Analysis of Branch Misses in Quicksort*

Conrado Martínez† Markus E. Nebel‡§ Sebastian Wild‡

November 11, 2014

Abstract

The analysis of algorithms mostly relies on counting classic elementary operations like additions, multiplications, comparisons, swaps etc. This approach is often sufficient to quantify an algorithm's efficiency. In some cases, however, features of modern processor architectures like pipelined execution and memory hierarchies have significant impact on running time and need to be taken into account to get a reliable picture. One such example is Quicksort: It has been demonstrated experimentally that under certain conditions on the hardware the classically optimal balanced choice of the pivot as median of a sample gets *harmful*. The reason lies in mispredicted branches whose rollback costs become dominating.

In this paper, we give the first precise analytical investigation of the influence of pipelining and the resulting branch mispredictions on the efficiency of (classic) Quicksort and Yaroslavskiy's dual-pivot Quicksort as implemented in Oracle's Java 7 library. For the latter it is still not fully understood why experiments prove it 10 % faster than a highly engineered implementation of a classic single-pivot version. For different branch prediction strategies, we give precise asymptotics for the expected number of branch misses caused by the aforementioned Quicksort variants when their pivots are chosen from a sample of the input. We conclude that the difference in branch misses is too small to explain the superiority of the dual-pivot algorithm.

1 Introduction

Quicksort (QS) is one of the most intensively used sorting algorithms, e.g., as the default sorting method in the standard libraries of C, C++, Java and Haskell. Classic Quicksort (CQS) uses one element of the input as *pivot* P according to which the input is *partitioned* into the elements smaller than P and the ones larger than P, which are then sorted recursively by the same procedure.

The choice of the pivot is essential for the efficiency of Quicksort. If we always use the smallest or largest element of the (sub-)array, quadratic runtime results, whereas using the median gives an (asymptotically) comparison-optimal sorting algorithm. Since the precise computation of the median is too expensive, *sampling strategies* have been invented: out of a sample of k randomly selected elements of the input, a certain *order statistic* is selected as the pivot — the so-called *median-of-three* strategy is one prominent example of this approach.

In theory, Quicksort can easily be generalized to split the input into $s \geq 2$ partitions around $s-1$ pivots. (CQS corresponds to $s = 2$). However, the implementations of Sedgewick and others did not perform as well in running time experiments as classic single-pivot Quicksort [13]; it was common belief that the overhead of using several pivots is too large in practice. In 2009, however, Vladimir Yaroslavskiy proposed a new dual-pivot variant of Quicksort which surprisingly outperformed the highly engineered classic Quicksort of Java 6, which

*Part of this research was done during a visit at UPC, for which the second and third authors acknowledge support by project TIN2007-66523 *Formal methods and algorithms for system design (FORMALISM)* of the Spanish Ministry of Economy and Competitiveness

†Department of Computer Science, Univ. Politècnica de Catalunya, Email: conrado@cs.upc.edu

‡Computer Science Department, University of Kaiserslautern, Email: {wild,nebel}@cs.uni-kl.de

§Department of Mathematics and Computer Science, University of Southern Denmark

then lead to its replacement in Java 7 by Yaroslavskiy's dual-pivot Quicksort (YQS).

An analytic explanation of the superiority of YQS was lacking at that time (and is still open to some extent). We showed that YQS indeed saves 5 % of comparisons (for random pivots); but also that it needs 80 % *more* swaps than CQS [16, 17]. Arguably, these traditional cost measures do not explain the running times well and it is likely that features of modern CPUs like *memory hierarchies* and *pipelines* are the prime reason for the algorithm's efficiency. This would be in accordance with the aforementioned results in which multi-pivot Quicksort was less efficient, as the experiments were done in a time when computers simply did not have such features.

In this paper we address the effects of pipelines, which are used to speed up execution as follows: Inside the CPU, machine instructions are split into *phases* like "fetching the instruction", "decoding and loading data", "executing the instruction", and "writing back results". Each phase takes one CPU cycle. Modern CPUs execute different phases in parallel, i.e., if there are L phases, one can execute L instructions at once, each in a different phase, resulting in a speed-up of L.

The downside of this idea, however, comes with *conditional jumps*. For those, the CPU will have to decide the outcome *before* it has actually been computed — otherwise it could not go on with the first phases of the subsequent commands. To cope with that, several *branch prediction schemes* have been invented, which try to *guess* the actual outcome. In the simplest case each branch (conditional jump) is marked "probably taken" or "probably not taken" at compile time and the CPU acts accordingly. As branch outcomes most often depend on data not known at compile time, this strategy is quite limited. In order to *adapt* predictions to actually observed behavior special hardware support is needed. For the so-called *1-bit predictor*, the CPU stores for each branch in the code whether or not it was taken the last time it was executed and assumes the same behavior for the future. In a *2-bit prediction* scheme, the CPU has to make a wrong prediction twice before switching. Such simple schemes have been used by the first CPUs with pipelining. Modern microprocessors implement more sophisticated heuristics [3], which try to recognize common patterns in branching behavior. As they seem too intricate for precise analysis and are probably *inferior* to the simple schemes in Quicksort[1] we focus on the basic predictors from above.

In case of a false prediction (*branch misprediction* or *branch miss*, briefly BM) the CPU has to *undo* all erroneously executed steps (phases) and load the correct instructions instead. A branch miss is thus a costly event. As an extreme example, Kaligosi and Sanders [7] observed on a Pentium 4 Prescott CPU (a processor with an extremely long pipeline and thus a high cost per BM) that the running time penalty of a BM is so high that a very skewed pivot choice outperformed the typically optimal median pivot, even though the latter leads to much less executed instructions in total. The effect was not reproducible, however, on the slightly different Pentium 4 Willamette [1]. Here two effects counteract: a biased pivot makes branches easier to predict but also gives unbalanced subproblem sizes. Brodal and Moruz [2] have shown that this is a general trade-off in comparison-based sorting: One can only save comparisons at the price of additional branch misses.

Differences in the number of branch misses might be an explanation for the superiority of YQS and will thus be analyzed in this paper in connection with pivot sampling. We will also reconsider branch misses in classic Quicksort by continuing the analyses of Kaligosi and Sanders [7] and Biggar et al. [1], we present precise leading terms[2] and explicitly address pivot sampling for finite sample sizes k.

We find that CQS and YQS cause roughly the same number of branch misses and hence pipelining effects are *not* a likely explanation for the superiority of YQS. Even if this result in

[1] There are no branch patterns in Quicksort (cf. Section 5.1)!

[2] By our discussion in Section 5.1, we also answer a question posed by Kaligosi and Sanders [7] in their footnote 2, where they ask for an argument to make their heuristic analysis — assuming a constant probability α for an element to be small — rigorous. We find it most appropriate to answer a footnote question also in a footnote, even though we consider the settlement of this open problem quite noteworthy.

Algorithm 1. Classic Crossing-Pointer Partitioning.

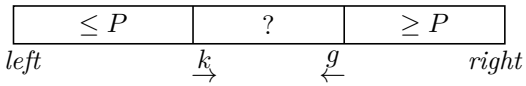
PARTITIONSEDGEWICK($A, \mathit{left}, \mathit{right}, p$)

// Assumes $\mathit{left} \le \mathit{right}$ and a sentinel $A[0] = -\infty$.
// Rearranges A such that with return value i_p holds

$$\begin{cases} \forall\ \mathit{left} \le j < i_p, & \mathtt{A}[j] \le p; \\ \forall\ i_p \le j \le \mathit{right}, & \mathtt{A}[j] \ge p. \end{cases}$$

```
1  k := left − 1;    g := right
2  do
3       do k := k + 1 while A[k] < p end while
4       do g := g − 1 while A[g] > p end while
5       if g > k then Swap A[k] and A[g] end if
6  while g > k
7  return k
```

Invariant:



Algorithm 2. Yaroslavskiy's dual-pivot partitioning.

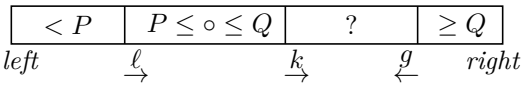
PARTITIONYAROSLAVSKIY ($\mathtt{A}, \mathit{left}, \mathit{right}, p, q$)

// Assumes $\mathit{left} \le \mathit{right}$.
// Rearranges A s. t. with return value (i_p, i_q) holds

$$\begin{cases} \forall\ \mathit{left} \le j \le i_p, & \mathtt{A}[j] < p; \\ \forall\ i_p < j < i_q, & p \le \mathtt{A}[j] \le q; \\ \forall\ i_q \le j \le \mathit{right}, & \mathtt{A}[j] \ge q. \end{cases}$$

```
1  ℓ := left;    g := right;    k := ℓ
2  while k ≤ g
3       if A[k] < p
4            Swap A[k] and A[ℓ]
5            ℓ := ℓ + 1
6       else
7            if A[k] ≥ q
8                 while A[g] > q and k < g
9                      g := g − 1
10                end while
11                if A[g] ≥ p
12                     Swap A[k] and A[g]
13                else
14                     Swap A[k] and A[g]
15                     Swap A[k] and A[ℓ]
16                     ℓ := ℓ + 1
17                end if
18                g := g − 1
19           end if
20      end if
21      k := k + 1
22 end while
23 return (ℓ − 1, g + 1)
```

Invariant:



isolation appears negative, it still entails valuable new insights: it provides further evidence for the hypothesis of Kushagra et al. [8] that it is the impact of memory hierarchies in modern computers that renders YQS faster.

2 Generalized Quicksort

In this section, we review classic Quicksort and Yaroslavskiy's dual-pivot variant and introduce the generalized pivot-sampling method.

2.1 Generalized Pivot Sampling. For the one-pivot case, our pivot selection process is declaratively specified as follows: for $\mathbf{t} = (t_1, t_2) \in \mathbb{N}^2$ a fixed parameter, choose a random sample $\mathbf{V} = (V_1, \dots, V_k)$ of size $k = k(\mathbf{t}) := t_1 + t_2 + 1$ from the input. If we denote the *sorted* sample by $V_{(1)} \le V_{(2)} \le \cdots \le V_{(k)}$, we choose the pivot $P := V_{(t_1+1)}$ such that it divides the sorted sample into regions of respective sizes t_1 and t_2:

$$\underbrace{V_{(1)} \dots V_{(t_1)}}_{t_1 \text{ elements}} \ \le\ \underbrace{V_{(t_1+1)}}_{=P} \ \le\ \underbrace{V_{(t_1+2)} \dots V_{(k)}}_{t_2 \text{ elements}}.$$

The choice $\mathbf{t} = 0$ corresponds to no sampling at all.

For dual-pivot Quicksort, we have to choose two pivots instead of just one. Therefore, with $\mathbf{t} \in \mathbb{N}^3$, we choose a sample of size $k = t_1 + t_2 + t_3 + 2$ and take $P := V_{(t_1+1)}$ and $Q := V_{(t_1+t_2+2)}$ as pivots, which divide the sorted sample into three regions of respective sizes t_1, t_2 and t_3. Note that by definition, P is the small(er) pivot and Q is the large(r) one.

2.2 Classic Crossing-Pointer Partitioning. The classic implementation of (single-pivot) Quicksort dates back to Hoare's original publication of the algorithm [6], later refined and popularized by Sedgewick [14]. Detailed code for the partitioning procedure is given in Algorithm 1. It uses two

"crossing pointers", k and g, starting at the left resp. right end of the array and moving towards each other until they meet. When a pointer reaches an element that does not belong to this pointer's partition, it stops. Once both have stopped, the out-of-order pair is exchanged and the pointers go on. The array is thereby kept invariably in form shown below the code in Algorithm 1.

2.3 Yaroslavskiy's Dual-Pivoting Method. Yaroslavskiy's partitioning method also consists of two indices, k and g, that start at the left resp. right end and scan the array until they meet. Additionally, however, a third index ℓ "lags" behind k to further divide the region left of k. Detailed pseudocode is given in Algorithm 2, where we also give the invariant maintained by the algorithm (below the code).

2.4 From Partitioning to Sorting. When partitioning is finished, k and g have met and thus for CQS, divide the array into two ranges, containing the elements smaller resp. larger than P; for YQS, ℓ and g induce *three* ranges, containing the elements that are smaller than P, between P and Q resp. larger than Q; (see also the invariants, when the "?"-area has vanished). Those regions are then sorted recursively, independently of each other.

We omit detailed pseudocode for the full sorting procedures since the only contributions to the leading term of costs come form partitioning.[3] (Recall that we consider $k = O(1)$ constant.)

To implement generalized pivot sampling, k elements from the array are chosen and the needed order statistic(s) are selected from this sample. Note that after sampling, we know for sure to which partition the sample elements belong, so we can exclude them from partitioning: the partitioning methods as they are given here are only applied to the "ordinary" elements, i.e., the $n - k$ elements that have not been part of the sample.

[3]Note, however, that some care is needed to preserve randomness in recursive calls, which is in turn crucial to set up recurrence equations. See Appendix B of [11] for further details on how to achieve this for Generalized Yaroslavskiy Quicksort.

For subarrays with at most w elements, we switch to Insertionsort (cf. [14] resp. [11]), where w is constant and at least k. The resulting algorithms, Generalized Classic resp. Yaroslavskiy Quicksort with pivot sampling parameter $\mathbf{t}$ and Insertionsort threshold w, are henceforth called $\mathrm{CQS}^w_{\mathbf{t}}$ and $\mathrm{YQS}^w_{\mathbf{t}}$.

3 Notation and Preliminaries

To unify formal notation, we denote by $s \geq 2$ the number of partitions produced in one step, i.e., $s = 2$ for CQS, $s = 3$ for YQS, and no other values for s are considered in this paper. Recall that $s-1$ pivots are needed for partitioning into s parts.

We write vectors in bold font, for example $\mathbf{t} = (t_1, \ldots, t_s)$. For concise notation, we use expressions like $\mathbf{t} + 1$ to mean *element-wise* application, i.e., $\mathbf{t}+1 = (t_1+1, \ldots, t_s+1)$. By $\mathrm{Dir}(\boldsymbol{\alpha})$, we denote a random variable with *Dirichlet distribution* and shape parameter $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d) \in \mathbb{R}^d_{>0}$. Likewise, $\mathcal{U}(a, b)$ is a random variable uniformly distributed in the interval (a, b). We use "$\overset{\mathcal{D}}{=}$" to denote equality in distribution.

As usual for the average case analysis of sorting algorithms, we assume the *random permutation model*, i.e., all elements are different and every ordering of them is equally likely. The input is given as array $\mathtt{A}$ of length n and we denote the initial entries of $\mathtt{A}$ by $U_1, \ldots, U_n$. We further assume that $U_1, \ldots, U_n$ are i. i. d. uniformly $\mathcal{U}(0, 1)$ distributed; as their ordering forms a random permutation [9], this assumption is without loss of generality.

For YQS, we call an element *small*, *medium*, or *large* if it is smaller than P, between P and Q, or larger than Q, respectively; for CQS, elements are either smaller than P or larger.

4 Generalized Quicksort Recurrence

For $\mathbf{t} \in \mathbb{N}^s$ and H_n the nth harmonic number, we define the *discrete entropy* $\mathcal{H}$ as

$$\mathcal{H} \;=\; \mathcal{H}(\mathbf{t}) \;=\; \sum_{l=1}^{s} \frac{t_l + 1}{k+1} (H_{k+1} - H_{t_l+1}) \,. \tag{4.1}$$

In the limit $k \to \infty$, such that the ratios t_l/k converge to constants τ_l, $\mathcal{H}(\mathbf{t})$ coincides with the

entropy function $\mathcal{H}^*$ of information theory:

$$\mathcal{H}(\mathbf{t}) \sim -\sum_{l=1}^{s} \tau_l(\ln(t_l+1)-\ln(k+1)) \tag{4.2}$$

$$\sim -\sum_{l=1}^{s} \tau_l \ln(\tau_l) =: \mathcal{H}^*(\boldsymbol{\tau})\,. \tag{4.3}$$

The first step follows from the asymptotic equivalence $H_n \sim \ln(n)$ as $n \to \infty$.

THEOREM 4.1. (QUICKSORT RECURRENCE)
The total expected costs for sorting a random permutation with Quicksort using a partitioning method that incurs expected costs $\mathbb{E}[T_n]$ *of the form* $\mathbb{E}[T_n] = an + O(1)$ *to produce* s *partitions and whose* $s-1$ *pivots are chosen by generalized pivot sampling with parameter* $\mathbf{t} \in \mathbb{N}^s$ *are asymptotically* $\sim \frac{a}{\mathcal{H}}\, n \ln n$*, where* $\mathcal{H}$ *is given by* (4.1).

Theorem 4.1 has first been proven by Hennequin [5, Proposition III.9] using arguments on the Cauchy-Euler differential equations for the corresponding generating function of costs. A more concise and elementary proof using Roura's *Continuous Master Theorem* [12] is given in the appendix of [11].

5 Branch Mispredictions

Thanks to Theorem 4.1 we can easily make the transition from expected *partitioning* costs to the corresponding *overall* costs, so in the following, we can focus on the first partitioning step only.

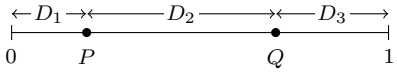
For our Quicksort variants, there are two different types of branches: those which correspond to a key comparison and all others (loop headers etc.). It turns out that all non-comparison branches are highly predictable, meaning that any reasonable prediction scheme will only incur a constant number of mispredictions (per partitioning step):

In CQS we have one backward branch at the end of the outer loop with condition "$g > k$" (line 6 of Algorithm 1) and an if-statement with the same condition (line 5), both are violated exactly once, then we exit. Similar in YQS, there is an outer loop branch with "$k \le g$" (line 2 in Algorithm 2) and an inner one with "$k < g$" (line 8), which again fail at most once. For the leading term of the number of branch misses (BM), we can therefore focus on the comparison-based branches (two in CQS and four in YQS).

All prediction schemes analyzed in this paper are *local* in the sense that for any branch instruction in the code, the prediction of its next outcome only depends on formerly observed behavior of this very branch instruction. As a consequence, predictions of one branch instruction in the code are *independent* of the history of any *other* branch instruction. Note however that the histories themselves might be highly inter-dependent in general, which complicates the analysis of branch misses.

5.1 Dirichlet Vectors and Conditional Independence.

Recall that our input consists of i. i. d. $\mathcal{U}(0,1)$ variables. If, for CQS, we *condition* on the pivot value, i.e., consider P fixed, an ordinary element U is small, if $U \in (0,P)$, and large if $U \in (P,1)$. If we call the (random) lengths of these two intervals $\mathbf{D} = (D_1, D_2) = (P, 1-P)$, then D_1 and D_2 are the *probabilities* for an element to be small resp. large. For YQS, we condition on both P and Q and correspondingly get $\mathbf{D} = (D_1, D_2, D_3) = (P, Q-P, 1-Q)$ as the probabilities for small, medium resp. large elements:



The random variable $\mathbf{D} \in [0,1]^s$ is a vector of *spacings* induced by order statistics from a sample of $\mathcal{U}(0,1)$ variables in the unit interval, which is known to have a *Dirichlet* $\mathrm{Dir}(\mathbf{t}+1)$ distribution (see Appendix C).

The vital observation for our analysis is that the probabilities $\mathbf{D}$ of the class of an element U are *independent* of all other (ordinary) elements! For the comparisons during partitioning, this implies that their *outcomes* are i. i. d.; precisely speaking: *the sequence of (binary) random variables that correspond to the outcomes of all executions (in one partitioning step) of the key comparison at one comparison location are i. i. d.,* and their distribution only depends on the pivot *values* (via $\mathbf{D}$). The outcome of a comparison does neither depend on the position in the array, nor on the number of, say, small elements that we have already seen.

As the outcomes for different comparison locations are always independent, so are the corresponding branch histories and predictions. Conditioning on $\mathbf{D}$, we can therefore count the branch misses separately for all comparison locations.

5.2 Probabilities of Branches in CQS. CQS has two comparison-based branch locations: $C^{(c1)}$ and $C^{(c2)}$ in the two inner loops (lines 3 and 4 in Algorithm 1). The first one jumps back to the loop header if $\mathtt{A}[k] < P$, the second one if $\mathtt{A}[g] > P$. Conditional on $\mathbf{D} = (D_1, D_2)$, the two branches are executed $C_n^{(c1)} = D_1 n + O(1)$ resp. $C_n^{(c2)} = D_2 n + O(1)$ times in expectation (in the first partitioning step) and each time, they are taken i. i. d. with probability $\mathbb{P}_{\text{taken}}^{(c1)} = D_1$ resp. $\mathbb{P}_{\text{taken}}^{(c2)} = D_2$, (the probabilities for the element to be small resp. large).

Note that CQS is symmetric: At both $C^{(c1)}$ and $C^{(c2)}$, we compare an element with P, so the two branches behave the same w. r. t. branch misses. It is then convenient to work with the *combined* (virtual) comparison location $C^{(c)}$ which is executed $C_n^{(c)} = C_n^{(c1)} + C_n^{(c2)} = n + O(1)$ times with branch probability $\mathbb{P}_{\text{taken}}^{(c)} = D_1$. (Such tricks will fail for the asymmetric YQS.) Denoting by $\mathbb{P}_{\text{BM}}^{(c)}$ the probability for a BM at $C^{(c)}$, the expected number of BMs in one partitioning step of CQS is then simply

$$\mathbb{E}[T_{BM}(n)] \;=\; \mathbb{E}[\mathbb{P}_{\text{BM}}^{(c)}]n + O(1)\,. \tag{5.4}$$

5.3 Probabilities of Branches in YQS. For YQS, we have four comparison locations: $C^{(y1)}$ (line 3 in Algorithm 2), $C^{(y2)}$ (line 7), $C^{(y3)}$ (line 8) and $C^{(y4)}$ (line 11). Table 1 lists their execution frequencies. Note that $C^{(y2)}$ is only reached for elements where $C^{(y1)}$ determined that they are not small, so at $C^{(y2)}$, we already know the element is either medium or large. Similarly, $C^{(y4)}$ only handles elements that $C^{(y3)}$ proved to be non-large. Recalling that the probability of an (ordinary) element to be small, medium or large is D_1, D_2 and D_3, respectively, a look at the comparisons yields

$$\mathbb{P}_{\text{taken}}^{(y1)} \;=\; D_2 + D_3\,, \quad \mathbb{P}_{\text{taken}}^{(y2)} \;=\; \frac{D_2}{D_2 + D_3}\,, \tag{5.5}$$

Location	Expectation conditional on $\mathbf{D}$	
$C_n^{(c1)}$	D_1	$\cdot\,(n + O(1))$
$C_n^{(c2)}$	D_2	$\cdot\,(n + O(1))$
$C_n^{(y1)}$	$(D_1 + D_2)$	$\cdot\,(n + O(1))$
$C_n^{(y2)}$	$(D_1 + D_2)(D_2 + D_3)$	$\cdot\,(n + O(1))$
$C_n^{(y3)}$	D_3	$\cdot\,(n + O(1))$
$C_n^{(y4)}$	$D_3(D_1 + D_2)$	$\cdot\,(n + O(1))$

Table 1: The expected execution frequencies of the comparisons locations in CQS and YQS, conditional on $\mathbf{D}$. The full distributions of the frequencies are given in [11], the conditional expectations are then easily computed using the lemmas in Appendix C of [11].

$$\mathbb{P}_{\text{taken}}^{(y3)} \;=\; D_1 + D_2\,, \quad \mathbb{P}_{\text{taken}}^{(y4)} \;=\; \frac{D_1}{D_1 + D_2}\,. \tag{5.6}$$

There are no symmetries to exploit in YQS (all comparisons are of different type), so we will get different BM probabilities $\mathbb{P}_{\text{BM}}^{(y1)}, \ldots, \mathbb{P}_{\text{BM}}^{(y4)}$ for all locations. The expected number of BM in one partitioning step is

$$\mathbb{E}[T_{BM}(n)] \;=\; \sum_{l=1}^{4} \mathbb{E}\Big[C_n^{(yl)} \cdot \mathbb{P}_{\text{BM}}^{(yl)}\Big]\,. \tag{5.7}$$

5.4 Branch Prediction Schemes. For each comparison location $C^{(i)}$ in the code, we determined the probability $\mathbb{P}_{\text{taken}}^{(i)} = \mathbb{P}_{\text{taken}}^{(i)}(\mathbf{d})$ that, conditional on $\{\mathbf{D} = \mathbf{d}\}$, this branch will be taken. The optimal strategy would then be to predict this branch to be taken iff $\mathbb{P}_{\text{taken}}^{(i)}(\mathbf{D}) \geq \frac{1}{2}$ — which, of course, is not possible since we do not know $\mathbf{D}$ at runtime. (This theoretical scheme has been considered by Kaligosi and Sanders [7] as benchmark.)

Actual adaptive prediction schemes try to *estimate* $\mathbb{P}_{\text{taken}}^{(i)}$ using a limited history of previous outcomes and base predictions on their current estimate. As the CPU has to keep track of this history for many branches, typical memory sizes are as low as 1 or 2 bits. However, in practice it is impossible to reserve even just a single bit of branch history for every possible branch-instruction location. Therefore actual hardware prediction units use *hash tables* of history storage, which means that all branch instructions whose addresses hash

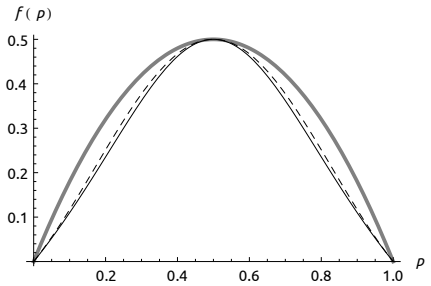


Figure 1: Comparison of the steady-state miss-rate functions for the 1-bit (thick gray), the 2-bit saturating-counter (black) and the 2-bit flip-on-consecutive (dashed) predictors. The x-axis varies p from 0 to 1. All functions are symmetric around $1/2$ — where they have a peak with value $1/2$ — and then drop to 0 as p approaches 0 or 1. Note that the 2-bit saturating counter leads to the best predictions for any p (as long as all the branch executions are i. i. d. and taken with probability p).

to the same value will share one history storage. The resulting *aliasing effects* have typically small, but rather chaotic influences on predictions in practice. We ignore those to keep analysis tractable.

The behavior of an (idealized) local adaptive prediction scheme (for a single branch instruction) forms a *Markov chain* over the states of its finite memory of past behavior. Each state corresponds to a prediction ("taken" / "not taken") and has two successor states depending on the actual outcome.

As the involved Markov chains have only 2 or 4 states, they approach their stationary distributions very quickly.[4] For the asymptotic number of branch misses for partitioning a large array, we can therefore assume the predictor automaton to have reached its steady state. Averaging over the states and the outcomes of the next branch, we get the (expected) BM probability: $\mathbb{P}^{(i)}_{\mathrm{BM}} = f(\mathbb{P}^{(i)}_{\mathrm{taken}})$ for a (deterministic) function $f : [0,1] \to [0,1]$ depending on the prediction scheme that we call its *steady-state miss-rate function*. (Biggar et al. [1] call $p \mapsto 1 - f(p)$ the *steady-state predictability*.) The schemes considered herein only differ in the topology of the Markov chain and thus in $f(p)$.

[4]Numeric experiments show that after 100 iterations the dependence of the state distribution on the initial state is less than 10^{-6}.

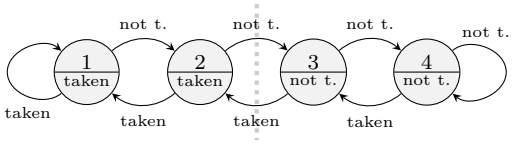


Figure 2: 2-bit saturating-counter predictor.

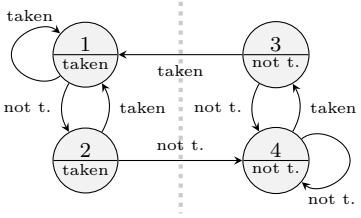


Figure 3: 2-bit flip-on-consecutive predictor.

1-Bit Predictor. The 1-bit predictor is arguably the simplest possible adaptive prediction scheme: it always predicts a branch to behave the same way as the last time it was executed. We thus encounter a branch miss for a comparison location $C^{(i)}$ if and only if two subsequent executions happen on elements with different comparison results. As the branch probabilities for two executions of the same comparison location in one partitioning step are the same and branching is independent, we get

$$f_{\text{1-bit}}(p) \;=\; 2p(1-p)\;. \tag{5.8}$$

2-Bit Saturating-Counter Predictor. The very limited memory of the 1-bit predictor fails on the frequent pattern of a loop branch that is taken most of the time and only once in a while used to leave the loop. In this situation, it would be much better to treat the loop exit as an outlier that does not change the prediction. 2-bit predictors can achieve that. They use 2 bits per branch instruction to encode one of four states of a finite automaton. The prediction is then made based on this current state and after the branch has executed, the state is updated according to the transition matrix of the automaton.

There are two variants of 2-bit predictors described in the literature, that use slightly different automata and thus give slightly different results. The first one is the 2-bit saturating counter (2-bit sc) shown in Figure 2, which is used by Brodal and Moruz [2] and Biggar et al. [1]. The second one is 2-bit flip-on-consecutive predictor (2-bit fc)

described by Kaligosi and Sanders [7]; see below. As both predictors are sensible choices, we analyze both. Moreover, it is interesting to see the difference between the two; see Figure 1 for that, as well.

To derive the steady-state miss-rate function, we translate the automaton shown in Figure 2 to a Markov chain, compute its steady-state distribution and from that the expected misprediction rate. Details are given in Appendix B. The resulting function is

$$f_{\text{2-bit sc}}(p) \;=\; \frac{p(1-p)}{1-2p(1-p)}\,. \tag{5.9}$$

2-bit Flip-On-Consecutive Predictor. The second 2-bit variant flips its prediction only after two consecutive mispredictions and we thus call it "2-bit flip-on-consecutive predictor" (see Figure 3). It is analyzed in the very same manner as 2-bit sc, details are again given in Appendix B, where we find

$$f_{\text{2-bit fc}}(p) \;=\; \frac{2p^2(1-p)^2+p(1-p)}{1-p(1-p)}\,. \tag{5.10}$$

5.5 Results. Finally, we are in the position to put everything together. As the involved constants become rather large, we need to introduce some shorthand notation: We write $\boldsymbol{\alpha} = \mathbf{t}+1 = (t_1+1,\ldots,t_s+1)$ and $\kappa = k+1 = \alpha_1+\cdots+\alpha_s$. Moreover, $x^{\overline{n}}$ denotes the nth rising factorial power of x and by $\gamma^{(c)}_{a,b}$ we denote the integral

$$\gamma^{(c)}_{a,b} \;=\; \frac{1}{\mathrm{B}(a,b)}\int_0^1 \frac{x^a(1-x)^b}{\frac{1}{c}-x(1-x)}\,dx\,. \tag{5.11}$$

where $\mathrm{B}(a,b)$ is the Beta function (see Appendix C). We will only need the cases $c=1$ and $c=2$, for which we have explicit, but unwieldy expressions (see Figure 4 on page 9).

Theorem 5.1. (Main Result)
Let $\mathbb{E}[BM_n^{\mathrm{CQS}}]$ and $\mathbb{E}[BM_n^{\mathrm{YQS}}]$ be the expected number of branch misses incurred when sorting a random permutation of length n with classic resp. Yaroslavskiy's Quicksort under generalized pivot sampling with parameter $\mathbf{t} = \boldsymbol{\alpha}-1 \in \mathbb{N}^s$. Then

$$\mathbb{E}[BM_n^{\mathrm{CQS}}] \;\sim\; \frac{a_{\mathrm{CQS}}}{\mathcal{H}}\,n\ln n \text{ and}$$
$$\mathbb{E}[BM_n^{\mathrm{YQS}}] \;\sim\; \frac{a_{\mathrm{YQS}}}{\mathcal{H}}\,n\ln n\,,$$

with $\mathcal{H}=\mathcal{H}(\mathbf{t})$ given in (4.1) and the constants

$$\begin{aligned} a_{\mathrm{CQS}} &= g_{\alpha_1,\alpha_2} \text{ and}\\ a_{\mathrm{YQS}} &= \left(\tfrac{\alpha_1}{\kappa}g_{\alpha_1+1,\alpha_2+\alpha_3}+\tfrac{\alpha_2}{\kappa}g_{\alpha_1,\alpha_2+\alpha_3+1}\right)\\ &\quad+\left(\tfrac{\alpha_1\alpha_2}{\kappa^{\overline{2}}}g_{\alpha_2+1,\alpha_3}+\tfrac{\alpha_1\alpha_3}{\kappa^{\overline{2}}}g_{\alpha_2,\alpha_3+1}\right.\\ &\qquad\left.+\tfrac{\alpha_2^{\overline{2}}}{\kappa^{\overline{2}}}g_{\alpha_2+2,\alpha_3}+\tfrac{\alpha_2\alpha_3}{\kappa^{\overline{2}}}g_{\alpha_2+1,\alpha_3+1}\right)\\ &\quad+\left(\tfrac{\alpha_3}{\kappa}g_{\alpha_1+\alpha_2,\alpha_3+1}\right)\\ &\quad+\left(\tfrac{\alpha_1\alpha_3}{\kappa^{\overline{2}}}g_{\alpha_1+1,\alpha_2}+\tfrac{\alpha_2\alpha_3}{\kappa^{\overline{2}}}g_{\alpha_1,\alpha_2+1}\right) \end{aligned}$$

where $g_{x,y}$ depends on the prediction scheme:

(i) 1-bit: $g_{x,y} = 2xy/(x+y)^{\overline{2}}$,

(ii) 2-bit sc: $g_{x,y} = \frac{1}{2}\gamma^{(2)}_{x,y}$,

(iii) 2-bit fc: $g_{x,y} = \frac{2xy}{(x+y)^{\overline{2}}}\gamma^{(1)}_{x+1,y+1}+\gamma^{(1)}_{x,y}$.

In the limit $k\to\infty$ s. t. $\mathbf{t}/k \to \boldsymbol{\tau}\in[0,1]^s$, we find $\mathbb{E}[BM_n]\sim\frac{a^*}{\mathcal{H}^*}\,n\ln n$ for $\mathcal{H}^*$ defined in (4.2) and an algorithm-dependent constant a^*:

$$\begin{aligned} a^*_{\mathrm{CQS}} &= f(\tau_1) \text{ and}\\ a^*_{\mathrm{YQS}} &= (\tau_1+\tau_2)\cdot f(\tau_2+\tau_3)\\ &\quad+(\tau_1+\tau_2)(\tau_2+\tau_3)\cdot f(\tau_2)\\ &\quad+\tau_3\cdot f(\tau_1+\tau_2)\\ &\quad+\tau_3(\tau_1+\tau_2)\cdot f(\tau_1), \end{aligned}$$

where f is the steady-state miss-rate function of the prediction scheme, see (5.8), (5.9) resp. (5.10).

Proof. We plug the different steady-state miss rate functions into (5.4) resp. (5.7) and apply Theorem 4.1. What remains is to compute the leading term constants, i.e., $\mathbb{E}[C_n^{(i)}f(\mathbb{P}^{(i)}_{\mathrm{taken}})]$ for all comparison locations. Table 1 (page 6) gives the expected execution frequencies $C_n^{(i)}$ conditional on $\mathbf{D}$ and the branch taken probabilities are $\mathbb{P}^{(c)}_{\mathrm{taken}} = D_1$ for CQS and as given in (5.5) for YQS.

Using the properties of the Dirichlet distribution collected in Appendix C, we can rewrite the involved expectations/integrals until we can either evaluate them explicitly (as is the case for 1-bit) or express them in terms of $\gamma^{(c)}_{a,b}$. Full detail computations are given in Appendix D.

For the $k\to\infty$ part, one might compute the limit of the above terms; however, there is a simpler direct argument: For $k\to\infty$ s. t. $\mathbf{t}/k\to$

$$\int_0^1 \frac{x^a(1-x)^b}{1-x(1-x)}\,dx \;=\; -\sum_{i=0}^{b-1} \mathrm{B}(a-i,b-i) \;+\; \sum_{i=1}^{\lfloor\frac{a-b}{3}\rfloor} (-1)^{i-1}\left(\tfrac{1}{(a-b)-3i+2}+\tfrac{1}{(a-b)-3i+1}\right) \;+\; \rho_1(a-b). \qquad (a \geq b)$$

$$\int_0^1 \frac{x^a(1-x)^b}{\frac{1}{2}-x(1-x)}\,dx \;=\; -\sum_{i=0}^{b-1} 2^{-i}\mathrm{B}(a-i,b-i) \;+\; 2^{-b}\sum_{i=1}^{\lfloor\frac{a-b}{4}\rfloor} (-\tfrac{1}{4})^{i-1}\left(\tfrac{1}{(a-b)-4i+3}+\tfrac{1}{(a-b)-4i+2}+\tfrac{1/2}{(a-b)-4i+1}\right) \;+\; 2^{-b}\rho_2(a-b).$$

$$\rho_1(d) \;=\; (-1)^{\lfloor\frac{d}{3}\rfloor}\begin{cases} \frac{2\pi}{3\sqrt{3}} & \text{if } d\equiv 0 \pmod 3 \\ \frac{\pi}{3\sqrt{3}} & \text{if } d\equiv 1 \pmod 3 \\ 1-\frac{\pi}{3\sqrt{3}} & \text{if } d\equiv 2 \pmod 3 \end{cases}, \qquad \rho_2(d) \;=\; \left(-\tfrac{1}{4}\right)^{\lfloor\frac{d}{4}\rfloor}\begin{cases} \pi & \text{if } d\equiv 0 \pmod 4 \\ \pi/2 & \text{if } d\equiv 1 \pmod 4 \\ 1 & \text{if } d\equiv 2 \pmod 4 \\ \frac{3}{2}-\frac{\pi}{4} & \text{if } d\equiv 3 \pmod 4 \end{cases}.$$

Figure 4: Explicit expressions for the integrals involved in $\gamma^{(1)}_{a,b}$ and $\gamma^{(2)}_{a,b}$. The formulas are only valid for $a \geq b$, but since the integrals are symmetric, one can simply use $a' = \max\{a,b\}$ and $b' = \min\{a,b\}$. The proof consists in finding recurrences for the polynomial long division of the integrand, solving these recurrences and integrating them summand by summand. Details are given in Appendix C.

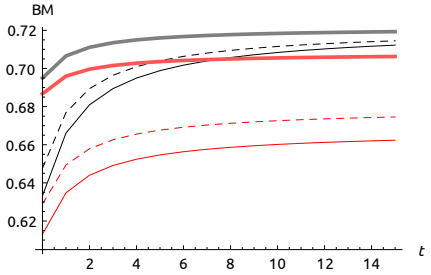


Figure 5: Branch mispredictions, as a function of t, in CQS (black) and YQS (red) with 1-bit branch prediction (fat), 2-bit saturating counter (thin solid) and 2-bit flip-consecutive (dashed) using symmetric sampling: $\mathbf{t}_{\mathrm{CQS}} = (3t+2, 3t+2)$ and $\mathbf{t}_{\mathrm{YQS}} = (2t+1, 2t+1, 2t+1)$

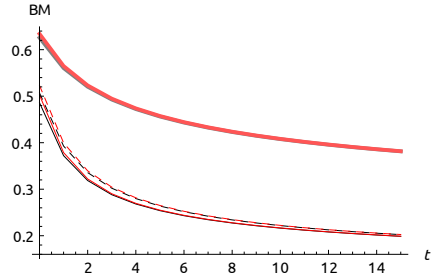


Figure 6: Branch mispredictions, as a function of t, in CQS (black) and YQS (red) with 1-bit (fat), 2-bit sc (thin solid) and 2-bit fc (dashed) predictors, using extremely skewed sampling: $\mathbf{t}_{\mathrm{CQS}} = (0, 6t+4)$ and $\mathbf{t}_{\mathrm{YQS}} = (0, 6t+3, 0)$

$\boldsymbol{\tau}$ the $\mathrm{Dir}(\mathbf{t}+1)$ distribution degenerates to a deterministic vector, i.e., $\mathbf{D} \to \boldsymbol{\tau}$ in probability. By the *continuous mapping theorem*, we also have the limit (in probability) $f(D_1) \to f(\tau_1)$ and thus $\mathbb{E}[f(D_1)] \to f(\tau_1)$. □

6 Discussion

Table 2 (page 10) summarizes the leading factor (the constant in front of $n \ln n$) in the total expected number of branch mispredictions for both CQS and YQS under the various branch prediction schemes and different pivot sampling strategies.

In practice, classic Quicksort implementations typically use median-of-3 sampling, while in Oracle's YQS from Java 7 the chosen pivots are the second and the fourth in a sample of 5 (tertiles-of-5). With 1-bit prediction, this results in approximately $0.6857 n \ln n$ vs. $0.6867 n \ln n$ BMs in the asymptotic average; for the other branch prediction strategies the difference is similar. It is very unlikely that the substantial differences in running times between CQS and YQS are caused by this tiny difference in the number of branch misses.

6.1 BM-Optimal Sampling.

Figure 5 shows the leading factor of BMs as a function of t, where pivots are chosen equidistantly from samples of size $k = 6t+5$, i.e., in CQS we use the median as pivot, in YQS the tertiles. Notice that, contrary to many other performance measures, sampling can be harmful with respect to branch mispredictions. In particular, notice that with symmetric sampling (i.e., median-of-$(2t+1)$ for CQS, tertiles-of-$(3t+2)$ for YQS) the expected number of BMs *increases*

		Classic Quicksort ($\mathrm{CQS}^w_{\mathbf{t}}$)		Yaroslavskiy ($\mathrm{YQS}^w_{\mathbf{t}}$)	
	1-bit	$2/3$	$= 0.\overline{6}$	$101/50$	$= 0.67\overline{3}$
no sampling	2-bit sc	$\pi/2 - 1$	≈ 0.57080	$31\pi/40 - 37/20$	≈ 0.58473
	2-bit fc	$4\pi/\sqrt{3} - 20/3$	≈ 0.58853	$49\pi/5\sqrt{3} - 1288/75$	≈ 0.60190
$k = 5$,	1-bit	$180/259$	≈ 0.69498	$274/399$	≈ 0.68671
$\mathbf{t}_{\mathrm{CQS}} = (2, 2)$,	2-bit sc	$225\pi/74 - 330/37$	≈ 0.63322	$785\pi/532 - 535/133$	≈ 0.61306
$\mathbf{t}_{\mathrm{YQS}} = (1, 1, 1)$	2-bit fc	$1200\pi\sqrt{3}/37 - 45540/259$	≈ 0.64766	$1280\pi\sqrt{3}/133 - 20644/399$	≈ 0.62899
$k = 5$,	1-bit	$600/959$	≈ 0.62565	$4070/6419$	≈ 0.63406
$\mathbf{t}_{\mathrm{CQS}} = (4, 0)$,	2-bit sc	$420/137 - 225\pi/274$	≈ 0.48592	$3135/917 - 3405\pi/3668$	≈ 0.50242
$\mathbf{t}_{\mathrm{YQS}} = (0, 3, 0)$	2-bit fc	$23340/959 - 600\pi\sqrt{3}/137$	≈ 0.50691	$335500/6419 - 8720\pi\sqrt{3}/917$	≈ 0.52299
$k \to \infty$,	1-bit	$1/2 \ln 2$	≈ 0.72135	$7/9 \ln 3$	≈ 0.70796
$\boldsymbol{\tau}_{\mathrm{CQS}} = (1/2, 1/2)$,	2-bit sc	$1/2 \ln 2$	≈ 0.72135	$11/15 \ln 3$	≈ 0.66751
$\boldsymbol{\tau}_{\mathrm{YQS}} = (1/3, 1/3, 1/3)$	2-bit fc	$1/2 \ln 2$	≈ 0.72135	$47/63 \ln 3$	≈ 0.67907
$k \to \infty$,	1-bit	*(large, but explicit term)*	≈ 0.55370	*(large, but explicit term)*	≈ 0.55987
$\boldsymbol{\tau}_{\mathrm{CQS}} = (1/10, 9/10)$	2-bit sc	*(large, but explicit term)*	≈ 0.33762	*(large, but explicit term)*	≈ 0.34509
$\boldsymbol{\tau}_{\mathrm{YQS}} = (1/10, 8/10, 1/10)$	2-bit fc	*(large, but explicit term)*	≈ 0.35900	*(large, but explicit term)*	≈ 0.36746

Table 2: Coefficient of the leading term of BMs in $\mathrm{CQS}^w_{\mathbf{t}}$ and $\mathrm{YQS}^w_{\mathbf{t}}$ for various combinations of branch prediction schemes and sampling parameters.

(and approaches a limit) as the size of the sample grows.

We can weaken this undesirable effect by choosing *skewed* pivots. Figure 6 shows the same sample sizes as Figure 5, with the $\mathbf{t}$ that gives the minimal number of BM for this sample size, which is to choose the extreme order statistics. Not surprisingly, CQS and YQS behave almost identically when we use such highly skewed sampling parameters. It is worth mentioning, though, that for $\mathrm{YQS}^w_{\mathbf{t}}$, $\mathbf{t} = (0, 6t + 3, 0)$ is better than $\mathbf{t} = (6t + 3, 0, 0)$ or $\mathbf{t} = (0, 0, 6t + 3)$, as far as BMs are concerned. Of course, such skewed $\mathbf{t}$ seriously penalize other performance measures such as comparisons, cache misses, etc. as unbalanced partitions become more likely.

In the limiting situation of very large samples, i.e., for $k \to \infty$ with $\mathbf{t}/k \to \boldsymbol{\tau} \in [0, 1]^s$, the leading coefficient of the expected number of BMs tends to 0 if we pick the smallest or the largest element in the sample, i.e., $\boldsymbol{\tau}_{\mathrm{CQS}} = (0, 1)$ or $\boldsymbol{\tau}_{\mathrm{CQS}} = (1, 0)$ in CQS. The same happens in YQS if we pick extremal pivots, i.e., if we take $\boldsymbol{\tau}_{\mathrm{YQS}} = (0, 0, 1)$ (the two smallest elements), $\boldsymbol{\tau}_{\mathrm{YQS}} = (0, 1, 0)$ (the smallest and the largest), or $\boldsymbol{\tau}_{\mathrm{YQS}} = (1, 0, 0)$ (the two largest elements). This limiting behavior is independent of the branch prediction strategy, but the trend doesn't show up unless the sample size is unrealistically large. Due to its asymmetries, the convergence to the limit in YQS is not equal for the three choices of $\boldsymbol{\tau}$.

6.2 Overall Costs for CQS. As the use of very skewed pivots severely runs against other important cost measures (including comparisons and cache misses), we need to consider the *combined* cost of several measures to determine choices for $\mathbf{t}$ with good practical performance. As a simplified model (which still exhibits the fundamental features of the problem) we shall consider the linear combination of low-level machine instructions (here: Bytecodes (BC)) and branch misses:

$$Q \;=\; BC \,+\, \xi \cdot BM.$$

The relative cost ξ of one BM (in terms of BCs) depends on the machine. As a mispredicted branch always entails a complete stall of the pipeline, we need at least $\xi \geq L$ (the number of stages) to recover full speed. Rollback of erroneously executed instructions might require another L cycles of additional work, so $L \leq \xi \leq 2L$ seems a reasonable range.

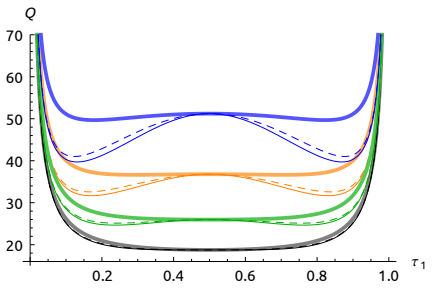


Figure 7: The function $q_\xi(\boldsymbol{\tau})$ for CQS with sample size $k \to \infty$ and $\boldsymbol{\tau} = (\tau_1, 1-\tau_1)$, for 1-bit (fat lines), 2-bit sc (thin solid) and 2-bit fc (dashed). For each branch prediction strategy, we give four plots corresponding to $\xi = 5$ (black), $\xi = 15$ (green), $\xi = 30$ (orange) and $\xi = 50$ (blue).

As long as we are only interested in the main order term of Q, we only need the main order term of BC. For $\mathrm{CQS}^w_{\mathbf{t}}$ with $\mathbf{t} = (t_1, t_2)$ we have [10, 15]

$$\mathbb{E}[T_{BC}(n)] \;=\; \left(6 + 18\frac{(t_1+1)(t_2+1)}{(k+1)(k+2)}\right)\cdot n \;+\; O(1)\,.$$

We are only interested in the coefficient $q_\xi(\mathbf{t})$ of the leading term of $\mathbb{E}[T_Q(n)]$ which depends on ξ and the sampling parameter $\mathbf{t}$.

Again, we consider the limit $k \to \infty$ to reduce the parameter space and write

$$q_\xi(\boldsymbol{\tau}) \;=\; \lim_{\substack{k\to\infty:\\ \mathbf{t}/k\to\boldsymbol{\tau}}} q_\xi(\mathbf{t}).$$

Figure 7 shows the value of $q_\xi(\boldsymbol{\tau})$ for several values of ξ and the three different branch prediction strategies. For $\xi = 5$, the three prediction schemes yield almost identical results as the weight of BMs is rather small. For larger values of ξ, the shape of the function $q_\xi(\boldsymbol{\tau})$ changes and differences between the prediction schemes become more pronounced. The two 2-bit strategies do not differ very significantly (2-bit sc is always slightly better), but both perform much better than 1-bit prediction.

6.3 Total-Costs-Optimal Sampling in CQS.

It is natural to ask for choices of $\mathbf{t}$ (and implicitly $k = k(\mathbf{t})$) that minimize $q_\xi(\mathbf{t})$, for a given value of ξ. In the limit of $k \to \infty$, we look for optimal $\boldsymbol{\tau}^*$.

It is well-known that if $\xi = 0$ (the contribution of BMs to $q_\xi(\mathbf{t})$ is disregarded) for finite-size samples the best choice for the pivot is the median of the sample, thus for any k,

$$\mathbf{t}^*(0,k) = (\lfloor \tfrac{k-1}{2} \rfloor, \lceil \tfrac{k-1}{2} \rceil).$$

Here, by $\mathbf{t}^* = \mathbf{t}^*(\xi, k)$ we mean the sampling parameter that minimizes $q_\xi(\mathbf{t})$, for fixed ξ and sample size k. In the limit when $k \to \infty$, we thus have that $\boldsymbol{\tau}^* = (\frac{1}{2}, \frac{1}{2})$ as long as $\xi = 0$ [10].

As can be seen in Figure 7, this changes for larger values of ξ: There is a threshold ξ_c such that for $\xi > \xi_c$ the optimal sampling parameter is $\boldsymbol{\tau}^* = (\tau^*, 1-\tau^*)$ with $\tau^* < 1/2$. (By symmetry, $(1-\tau^*, \tau^*)$ is also optimal). For $\xi > \xi_c$, τ^* is the unique real solution in $[0, 1/2)$ of

$$\tfrac{d}{d\tau} q_\xi((\tau, 1-\tau)) \;=\; 0.$$

Figure 8 shows the function $\tau^* = \tau^*(\xi)$ for our branch prediction strategies. The critical threshold ξ_c and the shape of $\tau^*(\xi)$ differ depending on the prediction scheme, but τ^* always decreases steadily to 0 as $\xi \to \infty$, in accordance with our finding that the number of BMs is minimized when $\mathbf{t}/k \to \boldsymbol{\tau} = (0, 1)$. ξ_c is the solution of the equation

$$\tfrac{d^2}{d\tau^2} q_\xi((\tau, 1-\tau))\Big|_{\tau=1/2} \;=\; 0,$$

because $\xi = \xi_c$ is the point where $\tau = 1/2$ changes from a local minimum to a local maximum. The respective thresholds are

$$\begin{aligned}
\xi_c^{(\text{1-bit})} &= \frac{3(7-6\ln 2)}{2\ln 2 - 1} \approx 22.0644,\\
\xi_c^{(\text{2-bit sc})} &= \frac{3(7-6\ln 2)}{4\ln 2 - 1} \approx 4.8084,\\
\xi_c^{(\text{2-bit fc})} &= \frac{9(7-6\ln 2)}{10\ln 2 - 3} \approx 6.5039.
\end{aligned}$$

6.4 Comparison with Experimental Data.

Kaligosi and Sanders [7] report that running time was significantly improved by choosing skewed pivots on a Pentium 4 Prescott processor. They picked the $(\tau \cdot n)$-th of the array to partition (because the array contents where regular and known in advance); this same effect can be achieved via sampling with $k \to \infty$ and picking the $(\tau \cdot k)$-th within

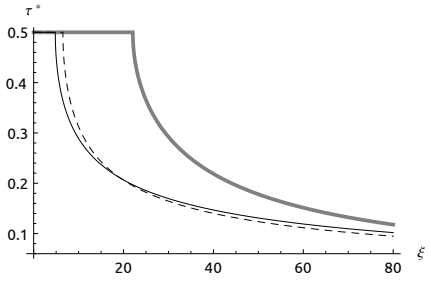


Figure 8: The function $\tau^* = \tau^*(\xi)$ for CQS with sample size $k \to \infty$ and $\boldsymbol{\tau} = (\tau, 1-\tau)$, for 1-bit (fat line), 2-bit sc (solid thin) and 2-bit fc (dashed).

the sample at each stage [10]. They also did a (partial) analysis of BMs in the infinite-size sampling regime. From their experiments, they concluded that $\tau^* \approx 1/10$. If we assume that the Pentium 4 used 2-bit fc predictor this corresponds to $\xi \approx 73$ (for 2-bit sc, we get $\xi \approx 83$). As $L = 31$ for this processor [1], this means that ξ is a bit larger than expected — or that our simple model is not fully accurate here. As the optimal τ^* is not much different for $\xi = 30$ or $\xi = 50$, see Figure 7, the model fits the observations satisfactorily.

Experiments conducted in other architectures (Opteron, Athlon, Sun) with shorter pipelines ($L \leq 20$) did not support the use of highly skewed pivots [1, 7], and τ^* would be closer to $1/2$.

In conclusion, despite we have proposed a very simplistic model ($BC + \xi BM$) of running time, it seems to capture essential features of the problem (the most important contribution missing is that due to memory hierarchies). The model has some explanatory power, even at a quantitative level, for the phenomena observed in experiments. As processor manufacturers often do not provide information or just very sketchy descriptions on their architectures, we can only "guess" the used branch prediction strategy and the value of ξ.

References

[1] P. Biggar, N. Nash, K. Williams, and D. Gregg. An experimental study of sorting and branch prediction. *Journal of Experimental Algorithmics*, 12: 1, June 2008.

[2] G. Brodal and G. Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *WADS 2005*, volume 3608 of *LNCS*, pages 385–395. Springer, 2005.

[3] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs, 2014. URL `http://www.agner.org/optimize/#manuals`.

[4] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley, 1994. ISBN 978-0-20-155802-9.

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

[6] C. A. R. Hoare. Algorithm 63: Partition. *Communications of the ACM*, 4(7):321, July 1961.

[7] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In T. Erlebach and Y. Azar, editors, *ESA 2006*, volume 4168 of *LNCS*, pages 780–791. Springer, 2006.

[8] S. Kushagra, A. López-Ortiz, A. Qiao, and J. I. Munro. Multi-Pivot Quicksort: Theory and Experiments. In *ALENEX 2014*, pages 47–60. SIAM, 2014.

[9] H. M. Mahmoud. *Sorting: A distribution theory*. John Wiley & Sons, 2000. ISBN 1-118-03288-8.

[10] C. Martínez and S. Roura. Optimal Sampling Strategies in Quicksort and Quickselect. *SIAM Journal on Computing*, 31(3):683, 2001.

[11] M. E. Nebel and S. Wild. Pivot Sampling in Dual-Pivot Quicksort. In *AofA 2014*, 2014. URL `http://arxiv.org/abs/1403.6602`.

[12] S. Roura. Improved Master Theorems for Divide-and-Conquer Recurrences. *Journal of the ACM*, 48 (2):170–205, 2001.

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

[14] R. Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[15] S. Wild. Java 7's Dual Pivot Quicksort. Master thesis, University of Kaiserslautern, 2012.

[16] S. Wild and M. E. Nebel. Average Case Analysis of Java 7's Dual Pivot Quicksort. In L. Epstein and P. Ferragina, editors, *ESA 2012*, volume 7501 of *LNCS*, pages 825–836. Springer, 2012.

[17] S. Wild, M. E. Nebel, and R. Neininger. Average Case and Distributional Analysis of Dual-Pivot Quicksort, 2013. URL `http://arxiv.org/abs/1304.0988`.

Appendix

A Index of Used Notation

In this section, we collect the notations used in this paper. (Some might be seen as "standard", but we think including them here hurts less than a potential misunderstanding caused by omitting them.)

Generic Mathematical Notation

$\ln n$ . . . . . natural logarithm.

$\mathbf{x}$ . . . . . to emphasize that $\mathbf{x}$ is a vector, it is written in **bold**; components of the vector are not written in bold: $\mathbf{x} = (x_1, \dots, x_d)$. operations on vectors are understood elementwise, e.g., $\mathbf{x} + 1 = (x_1 + 1, \dots, x_d + 1)$

X . . . . . to emphasize that X is a random variable it is Capitalized.

H_n . . . . . nth harmonic number; $H_n = \sum_{i=1}^{n} 1/i$.

$\mathrm{Dir}(\boldsymbol{\alpha})$ . . . . . Dirichlet distributed random variable with parameter $\boldsymbol{\alpha} \in \mathbb{R}^d_{>0}$; see Appendix C.

$\mathcal{U}(a, b)$ . . . . . uniformly in $(a, b) \subset \mathbb{R}$ distributed random variable.

$\mathrm{Gamma}(k, \theta)$ . . . Gamma distributed random variable with shape parameter $k \in \mathbb{R}_{>0}$ and scale parameter $\theta \in \mathbb{R}_{>0}$.

$\mathrm{Gamma}(k)$ . . . . Gamma$(k, 1)$ distributed random variable.

$\mathrm{B}(\alpha_1, \dots, \alpha_d)$ . . d-dimensional Beta function; defined by the (Lebesgue) integral $\int_{\Delta_d} x_1^{\alpha_1 - 1} \cdots x_d^{\alpha_d - 1} \, \mu(d\mathbf{x})$.

$\mathbb{E}[X]$ . . . . . expected value of X.

$\mathbb{E}[X \mid Y]$ . . . . . the conditional expectation of X given Y.

$\mathbb{P}(E)$, $\mathbb{P}(X = x)$. probability of an event E resp. probability for random variable X to attain value x.

$X \overset{\mathcal{D}}{=} Y$ . . . . . equality in distribution; X and Y have the same distribution.

$X_{(i)}$ . . . . . ith order statistic of a set of random variables $X_1, \dots, X_n$, i.e., the ith smallest element of $X_1, \dots, X_n$.

$a^{\underline{b}}$, $a^{\overline{b}}$ . . . . . factorial powers notation of [4]; "a to the b falling resp. rising".

$\gamma_{a,b}^{(c)}$ . . . . . see equation (5.11) on page 8

$\mathcal{H} = \mathcal{H}(\mathbf{t})$ . . . . discrete entropy; defined in equation (4.1) (page 4).

$\mathcal{H}^* = \mathcal{H}^*(\mathbf{p})$ . . . continuous (Shannon) entropy with base e; given in equation (4.2) (page 5).

Input to the Algorithm

n . . . . . length of the input array, i.e., the input size.

$\mathtt{A}$ . . . . . input array containing the items $\mathtt{A}[1], \dots, \mathtt{A}[n]$ to be sorted; initially, $\mathtt{A}[i] = U_i$.

U_i . . . . . ith element of the input, i.e., initially $\mathtt{A}[i] = U_i$. We assume $U_1, \dots, U_n$ are i. i. d. $\mathcal{U}(0, 1)$ distributed.

Notation Specific to the Algorithms

s . . . . . number of subproblems, i.e., $s = 2$ for classic Quicksort and $s = 3$ for Yaroslavskiy's Quicksort; $s - 1$ is thus the number of pivots in one partitioning step.

CQS, YQS . . . . CQS is the abbreviation for **c**lassic (single-pivot) **Q**uick**s**ort using Sedgewick's partitioning given in Algorithm 1; YQS likewise stands for **Y**aroslavskiy's (dual-pivot) **Q**uick**s**ort using the partitioning given in Algorithm 2.

$\mathbf{t} \in \mathbb{N}^s$ . . . . . pivot sampling parameter.

$k = k(\mathbf{t})$ . . . . . sample size; defined in terms of $\mathbf{t}$ as $k(\mathbf{t}) = t_1 + t_2 + \cdots + t_s + (s - 1) = \|\mathbf{t}\|_1 + \dim(\mathbf{t}) - 1$.

w . . . . . Insertionsort threshold; for $n \le w$, Quicksort recursion is truncated and we sort the subarray by Insertionsort.

$\mathrm{CQS}^w_{\mathbf{t}}$, $\mathrm{YQS}^w_{\mathbf{t}}$ . . CQS$^w_{\mathbf{t}}$ is the abbreviation for generalized classic Quicksort, where the pivot is chosen by generalized pivot sampling with parameter $\mathbf{t}$ and where we switch to Insertionsort for subproblems of size at most w; similarly YQS$^w_{\mathbf{t}}$ for Yaroslavskiy's dual-pivot partitioning.

$\mathbf{V} \in \mathbb{N}^k$ . . . . . (random) sample for choosing pivots in the first partitioning step.

P, Q . . . . . (random) values of chosen pivots in the first partitioning step; for classic Quicksort, only P.

$\mathbf{D} \in [0,1]^{s+1}$ . . . (random) spacings of the unit interval $(0,1)$ induced by the pivot(s), i.e., for CQS we have $\mathbf{D} = (P, 1-P)$ and for YQS $\mathbf{D} = (P, Q-P, 1-Q)$; in both cases, we have $\mathbf{D} \stackrel{\mathcal{D}}{=} \mathrm{Dir}(\mathbf{t}+1)$.

small element . . element U is small if $U < P$.

medium element . (only for YQS) element U is medium if $P < U < Q$.

large element . . for CQS: element U is large if $P < U$; for YQS: if $Q < U$.

ordinary element the $n-k$ array elements that have not been part of the sample.

k, g, ℓ . . . . . index variables used in the partitioning methods, see Algorithm 1 and Algorithm 2 (page 3); ℓ only appears in Algorithm 2.

Notation for Analysis of Branch Misses

BM . . . . . shorthand for **b**ranch **m**iss

1-bit, 2-bit sc, 2-bit fc
branch prediction schemes analyzed in this paper; see Section 5.4.

$C^{(c1)}, C^{(c2)}$ . . . . key comparison locations in CQS: $C^{(c1)}$ corresponds to the comparison in line 3, $C^{(c2)}$ to line 4 of Algorithm 1.

$C^{(c)}$ . . . . . the virtual combined comparison location consisting of $C^{(c1)}$ and $C^{(c2)}$.

$C^{(y1)}, \ldots, C^{(y4)}$. key comparison locations in YQS: $C^{(y1)}$ is in line 3, $C^{(y2)}$ in line 7, $C^{(y3)}$ in line 8 and $C^{(y4)}$ is in line 11 of Algorithm 2.

$C^{(i)}_n$ . . . . . $i \in \{c1, c2, c, y1, y2, y3, y4\}$; expected execution frequency of comparison location $C^{(i)}$ in the first partitioning step.

$\mathbb{P}^{(i)}_{\text{taken}}$ . . . . . $i \in \{c1, c2, c, y1, y2, y3, y4\}$; probability (conditional on $\mathbf{D}$) for the branch at comparison location $C^{(i)}$ to be taken; all executions of this branch are i. i. d.

$\mathbb{P}^{(i)}_{\text{BM}}$ . . . . . $i \in \{c1, c2, c, y1, y2, y3, y4\}$; probability (conditional on $\mathbf{D}$) for the branch at comparison location $C^{(i)}$ to be mispredicted; depends on the prediction scheme.

f_{bps}, $f_{bps}(p)$ . . . $bps \in \{$1-bit, 2-bit sc, 2-bit fc$\}$; steady-state miss-rate function for the branch prediction scheme bps; $f_{bps}(p)$ is the probability for a BM at a branch that is i. i. d. and taken with probability p — in the long run, i.e., when the predictor automaton has reached its steady state.

BM_n . . . . . (random) total number of branch misses to sort a random permutation of length n.

$T_\circ$ . . . . . (random) number of branch misses ($\circ = BM$), Bytecodes ($\circ = BC$) resp. combined costs ($\circ = Q$) of the first partitioning step on a random permutation of size n; $T_\circ(n)$ when we want to emphasize dependence on n.

a . . . . . coefficient of the linear term of $\mathbb{E}[T(n)]$ see Theorem 4.1 (page 5).

$\boldsymbol{\tau}$ . . . . . for the limiting case when $k \to \infty$, we assume that $\mathbf{t}/k = (t_1/k, \ldots, t_s/k) \to \boldsymbol{\tau}$. This corresponds to selecting precise order statistics, s. t. the relative size of subproblem i becomes precisely τ_i.

a^* . . . . . limit of a when $k \to \infty$ and $\mathbf{t}/k \to \boldsymbol{\tau}$.

ξ . . . . . cost of one branch miss relative to one CPU cycle; used in Q.

L . . . . . length of the instruction pipeline, i.e., the number of stages each instruction is broken into.

Q . . . . . combined cost measure $Q = BC + \xi \cdot BM$,

B Steady-State Miss-Rate Functions

In this appendix, we give details on the computation of the steady-state miss-rate for the two 2-bit predictor variants. These functions have also been derived by Biggar et al. [1] and Kaligosi and Sanders [7], respectively. We show the derivations here again for the reader's convenience.

B.1 2-Bit Saturating Counter. Consider a branch instruction, which is i. i. d. taken with probability p. The saturating-counter automaton then corresponds to the following Markov chain with states ordered as in Figure 2 (page 7):

$$\Pi = \Pi(p) = \begin{pmatrix} p & 1-p & 0 & 0 \\ p & 0 & 1-p & 0 \\ 0 & p & 0 & 1-p \\ 0 & 0 & p & 1-p \end{pmatrix} . \tag{B.1}$$

The stationary distribution is found to be

$$\pi(p) = \frac{1}{1-2p(1-p)} \tag{B.2}$$

$$\cdot \left(p^3, p^2(1-p), p(1-p)^2, (1-p)^3\right) . \tag{B.3}$$

The next branch execution is mispredicted iff we are in state 1 or 2 and the branch was not taken or if we are in state 3 or 4, but the branch was taken. Thus if we assume the predictor has reached its steady-state distribution, then we obtain the branch miss probability in dependence of the branch probability p as

$$f_{\text{2-bit sc}}(p) = \pi(p) \cdot (1-p, 1-p, p, p)^T \tag{B.4}$$

$$= \frac{p(1-p)}{1-2p(1-p)} . \tag{B.5}$$

B.2 2-Bit Flip-On-Consecutive. The analysis is along the same lines as above. The underlying Markov chain for the 2-bit fc predictor is shown in Figure 3 (page 7) and has the transition matrix

$$\tilde{\Pi} = \tilde{\Pi}(p) = \begin{pmatrix} p & 1-p & 0 & 0 \\ p & 0 & 0 & 1-p \\ p & 0 & 0 & 1-p \\ 0 & 0 & p & 1-p \end{pmatrix} . \tag{B.6}$$

with stationary distribution

$$\tilde{\pi}(p) = \frac{1}{1-p(1-p)} \tag{B.7}$$

$$\cdot \left(p^2, p^2(1-p), p(1-p)^2, (1-p)^2\right) . \tag{B.8}$$

Again assuming this steady state has been reached, the branch miss probability is

$$f_{\text{2-bit fc}}(p) = \tilde{\pi}(p) \cdot (1-p, 1-p, p, p)^T \tag{B.9}$$

$$= \frac{2p^2(1-p)^2 + p(1-p)}{1-p(1-p)} . \tag{B.10}$$

Appendices C & D

The remaining appendices cannot be given here due to the page limit for conference proceedings; they can be found in the arXiv version of this article, which is available at `http://arxiv.org/abs/1411.2059`.