomes of comparisons in branching blocks. In total, there are five different cycles in Figure 1:

$$C_1 = 1 \to 2 \to 3 \to 12 \circlearrowleft$$
$$C_2 = 1 \to 2 \to 4 \to 12 \circlearrowleft$$
$$C_3 = 1 \to 2 \to 4 \to 5 \to 8 \to 9 \to 11 \to 12 \circlearrowleft$$
$$C_4 = 1 \to 2 \to 4 \to 5 \to 8 \to 10 \to 11 \to 12 \circlearrowleft$$
$$C_5 = 5 \to 6 \to 7 \circlearrowleft$$

We now assign each cycle as costs the number of executed Java Bytecodes on this path. Again, this is automatically determined by MaLiJAn:

$$bc(C_1) = 24, \qquad bc(C_2) = 15, \qquad bc(C_3) = 44,$$
$$bc(C_4) = 36 \qquad \text{and} \qquad bc(C_5) = 10\ .$$

We have to take into account that C_3 and C_4 actually count as *two* iterations, since k and g move two steps closer to each other on these paths. Ordering the cycles by costs implies the following preference: $C_5 \prec C_2 \prec C_4 \prec C_3 \prec C_1$ (cf. Figure 11 on page 9).

We can now use the branching information from the control flow graph to make more iterations choose cheap cycles. To get many executions of C_5, we need $A[g] > q$ to hold for many indices, so small values for q are preferable. Moreover, it pays to avoid expensive C_1, so we prefer $A[k] < p$ to hold for few indices. This also means p should be chosen smaller than JRE7 does. At the same time, we should not choose extremely skewed pivots in order to get a reasonably balanced recursion tree. Together, this makes JRE7$_{(1,3)}$ a promising candidate to challenge symmetric JRE7$_{(2,4)}$.

However, we should not rely on guesswork, so we do an exhaustive search among the 10 possible choices for order statistics (x, y). We measure the running time needed by JRE7$_{(x,y)}$ to sort a list of 10^6 integers, averaged over 1000 random permutations. The setup is as in Yaroslavskiy's benchmark [5]: First, each algorithm sorts a fixed random list 12 000 times without measurement to allow the just-in-time compiler to optimize code, see Section 4.2 for more discussion. Then, each algorithm is run on 1000 random permutations and the average running time is reported (see Table 1 on the next page).

As running time measurements are inherently machine-dependent, we also look at the number of executed Bytecodes. As inputs, we consider almost sorted lists using the random model of Brodal et al. [2] (described in detail in Section 4). Almost sorted inputs amplify the differences in pivot sampling: As the sample positions are spread throughout the array, an almost sorted list implies that the sample's order statistics are very close to those of the whole list. On 100 random lists of length 10^5, the algorithms execute the number of Bytecodes shown in Table 1. This second experiment was done entirely in MaLiJAn, as well. Counting executed Bytecodes is considerably more effort if done ad hoc compared to running-time measurements, so we feel MaLiJAn can be put to good use here.

The asymmetric JRE7$_{(1,3)}$ is consistently best in both experiments. For the rest of this paper, we therefore focus on the comparison of JRE7$_{(1,3)}$ and JRE7.

Order Statistics (x, y)	$(1,2)$	$(1,3)$	$(1,4)$	$(1,5)$	$(2,3)$	$(2,4)$	$(2,5)$	$(3,4)$	$(3,5)$	$(4,5)$
Running Time (ms)	94.06	92.90	92.92	103.2	95.86	93.31	96.40	96.38	95.62	98.79
Number of Bytecodes $(\cdot 10^7)$	2.332	2.177	2.292	—	2.273	2.218	2.470	2.499	2.581	3.083

Table 1: Running times on random permutation of length 10^6 and numbers of executed Java Bytecodes on almost sorted lists of length 10^5 for all JRE7$_{(x,y)}$ variants.
For JRE7$_{(1,5)}$, no count could be determined as it caused stack overflows. This algorithm experiences quadratic worst case behavior on already sorted lists, so it is not a suitable candidate for library sort.

4 JRE7 vs. JRE7$_{(1,3)}$

In this section, we quantitatively study the efficiency of our proposed asymmetric pivot sampling strategy. To the authors' knowledge, there are no established benchmark input sets for sorting algorithms, so we confine ourselves to artificial input distributions. Among those, *random permutations* are a natural choice which are also well-understood from the theoretical viewpoint. Additionally, we consider *almost sorted* inputs. As argued above, such inputs intensify the impact of pivot sampling, thus providing a clearer distinction. We use the random model for almost sorted inputs by Brodal et al. [2]: To generate an input, each element $A[i]$ $(i = 0, \ldots, n-1)$ is chosen uniformly at random from $\{i-d, \ldots, i+d\}$, ensuring that it is different from all previous elements. Finally, elements are relabeled to $\{1, \ldots, n\}$ while preserving the relative order. We choose $d = 100$ independently of n in order to obtain rather strongly presorted lists. As the expected number of runs of such lists is very high, the JRE7 sorting method will indeed invoke Quicksort.

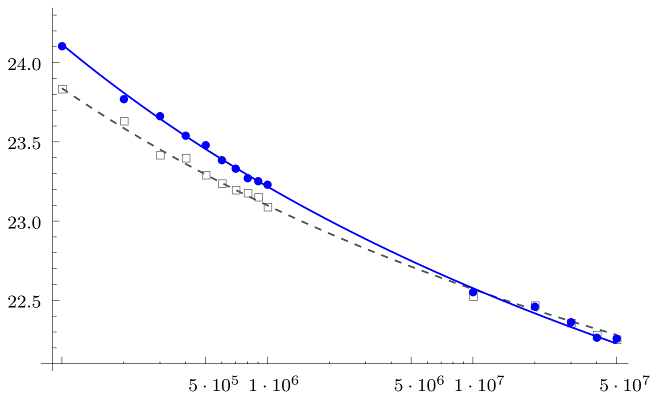

Figure 2: Predicted growth function for the number of executed Bytecodes on random permutations for JRE7 (gray dashed line and squares) and JRE7$_{(1,3)}$ (blue solid line and circles), normalized by $n \ln n$. The logarithmic horizontal axis depicts the input size. The model was trained on sizes up to 10^6 and nicely fits the larger data, as well.

Note that *equal elements* are dealt with by a specialized partitioning method in JRE7, which does *not* make use of two pivots. For these inputs, our variant behaves identically to the JRE7 implementation. Thus, we exclude the case of equal elements from our present discussion.

For both input distributions, we consider combinatorial cost measures and actual running times. Whereas abstract combinatorial measures can be misleading since they hide technical details, measured running times are machine-specific and highly sensitive to the experimental setup (see Section 4.2). By combining them, we can hope to find general trends in abstract measures that are confirmed by running time experiments.

4.1 Combinatorial Cost Measures. Combinatorial cost measures do not depend on details of the actual machine and can be counted deterministically for given algorithm and input. For sorting algorithms, the most prominent examples are the number of swaps and key comparisons. However, these measures are often too abstract for ranking algorithms by efficiency. Therefore, we consider also the number of executed Java Bytecode instructions (bc). For a given Java Bytecode implementation of an algorithm and a given input, bc can be determined exactly — MaLiJAn does it for us fully automatically.

MaLiJAn does not only count the desired measures for given inputs but also computes closed-form asymptotic extrapolations from basic-block-wise counters as described in Section 5.1. The corresponding results are shown in Tables 2 and 3.

Random Permutations. Table 2 on the facing page shows the asymptotics for the random permutation input model. For the number of swaps and comparisons, analytically proven results are available to compare the empirical ones against (combining results of [4] and [15]). The terms given by MaLiJAn are in good accordance with them: For JRE7, the correct expected number of comparisons is $1.70426\, n \ln n + \mathcal{O}(n)$ (MaLiJAn: $1.680 n \ln n$) and the number of swaps is $0.551378\, n \ln n + \mathcal{O}(n)$ (MaLiJAn: $0.574\, n \ln n$). For our asymmetric variant JRE7$_{(1,3)}$, we likewise have $1.86813\, n \ln n + \mathcal{O}(n)$ (MaLiJAn: $1.837\, n \ln n$) compar-

Algorithm	#comparisons	#swaps	#bytecodes
JRE7	$1.680\,n\ln n + 0.41n$	$0.574\,n\ln n + 0.84n$	$19.40\,n\ln n + 51.1n$
$\text{JRE7}_{(1,3)}$	$1.837\,n\ln n + 0.66n$	$0.456\,n\ln n + 1.17n$	$18.73\,n\ln n + 62.0n$
Yaroslavskiy	$1.9\,n\ln n + \mathcal{O}(n)$	$0.6\,n\ln n + \mathcal{O}(n)$	$23.8\,n\ln n + \mathcal{O}(n)$
Classic	$2\,n\ln n + \mathcal{O}(n)$	$0.\overline{3}\,n\ln n + \mathcal{O}(n)$	$18\,n\ln n + \mathcal{O}(n)$

Table 2: Expected cost measures for different Quicksort algorithms in the random permutation input model. The algorithms are the JRE7 Quicksort implementation, our asymmetric variant $\text{JRE7}_{(1,3)}$ thereof, Yaroslavskiy's basic algorithm (Algorithm 3 of [15]) and classic Quicksort as studied in [13]. Results for the two JRE7 variants are obtained using MaLiJAn. The formulæ for classic Quicksort and Yaroslavskiy's algorithm and can be found in [13] resp. [15].

Algorithm	#comparisons	#swaps	#bytecodes
JRE7	$1.162\,n\ln n + 2.28n$	$0.335\,n\ln n + 1.88n$	$15.10\,n\ln n + 67.9n$
$\text{JRE7}_{(1,3)}$	$1.170\,n\ln n + 3.56n$	$0.220\,n\ln n + 1.99n$	$13.52\,n\ln n + 84.9n$

Table 3: Expected cost measures for JRE7 Quicksort implementation and our asymmetric variant $\text{JRE7}_{(1,3)}$ for almost sorted input data (as defined in Section 4). All results are obtained using MaLiJAn.

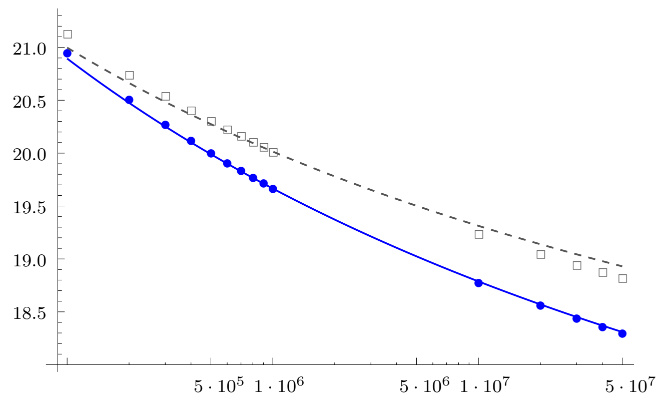

Figure 3: Predicted growth function for the number of executed Bytecodes on almost sorted arrays for JRE7 (gray dashed line and squares) and $\text{JRE7}_{(1,3)}$ (blue solid line and circles), normalized by $n \ln n$. The logarithmic horizontal axis depicts the input size. The model was trained on sizes up to 10^6 and fits the larger data, as well.

isons in expectation and $0.43956n \ln n + \mathcal{O}(n)$ (MaLiJAn: $0.456\,n \ln n$) swaps. As the other results are obtained from the very same trained stochastic model, we have confidence in MaLiJAn's results for the number of executed Bytecodes as well. In addition, we validate the asymptotics by comparing them with measurements from large inputs which were not used for training the model. Figure 2 on the preceding page shows that the closed forms accurately predict the expected number of Bytecodes for lists fifty times larger than the training data.

Almost Sorted Inputs. The closed form estimates of the combinatorial cost measures are given in Table 3. To the authors' knowledge, no analytic results are known for this input distribution. Again, we check our closed form asymptotic against larger measurements; see Figure 3.

In both input models, the symmetric pivot choice implies less comparisons but more swaps than our asymmetric variant. This is in line with corresponding results for classic Quicksort [11]. Moreover, the expected number of Bytecodes needed by $\text{JRE7}_{(1,3)}$ to sort an array is asymptotically less than with JRE7 for both input distributions. This shows that our asymmetric pivot sampling choice has the potential to increase Quicksort's efficiency.

Finally, we would like to stress how well MaLiJAn's asymptotic model fit the measurements of inputs *fifty* times larger than the training set. For distinctly different behavior — e. g. for almost sorted inputs — we can certainly rely on MaLiJAn's predictions.

4.2 Running Time. When aiming for practical applicability of results, only looking at the number of executed Bytecodes can be misleading. For example, Table 2 shows that classic Quicksort uses much less Bytecodes than Yaroslavskiy's basic algorithm. However, running time comparisons show converse behavior [15]. Therefore, this section complements the combinatorial measures with actual running times.

The measured runtimes as shown in Figure 4 (right) exhibit a strange feature. There seem to be two classes of inputs: some run significantly faster than others, and the two types are clearly separated. While clearly visible for random permutations, the effect is even more pronounced for almost sorted data; see Figure 5.

The bifurcation can not be found in any of the combinatorial measures, see for instance the number of executed Bytecodes in Figure 4 (left). Therefore, some

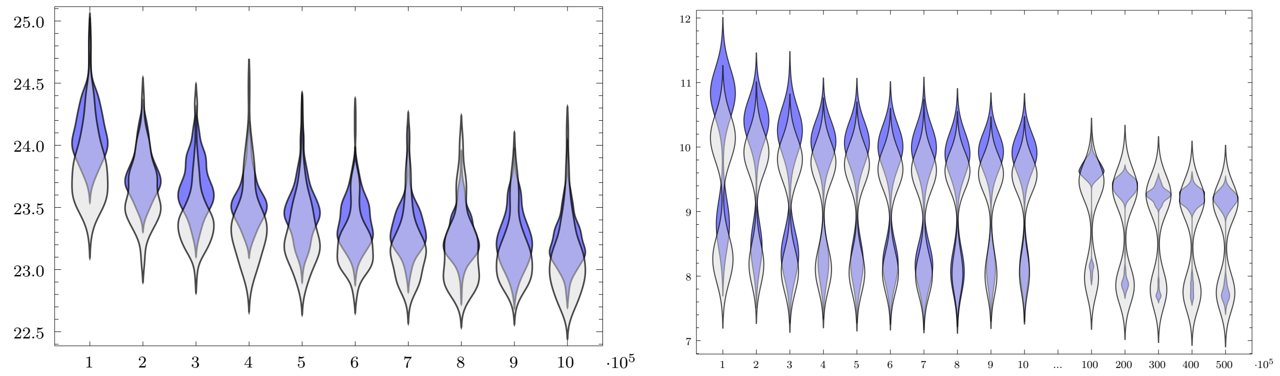

Figure 4: Violin plots for the observed number of executed Bytecodes (left) resp. runtimes on random permutations for JRE7 (light gray) and JRE7$_{(1,3)}$ (dark blue), normalized by $n \ln n$. The horizontal axis depicts the input size.

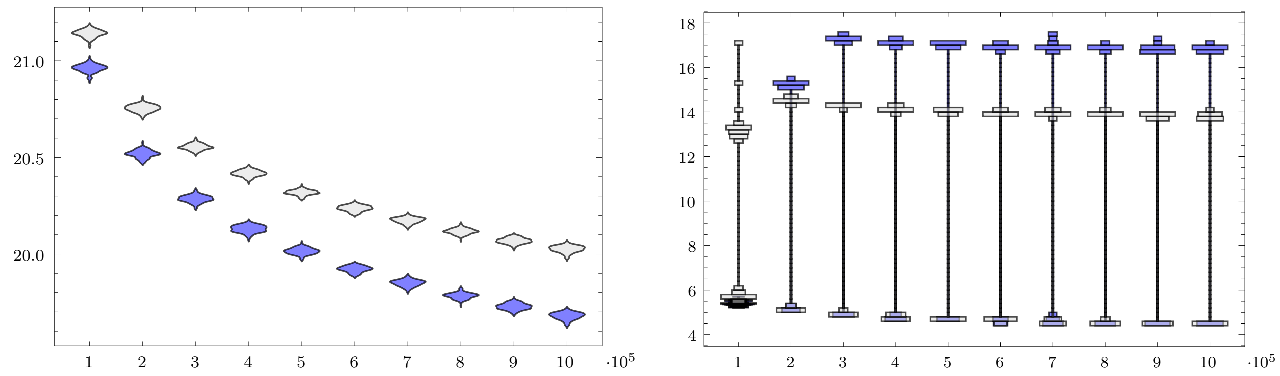

Figure 5: Violin plots for the observed number of executed Bytecodes (left) resp. runtimes (right) on almost sorted data for JRE7 (light gray) and JRE7$_{(1,3)}$ (dark blue), normalized by $n \ln n$. The horizontal axis depicts the input size.

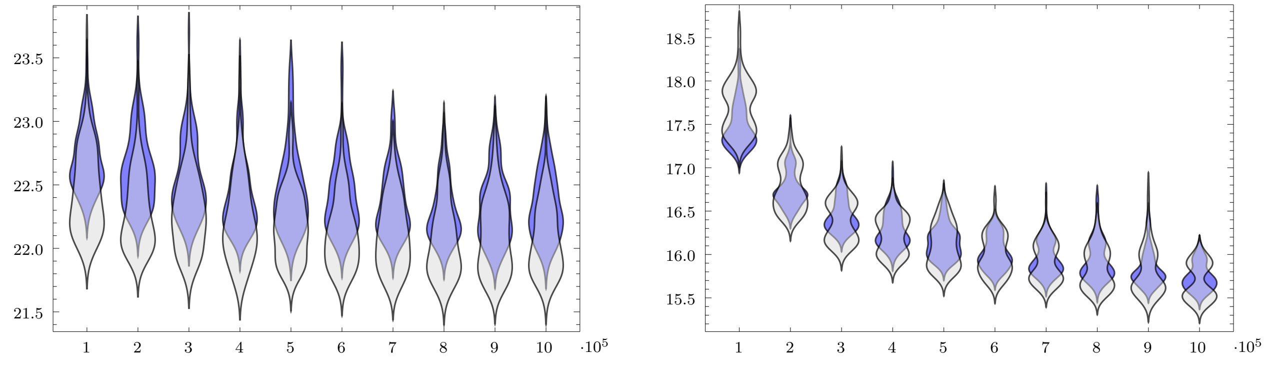

Figure 6: Violin plots for the observed running times on random permutations (left) resp. almost sorted data (right) for JRE7 (light gray) and JRE7$_{(1,3)}$ (dark blue) without JIT profiling, normalized by $n \ln n$. The horizontal axis depicts the input size.

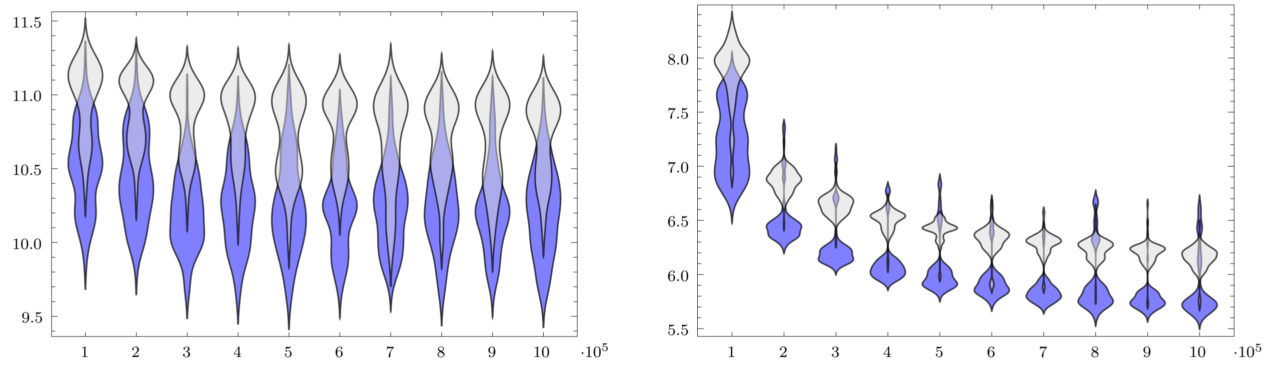

Figure 7: Violin plots for the observed running times on random permutations (left) resp. almost sorted data (right) for JRE7 (light gray) and JRE7$_{(1,3)}$ (dark blue) with JIT-warmup on fixed min, normalized by $n \ln n$. The horizontal axis depicts the input size.

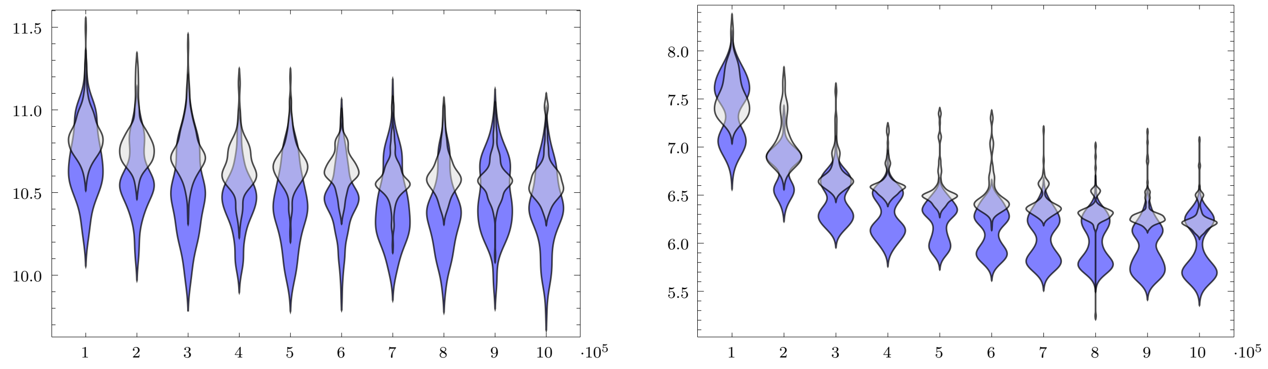

Figure 8: Violin plots for the observed running times on random permutations (left) resp. almost sorted data (right) for JRE7 (light gray) and JRE7$_{(1,3)}$ (dark blue) with JIT-warmup on fixed max, normalized by $n \ln n$. The horizontal axis depicts the input size.

runtime effect of the JVM has to be responsible. In fact, the split vanishes completely — for both input distributions! — if we prohibit the just-in-time compiler (JIT) from profiling the running code. This is possible by passing `-Xcomp` to the JVM, which forces it to compile the Bytecode at hand once at program start. In essence, we prevent data-dependent compiler optimisation. See Figure 6 for the resulting distribution of runtimes. This curiosity warrants further investigation.

So far, we have measured runtimes in isolation; for each input, we start a new JVM and run the algorithm a fixed number of times on it. This implies that the JIT collects profiling data on the same input the algorithm is ever run on (in this JVM instance). What happens if we force the JIT to profile on a fixed input instead? This is what Yaroslavskiy's benchmark [5] does, after all; interestingly, JRE7$_{(1,3)}$ shows slight improvements over JRE7 for most considered input types there, whereas only very few exhibit worse running times (see Appendix D). Our data from Figure 4 and 5 can not support this; maybe the behaviour of JIT is responsible for the seemingly bad runtime performance of JRE7$_{(1,3)}$?

We have therefore chosen two fixed inputs, namely those that performed best ("fixed min") resp. worst ("fixed max") in the model without explicity warmup (for a given input size), and repeat the whole runtime study for each of the two, with the only difference being that the JIT is "warmed up" with the respective input before measuring runtimes. Both experiments fail to reproduce the bifurcation; see Figures 7 and 8, respectively.

Also, the average runtimes with warmup on fixed min and fixed max are not nearly as far apart as in the non-warmup study. In fact, *all* choices for the warmup input lead to similar average runtimes (verified on a fixed input size). We conclude that runtime distributions and even averages depend heavily on which optimization JIT performs. It is unclear which warmup model is more realistic, and how to control resp. guide JIT towards an optimization that is good in expectation. It is not even clear whether there is a globally optimal choice, at all.

	JRE7 rp	$\text{JRE7}_{(1,3)}$ rp	JRE7 Brodal	$\text{JRE7}_{(1,3)}$ Brodal
`-Xcomp`	$20.099\, n \ln n + 26.0\, n$	$19.946\, n \ln n + 31.6\, n$	$11.950\, n \ln n + 54.1\, n$	$11.092\, n \ln n + 64.0\, n$
no warmup	$8.136\, n \ln n + 12.5\, n$	$8.344\, n \ln n + 14.8\, n$	$8.312\, n \ln n + 21.5\, n$	$9.845\, n \ln n + 37.0\, n$
fixed min warmup	$10.023\, n \ln n + 9.4\, n$	$11.387\, n \ln n + 15.4\, n$	$5.518\, n \ln n + 13.0\, n$	$5.375\, n \ln n + 19.0\, n$
fixed max warmup	$9.918\, n \ln n + 9.4\, n$	$11.604\, n \ln n + 15.7\, n$	$5.619\, n \ln n + 12.8\, n$	$5.465\, n \ln n + 19.3\, n$

Table 4: Asymptotic running time models for the two Quicksort variants and the random permutation (rp) and almost sorted (Brodal) input distribution. Basic block running times are determined by block sampling (as described in Section 5.2) using different JIT modes.

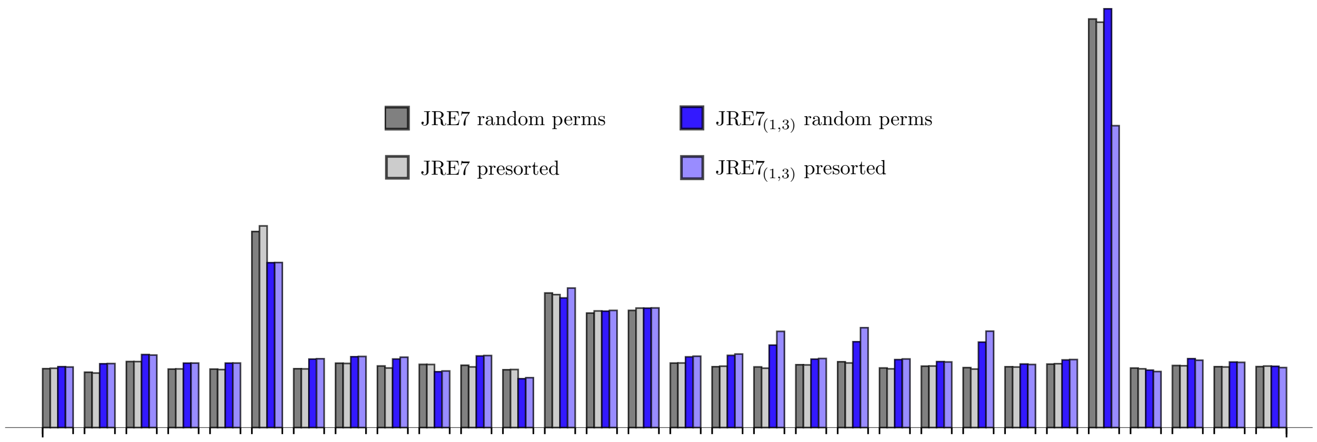

Figure 9: Relative basic block running times for all blocks that are executed $\mathcal{O}(n)$ times. The different algorithms and input distributions do not seem to influence block times much.

Furthermore, we note that without explicit warmup and without JIT profiling, JRE7 seems to outperform $\text{JRE7}_{(1,3)}$ on average, while both fixed warmup studies seem to favor the asymmetric pivot choice.

4.3 Asymptotic Running Times.

Section 5.2 describes an experimental methodology to assign each basic block its contribution to overall running time. Thereby, we can combine MaLiJAn's reliable asymptotic extrapolations of combinatorial cost measures with actual running times to an asymptotic running time extrapolation. Corresponding results are shown in Table 4.

Note that these asymptotics are more than plain extrapolations of measured running times. All extrapolating is done on block frequency counters which can be determined without noise. Only the constants these terms are multiplied with are determined by running time experiments. We have observed that this noise can indeed hide the true asymptotic behavior: The very same extrapolation heuristic that found the correct linearithmic growth from frequency counters attested linear growth for the noisy running times.

Quite surprisingly — except for the `-Xcomp` mode — all asymptotic running times favor the symmetric Quicksort implementation. This disagrees with above findings for the number of executed Bytecodes. Consequently, the running times of individual basic blocks must behave differently than the numbers of Bytecodes in the blocks.

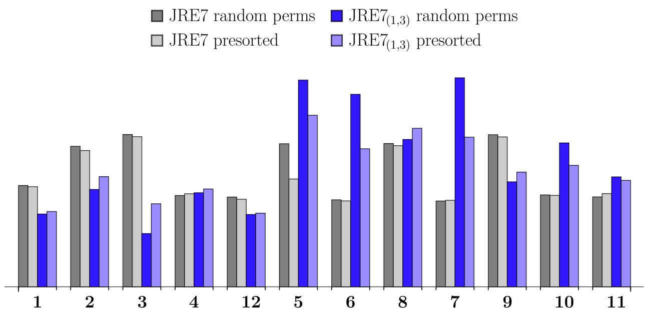

Figure 10: Relative basic block running times for all blocks that are executed $\Theta(n \log n)$ times. The numbers correspond to the blocks IDs used in Figure 1. It is clearly visible that the running time contribution of some basic blocks is heavily influenced by the pivot choice.

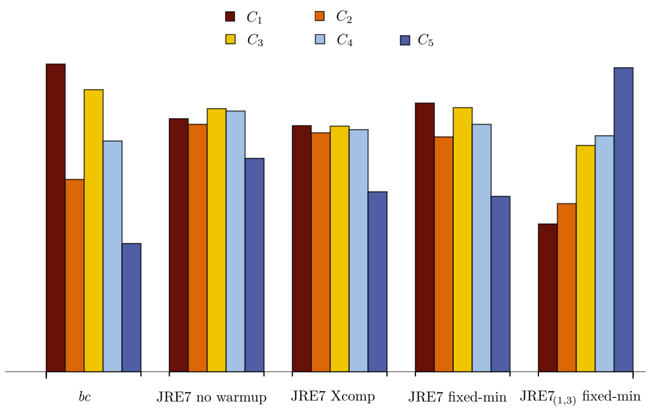

Figure 11: Relative costs for the five cycles $C_1, \ldots, C_5$ of Figure 1 introduced above in different cost measures, namely the number of Bytecodes and block running times for different algorithms and JIT modes.

4.4 Comparing Block Times. During the computation of asymptotic running times, MaLiJAn determines the running time contribution of each basic block, which allows for closer examination of before mentioned mismatch. For better comparability, we do not use the additional running time measurement described in Section 5.2. Instead, we use $c'(i) = \frac{b_i}{f_i}$ as block time estimate. Note that $c(i) = c'(i) \cdot \frac{T}{B}$ where $\frac{T}{B}$ is the average sampling interval, which will be close to constant, but is subject to noise.

Figure 9 on the preceding page shows that for most of the blocks that are executed only a linear number of times, the $c'(i)$ are essentially independent of the pivot choice. For the asymptotically dominating basic blocks — i. e. those with a linearithmic number of executions — surprisingly, the picture changes, as shown in Figure 10.

This carries over to the costs of the five cycles $C_1, \ldots, C_5$ identified in the control flow graph of the partitioning method; see Figure 11. A closer inspection of the figure explains why JRE7$_{(1,3)}$ performs worse than expected based on the number of executed Bytecodes: For JRE7 block times, C_5 is the cheapest cycle by far, whereas C_1 is rather expensive. However, the block times for JRE7$_{(1,3)}$ show exactly the opposite behavior! The short C_5 cycle — which JRE7$_{(1,3)}$ executes exceptionally often *by design* — suddenly becomes the least favorable iteration path.

This makes it plausible that an algorithmic modification improving the number of executed Bytecodes can still be inferior to another one. Even though we fail to provide an explanation for *why* this happens, at least having identified *where* in the code the difference is located might help future investigations.

5 Method

5.1 Maximum Likelihood Analysis. In this section, we briefly review the purely analytical study of algorithms and how we imitate it in our tool MaLiJAn. As we are interested in the practical efficiency of algorithms, we only consider average case analyses. The gold standard of the field is an analysis in the style of Knuth's "The Art of Computer Programming", which is based on the following assumption.

Assumption 5.1. *An instruction in the code listing of a program adds the same constant contribution to overall costs each time it is executed. In particular, the contribution does not depend on the context of execution.*

Knuth computes the expected running time of a given program w. r. t. a given input distribution by analyzing how often every single line of the code is executed on average. Assume the instructions of the program are labelled by line numbers $1, \ldots, k$. Denote by f_i the expected execution frequency of instruction i and by $c(i)$ its cost contribution. Then the total expected cost is given by

$$\sum_{i=1}^{k} c(i) \cdot f_i \,.$$

The hard part of the analysis is to determine the expected frequencies f_i. In our tool MaLiJAn, they are deduced from experiments via the maximum likelihood principle and certain extrapolation techniques. However, the corresponding technical details and correctness proofs are omitted here — a complete presentation of the theory already appeared in [10].

MaLiJAn allows studying algorithms on different levels of abstraction. If an abstract measure like counting elementary operations suffices to assess the impact of a code variation, we do not need separate experiments on different hardware environments: Elementary operation counts are platform-independent. For those effects that are not observable in the abstract model — e. g. running time due to cache misses or branch mispredictions — our methodology isolates and minimizes the part of the experiment that requires runtime measurements (see Section 5.2).

In all cases our tool does not just provide simple counts or measurements but also estimates of the asymptotic growths rates as functions in the symbolic input size n. Most notably, these growth rates are always based on combinatorial counts, which can be determined without noise. In contrast, direct extrapolation of runtime measurements inevitably includes noise and in fact, we have observed such noise to fool our tool's extrapolation heuristic on total runtimes, while the same heuristic found the correct asymptotic growth rate from combinatorial counts of the same runs.

In detail, we consider a program in Java Bytecode[2] and label its instructions with line numbers $1, \dots, k$. For a given execution of the program on some input, we call the sequence of visited line numbers the *trace* of this execution. So, a trace is formally a word over $\{1, \dots, k\}$. A given set of inputs thus induces a language over $\{1, \dots, k\}$. Similarly, a probability distribution over inputs induces a probability distribution over traces.

Traces are used to define our notion of algorithmic *cost measure*. Specifically, a cost measure c is characterized by a cost contribution $c(i) \in \{0, 1, 2, \dots\}$ for each line number $i \in \{1, \dots, k\}$. We define the cost $c(t)$ of a trace $t = t_1 \dots t_m$ by summation over its elements' costs, i. e. $c(t) = c(t_1) + \cdots + c(t_m)$. We are interested in the *expected* cost $\mathbb{E}\, c(t)$ of a trace t randomly chosen according to the input distribution. Our method inherently supports only such *additive* cost measures[3], for instance the total number of comparisons. Non-additive measures such as maximum memory usage elude us. This restriction is the formal version of Assumption 5.1 and thus also applies to Knuthian analysis.

Towards an automated approach, we note that the program's control flow graph — viewed as nondeterministic finite automaton — induces a regular overapproximation of said language of traces. Essentially, the control flow graph ignores dependencies between branch conditions. When equipped with transition probabilities $p_1, \dots, p_\ell$, the control flow graph becomes a *probabilistic* finite automaton, i. e. a Markov chain accepting words. Denote by $\mathbb{E}\, C = \mathbb{E}\, C(p_1, \dots, p_\ell)$ the expected costs of a random terminating run of this Markov chain. It can be computed exactly and symbolically in the unknown probabilities $p_1, \dots, p_\ell$ (see e. g. [9, Chapter 2]) with standard computer algebra systems. Even though the Markov chain accepts sequences that are not traces of the program, we have shown in [10] that $\mathbb{E}\, C = \mathbb{E}\, c(t)$ for suitable branch probabilities $p_1, \dots, p_\ell$.

It remains to obtain such suitable transition probabilities. The probabilities can be interpreted as the free parameters of a probability model, for which we compute estimates. To this end, we randomly sample inputs and record their actual traces. Taking relative transition frequencies from the traces indeed gives a maximum likelihood estimator for the parameters w. r. t. the given traces.

This is done separately for all observed input sizes, such that we get one transition probability estimate per size. Finally, we extend these to a *function* in n via extrapolation. In general, this is a heuristic step, and in fact the only part of the method where we sacrifice provable correctness. For many algorithms, however, the set of occurring functions is rather limited so that we can still hope for good results.[4] Moreover, MaLiJAn uses established statistics to empirically assess the quality of extrapolations. Once the probability model is trained it is treated as given. In accordance with the scientific method, we accept the inherent simplifications made while building our model and use it nevertheless to make predictions. MaLiJAn also offers basic support for validating the model.

Finally, the thus obtained transition probability functions $p_i(n)$ are inserted for the unknowns in the precise expected costs $C\big(p_1(n), \dots, p_\ell(n)\big)$. In [10, Theorem 5] it is shown that, assuming perfect extrapolation, these probabilities are indeed suitable in the above sense. We thus obtain a closed function in n for the expected costs — the same function that results from Knuthian analysis.

5.2 From Counting to Running Time. As a library designer, one is focused on the actual running time of algorithms. Therefore, the desired cost measure c assigns to every line number i the time $c(i)$ needed to execute this instruction. The seemingly simple task to determine these times $c(i)$ turns out to be quite challenging: As the running times of single instructions are in the range of few nanoseconds, direct measurement is out of the question. In [3], the authors nicely argue that determining the $c(i)$ via fitting from measured total running times and known execution frequencies f_i is not effective, either. The authors present a better method using "equivalent code fragments", but this requires manual work.

We propose a fully automatic approach called "basic block sampling". We divide the program into basic blocks, i. e. maximal blocks of sequential instructions. Then, we inject instructions at the beginning of each block to store an identifying number of the block in a global variable. This introduces a systematic error as each basic block becomes a few instructions longer, but it will be fairly small compared to other techniques of runtime measurement. Then, on a periodic basis, we concurrently read the global variable and store the block number. Note that this periodic job is done in parallel and hence does not influence the running

[2]This code may be generated from Java source by the Java compiler or from any other programming language that can be compiled to Bytecode.

[3]In essence, this is because we rely on the linearity of the expectation to split up $\mathbb{E}\, c(t)$ and that an instruction's cost is independent of its context in the trace.

[4]For example in the study in [10], our extrapolation heuristic could reproduce the expected costs of Bubblesort and Quicksort known from the literature.

time of the algorithm itself, i. e. it does not add to the systematic error. By repeating the run sufficiently often, the relative frequencies of the observed block numbers approach the relative running time contribution of the blocks (see Appendix C for a quantitative discussion).

From this, we get the vector $b = (b_1, \ldots, b_k)$ of observed block frequencies, i. e. block i has been seen b_i times in total. In separate runs, we also count f_i exactly, i.e. how often block i is executed in total, and we measure the total running time T in yet another run. Then, we use

$$c(i) := \frac{1}{f_i} \cdot \frac{b_i}{B} \cdot T, \qquad \text{where } B := \sum_{i=1}^{k} b_i$$

as an estimate of the block running times.

Note that we implicitly presumed Assumption 5.1. In practice, different running times for two executions of the same instruction can occur, e. g. because of cache misses. We will nevertheless use Assumption 5.1 and try to detect violations by testing the created runtime model at the end.

5.3 MaLiJAn. MaLiJAn is an integrated Java implementation of the method outlined in the previous section. It has a graphical user interface and uses Mathematica for symbolic calculations. Executables and further instructions can be obtained from the website `http://wwwagak.cs.uni-kl.de/malijan.html`.

6 Conclusion & Future Work

In this paper, we used our tool MaLiJAn to study the JRE7 implementation of Yaroslavskiy's dual pivot Quicksort and an asymmetric variant of it. We showed that our variant is asymptotically better w. r. t. the number of executed Java Bytecode instructions, both for random permutations and presorted arrays.

A closer look at the optimized algorithm shows that the use of asymmetric pivots changes the partitioning step in a way that favors cycles with only a small number of Bytecodes. Surprisingly, when using skewed pivots, the running time of the corresponding blocks increases heavily such that our optimization seems to fail. However, for the original benchmark used by the Java core library developers to assess the quality of optimizations, our asymmetric pivot choice achieves slight improvements for most input types, without getting significantly worse for any other input. Given the large efforts put into the highly-tuned library implementation, even these small improvements are remarkable in the authors' opinion.

Efficiency in practice heavily depends on details of the JIT compiler configuration for both investigated variants. In particular, intriguing clustering of running times was consistently observed if the JIT can use profiling information from the current input for compiling the algorithm. Further experiments have shown that different setups for the JIT warumup do not lead to such clustering.

7 Acknowledgements

We thank Florian Furbach, Michael Holzhauser and Vasil Tenev for their part in developing MaLiJAn, which started out as a students project. Furthermore, we are grateful to Mathematica Stack Exchange user kguler for showing us how to customize distribution charts.[5] Finally, we thank our anonymous referees for their helpful comments and suggestions incorporated in this final version.

References

[1] J. L. J. Bentley and M. D. McIlroy, *Engineering a sort function*, Software: Practice and Experience, 23 (1993), pp. 1249–1265.

[2] G. S. Brodal, R. Fagerberg, and G. Moruz, *On the adaptiveness of quicksort*, J. Exp. Algorithmics, 12 (2008), pp. 3.2:1–3.2:20.

[3] U. Finkler and K. Mehlhorn, *Runtime prediction of real programs on real machines*, in SODA 1997, M. E. Saks, ed., ACM/SIAM, Jan. 1997, pp. 380–389.

[4] P. Hennequin, *Analyse en moyenne d'algorithmes : tri rapide et arbres de recherche*, PhD Thesis, Ecole Politechnique, Palaiseau, 1991.

[5] Java Core Library Development Mailing List, *New portion of improvements for Dual-Pivot Quicksort.* `http://grokbase.com/p/openjdk/core-libs-dev/1056hs0qze/new-portion-of-improvements-for-dual-pivot-quicksort`, 2009.

[6] ——, *Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort.* `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`, 2009.

[7] ——, *Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quicksort: Improvements.* `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2676`, 2009.

[8] K. Kaligosi and P. Sanders, *How branch mispredictions affect quicksort*, in ESA 2006, T. Erlebach and Y. Azar, eds., Springer, 2006, pp. 780–791.

[9] R. Kemp, *Fundamentals of the Average Case Analysis of Particular Algorithms*, Wiley-Teubner, Stuttgart, 1984.

[10] U. Laube and M. E. Nebel, *Maximum likelihood analysis of algorithms and data structures*, Theoretical Computer Science, 411 (2010), pp. 188–212.

[5] Find the post in question under `http://mathematica.stackexchange.com/q/11097`.

[11] C. Martínez and S. Roura, *Optimal Sampling Strategies in Quicksort and Quickselect*, SIAM Journal on Computing, 31 (2001), p. 683.

[12] R. Sedgewick, *Quicksort*, PhD Thesis, Stanford University, 1975.

[13] ———, *The analysis of Quicksort programs*, Acta Inf., 7 (1977), pp. 327–355.

[14] R. Sedgewick and K. Wayne, *Algorithms*, Addison-Wesley, 2011.

[15] S. Wild and M. E. Nebel, *Average Case Analysis of Java 7's Dual Pivot Quicksort*, in ESA 2012, L. Epstein and P. Ferragina, eds., vol. LNCS 7501, Springer Berlin/Heidelberg, 2012, pp. 825–836.

Appendix

A Algorithms

Listing 1 shows the JRE7 implementation studied in this paper. For convenience, we also reproduce the basic variant of Yaroslavskiy's dual pivot Quicksort, see Algorithm 1 on the following page.

Listing 1: Core part of the JRE7 Quicksort implementation, namely pivot sampling and partitioning process. Special handling of equal keys has been removed for clarity; it would not be executed for the inputs considered in this paper.

```
void quicksort(int[] A, int left, int right) {
    int length = right - left + 1;
    if (length < 5) {
        ... // insertionsort for small lists
        return;
    }
    // cheap approximation for length/7 :
    int s7 = (length >> 3) + (length >> 6) + 1;
    int e3 = (left + right) >>> 1;
    int e2 = e3 - s7;
    int e1 = e2 - s7; if (length == 8) ++e1;
    int e4 = e3 + s7; int e5 = e4 + s7;
    // sort sample using insertionsort:
    sort5elements(A, e1, e2, e3, e4, e5);
    int pIndex = e2, qIndex = e4; // tertiles
    int l = left + 1, g = right - 1, k = l;
    int p = A[pIndex]; A[pIndex] = A[left];
    int q = A[qIndex]; A[qIndex] = A[right];

    while( k <= g ) {
        int ak = A[k];
        if (ak < p) {
            A[k] = A[l]; A[l] = ak; ++l;
        } else if (ak >= q) {
            while (A[g] > q && k < g) --g;
            if (A[g] < p) {
                A[k] = A[l]; A[l] = A[g]; ++l;
            } else {
                A[k] = A[g];
            }
            A[g] = ak; --g;
        }
        ++k;
    }
    --l; ++g;

    // Swap pivots into their positions
    A[left] = A[l]; A[l] = p;
    A[right] = A[g]; A[g] = q;
    quicksort(A, left, l - 1);
    quicksort(A, g + 1, right);
    quicksort(A, l,   g    );
}
```

B Experimental Setup

Java programs were compiled using javac version 1.7.0_03 from the Oracle Java Development Kit and run on the HotSpot 64-bit Server VM, version 1.7.0_03. All running time measurements were done on an Intel Core i7 920 processor with four cores with hyperthreading, running at 2.67GHz. This processor has 8MB of shared on-die L3 cache and the system has 6GB of main memory. The operating system is Ubuntu 10.10 with Linux 2.6.35-32-generic kernel. Whilst running the simulations, graphical user interface was completely disabled to have as little background services running as possible.

C Significance of Block Sampling

In this section, we have a closer look at the basic block sampling approach used to estimate the running time contributions of single basic blocks. On the test machines we used, the best achievable sampling intervals were $\approx 10\,\mu$s. This is fairly huge in comparison with the few nanoseconds a typical basic block needs to be executed. Consequently, we will miss all but $\approx 1\,‰$ of all blocks, which is pretty poor. However, assuming a deterministic algorithm, we can simply sample across m runs of the algorithm to increase the fraction of blocks we observe. But how should m be chosen?

We model the situation as follows: The execution is repeated m times and in each repetition, we write down the current block number $i \in \{1, \ldots, k\}$ at t i. i. d. uniformly chosen points in time. We can thus assume observed block numbers to be stochastically independent. Let b_i denote the number of times we observed basic block i and write $b = (b_1, \ldots, b_k)$ for the vector of all block frequencies. Further, denote by p_i the fraction of time the processor spends executing block i — summed over all occurrences of block i in the whole run. By definition $B := \sum b_i = t \cdot m$ and $\sum p_i = 1$. Then, b is *multinomially* distributed with parameters B and $p_1, \ldots, p_k$.

To assess the significance, the following basic facts on multinomial distributions are helpful:

$$\mathbb{E}\left[\frac{b_i}{B}\right] = p_i\,, \qquad \operatorname{Var}\left[\frac{b_i}{B}\right] = \frac{p_i(1-p_i)}{B}\,.$$

Now, we can use Chebychev's inequality to get

$$\Pr\left[\left|\frac{b_i}{B} - p_i\right| > \varepsilon\right] \le \frac{p_i(1-p_i)}{\varepsilon^2\, B} \le \frac{1}{4\,\varepsilon^2\, t\, m}\,.$$

This implies, that if we want to approximate all p_is to within a confidence interval of $p_i \pm \varepsilon$ at confidence level γ, we need $m \ge \big(4\varepsilon^2 t(1-\gamma)\big)^{-1}$ repetitions.

In the actual implementation, we sample in periodic distances for simplicity. Therefore, two samples from

Algorithm 1. Basic variant of Yaroslavskiy's dual pivot Quicksort given as Algorithm 3 in [15].

```
DualPivotQuicksortYaroslavskiy($A$, $left$, $right$)
    // Sort the array $A$ in index range $left, \dots, right$.
    if $right - left \geq 1$
        $p := A[left]$;    $q := A[right]$
        if $p > q$ then Swap $p$ and $q$ end if
        $\ell := left + 1$;    $g := right - 1$;    $k := \ell$
        while $k \leq g$
            if $A[k] < p$
                Swap $A[k]$ and $A[\ell]$
                $\ell := \ell + 1$
            else
                if $A[k] > q$
                    while $A[g] > q$ and $k < g$ do $g := g - 1$ end while
                    Swap $A[k]$ and $A[g]$
                    $g := g - 1$
                    if $A[k] < p$
                        Swap $A[k]$ and $A[\ell]$
                        $\ell := \ell + 1$
                    end if
                end if
            end if
            $k := k + 1$
        end while
        $\ell := \ell - 1$;    $g := g + 1$
        Swap $A[left]$ and $A[\ell]$    // Bring pivots to final position
        Swap $A[right]$ and $A[g]$
        DualPivotQuicksortYaroslavskiy($A$, $left$, $\ell - 1$)
        DualPivotQuicksortYaroslavskiy($A$, $\ell + 1$, $g - 1$)
        DualPivotQuicksortYaroslavskiy($A$, $g + 1$, $right$)
    end if
```

the same run are not guaranteed to be stochastically independent. However, as the delays are subject to random noise exceeding the typical executing time of a single basic block by several orders of magnitude, we can still expect good estimates.

For the running time predictions of Table 2, we used block sampling with $B \approx 10^5$. Assuming independence of measurements and a confidence level of $\gamma = 99\,\%$, we obtain $\varepsilon \approx 0.0158$. Therefore, we can expect the obtained relative runtime contributions of basic blocks to be accurate within $1.58\,\%$.

D Yaroslavskiy's Benchmark

Below, you see the output of one run of Yaroslavskiy's original runtime benchmark [5]. It starts by sorting the same random permutation 12 000 times as a warm-up to trigger JIT compilation. Then, 100 instances of a variety of different (random) lists of length 1 000 000 is sorted and the minimum and average running times are reported.

```
start warm up
.............
  end warm up

     random =======
                   jdk7: min 90.696726  avg 91.72415609000001
       jdk7 bytecodeopt: min 90.090356  avg 90.68792782

      equal =======
                   jdk7: min 1.718005  avg 1.78127236
       jdk7 bytecodeopt: min 1.390497  avg 1.4663646799999999

  stagger 1 =======
                   jdk7: min 7.492068  avg 7.81549246
       jdk7 bytecodeopt: min 7.369246  avg 7.75822699

  stagger 2 =======
                   jdk7: min 10.028247  avg 10.47358591
       jdk7 bytecodeopt: min 10.030905  avg 10.51836514

  stagger 4 =======
                   jdk7: min 15.416064  avg 15.95681691
       jdk7 bytecodeopt: min 15.426902  avg 15.91885334

  stagger 8 =======
                   jdk7: min 19.922822  avg 20.366305620000002
       jdk7 bytecodeopt: min 19.819549  avg 20.32786134

organ pipes =======
                   jdk7: min 7.70015  avg 8.09365574
       jdk7 bytecodeopt: min 7.710011  avg 8.15790016

  ascendant =======
                   jdk7: min 1.073704  avg 1.12440074
       jdk7 bytecodeopt: min 1.075406  avg 1.08757955

 descendant =======
                   jdk7: min 1.828119  avg 1.97187739
       jdk7 bytecodeopt: min 1.939211  avg 1.97484294

period 1..2 =======
                   jdk7: min 3.469296  avg 3.63163509
       jdk7 bytecodeopt: min 3.645568  avg 3.7014117200000003

period 1..3 =======
                   jdk7: min 3.161118  avg 3.34622231
       jdk7 bytecodeopt: min 2.982841  avg 3.05186187

period 1..4 =======
                   jdk7: min 5.478701  avg 5.67088424
       jdk7 bytecodeopt: min 5.274278  avg 5.35144238

period 1..5 =======
                   jdk7: min 4.930893  avg 5.15629265
       jdk7 bytecodeopt: min 4.415805  avg 4.52956761

period 1..6 =======
                   jdk7: min 5.810709  avg 5.92560454
       jdk7 bytecodeopt: min 5.698213  avg 5.808113059999999

period 1..7 =======
                   jdk7: min 7.736918  avg 7.93364137
       jdk7 bytecodeopt: min 7.233479  avg 7.40051437

period 1..8 =======
                   jdk7: min 5.469732  avg 5.64847769
       jdk7 bytecodeopt: min 5.158171  avg 5.36201805
```

```
  stagger 3 =======
                   jdk7: min 11.431634  avg 11.710738019999999
       jdk7 bytecodeopt: min 11.431929  avg 11.74521591

  stagger 5 =======
                   jdk7: min 15.747455  avg 16.329504189999998
       jdk7 bytecodeopt: min 15.730531  avg 16.35218499

  stagger 6 =======
                   jdk7: min 16.608467  avg 17.15020702
       jdk7 bytecodeopt: min 16.550892  avg 17.23135113

  stagger 7 =======
                   jdk7: min 17.337118  avg 17.835802920000003
       jdk7 bytecodeopt: min 17.163399  avg 17.776122670000003

random 1..2 =======
                   jdk7: min 7.781607  avg 7.946202400000001
       jdk7 bytecodeopt: min 7.232094  avg 7.35838576

random 1..3 =======
                   jdk7: min 9.024478  avg 9.149986029999999
       jdk7 bytecodeopt: min 8.455576  avg 8.577225140000001

random 1..4 =======
                   jdk7: min 10.946617  avg 11.094569589999999
       jdk7 bytecodeopt: min 10.319744  avg 10.40749047
```