Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms

Gerth Stølting Brodal[1,⋆] and Gabriel Moruz[1]

BRICS[⋆⋆], Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: {`gerth,gabi`}`@daimi.au.dk`

Abstract. Branch mispredictions is an important factor affecting the running time in practice. In this paper we consider tradeoffs between the number of branch mispredictions and the number of comparisons for sorting algorithms in the comparison model. We prove that a sorting algorithm using $O(dn \log n)$ comparisons performs $\Omega(n \log_d n)$ branch mispredictions. We show that Multiway MergeSort achieves this tradeoff by adopting a multiway merger with a low number of branch mispredictions. For adaptive sorting algorithms we similarly obtain that an algorithm performing $O(dn(1 + \log(1 + \text{Inv}/n)))$ comparisons must perform $\Omega(n \log_d(1 + \text{Inv}/n))$ branch mispredictions, where Inv is the number of inversions in the input. This tradeoff can be achieved by GenericSort by Estivill-Castro and Wood by adopting a multiway division protocol and a multiway merging algorithm with a low number of branch mispredictions.

1 Introduction

Modern CPUs include branch predictors in their architecture. Increased CPU pipelines enforce the prediction of conditional branches that enter the execution pipeline. Incorrect predictions determine the pipeline to be flushed with the consequence of a significant performance loss (more details on branch prediction schemes can be found in Section 2).

In this paper we consider comparison based sorting algorithms, where we assume that all element comparisons are followed by a conditional branch on the outcome of the comparison. Most sorting algorithms satisfy this property. Our contributions consist of tradeoffs between the number of comparisons required and the number of branch mispredictions performed by deterministic comparison based sorting and adaptive sorting algorithms.

We prove that a comparison based sorting algorithm performing $O(dn \log n)$ comparisons uses $\Omega(n \log_d n)$ branch mispredictions. We show that a variant of Multiway MergeSort adopting a d-way merger with a low number of branch mispredictions can achieve this tradeoff.

⋆ Supported by the Carlsberg Foundation (contract number ANS-0257/20) and the Danish Natural Science Foundation (SNF).

⋆⋆ Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation.

A well known result concerning sorting is that an optimal comparison based sorting algorithm performs $\Theta(n \log n)$ comparisons [4, Section 9.1]. However, in practice, there is often the case that the input sequence is nearly sorted. In such cases, one would expect a sorting algorithm to be faster than on random input inputs. To quantify the presortedness of a given sequence, several *measures of presortedness* have been proposed. A common measure of presortedness is the number of inversions in the input, Inv, formally defined by $\text{Inv}(X) = |\{(i, j) \mid i < j \ \wedge x_i > x_j\}|$ for a sequence $X = (x_1, \ldots, x_n)$.

A sorting algorithm is denoted *adaptive* if its time complexity is a function that depends both on the size of the input sequence and the presortedness existent in the input [14]. For a survey concerning adaptive sorting algorithms and definitions of different measures of presortedness refer to [6].

For comparison based adaptive sorting algorithms we prove that an algorithm that uses $O(dn(1 + \log(1 + \text{Inv}/n)))$ comparisons performs $\Omega(n \log_d(1 + \text{Inv}/n))$ branch mispredictions. This tradeoff is achieved by GenericSort introduced by Estivill-Castro and Wood [5] by adopting a d-way division protocol and d-way merging that performs a low number of branch mispredictions. The division protocol is a d-way generalization of the binary greedy division protocol considered in [1].

In [2] it was shown that the number of mispredictions performed by standard binary MergeSort is adaptive with respect to the measure Inv. The number of comparisons and branches performed is $O(n \log n)$ but the number of branch mispredictions is $O(n \log(\text{Inv}/n))$, assuming a dynamic prediction scheme that predicts the next outcome of a branch based on the previous outcomes of the same branch.

Sanders and Winkel [15] presented a version of distribution sort that exploited special machine instructions to circumvent the assumption that each comparison is followed by a conditional branch. E.g. does the Intel Itanium 2 have a wide variety of *predicated instructions*, i.e. instructions that are executed even if its predicate is false, but the results of that instruction are not committed into program state if the predicate is false. Using predicated instructions HeapSort [7, 16] can be implemented to perform $O(n \log n)$ comparisons, $O(n \log n)$ predicated increment operations, and $O(n)$ branch mispredictions (assuming a static prediction scheme, see Section 2), by simply using a predicated increment operation for choosing the right child of a node during the bubble-down phase of a deletemin operation.

The rest of the paper is structured as follows. In Section 2 we give an overview of the different branch predictions schemes implemented in the nowadays CPUs. In Section 3 we prove lower bound tradeoffs between the number of comparisons and the number of branch mispredictions for comparison based sorting and adaptive sorting algorithms. Matching upper bounds are provided in Sections 4 and 5, where we show how variants of multiway MergeSort and GenericSort, respectively, achieve the optimal tradeoffs between comparisons and branch mispredictions.

2 Branch prediction schemes

Branch mispredictions are an important factor affecting the running time in practice [9]. Nowadays CPUs have high memory bandwidth and increased pipelines, e.g. Intel Pentium IV Prescott has a 31 stage pipeline. The high memory bandwidth severely lowers the effect of caching over the actual running time when computation takes place in the internal memory.

When a conditional branch enters the execution pipeline of the CPU, its outcome is not known and therefore must be predicted. If the prediction is incorrect, the pipeline is flushed as it contains instructions corresponding to a wrong execution path. Obviously, each branch misprediction results in performance losses, which increase with the length of the pipeline.

Several branch prediction schemes have been proposed. A classification of the branch prediction schemes is given in Figure 1.

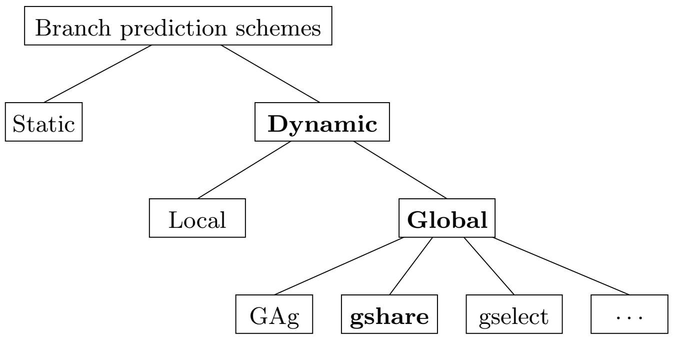

Fig. 1. A classification of the branch prediction schemes. The most popular branch predictors in each category are emphasized.

In a static prediction scheme, every branch is predicted in the same direction every time according to some simple heuristics, e.g. all forward branches taken, all backward branches not taken. Although simple to implement, their accuracy is low and therefore they are not widely used in practice.

The dynamic schemes use the execution history when predicting a given branch. In the local branch prediction scheme (see Figure 2, left) the direction of a branch is predicted using its past outputs. It uses a *pattern history table* (*PHT*) to store the last branch outcomes, indexed after the lower n bytes of the address of the branch instruction. However, the direction of a branch might depend on the output of other previous branch instructions and the local prediction schemes do not take advantage of it. To deal with this issue global branch prediction schemes were introduced [17]. They use a *branch history register* (BHR) that stores the outcome of the most recent branches. The different global prediction schemes vary only in the way the prediction table is looked up.

Three global branch prediction schemes proved very effective and are widely implemented in practice [13]. The *GAg* (Figure 2, middle) uses only the last m bits of the *BHR* to index the pattern history table, while *gshare* address the *PHT* by xor-ing the last bits n of the branch address with the last m bits of the *BHR*. Finally *gselect* concatenates the *BHR* with the lower bits of the branch address to obtain the index for the *PHT*.

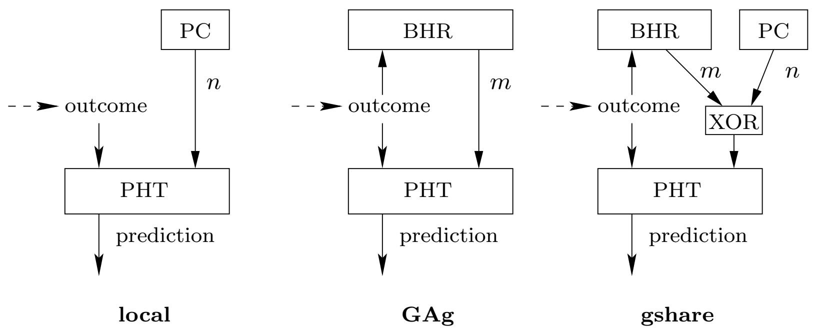

Fig. 2. Branch misprediction schemes.

The predictions corresponding to the entries in the *PHT* are usually obtained by the means of *two-bit saturating counters*. A two-bit saturating counter is an automaton consisting of four states, as shown in Figure 3.

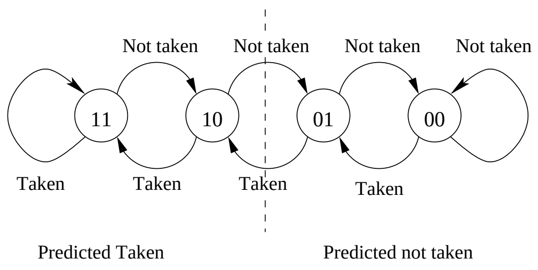

Fig. 3. Two-bit saturating counter.

Note that for the dynamic branch prediction schemes the same index in the *PHT* might correspond to several branches which would affect each other's predictions, constructively or destructively. This is known as the *aliasing effect* and reducing its negative effects is one of the main research areas in branch prediction schemes design.

Much research has been done on modeling branch mispredictions, especially in static analysis for upper bounding the worst case execution time (also known as WCET) [3, 11]. However, the techniques proposed involve too many hardware details and are too complicated to be used for giving branch misprediction complexities for algorithms. For the algorithms introduced in this paper, we show that even using a static branch prediction scheme, we can yield algorithms that achieve the lower bound tradeoffs between the number of comparisons and the number of branch mispredictions performed.

3 Lower bounds for sorting

In this section we consider deterministic comparison based sorting algorithms and prove lower bound tradeoffs between the number of comparisons and the number of branch mispredictions performed, under the assumption that each comparison between two elements in the input is immediately followed by a conditional branch that might be predicted or mispredicted. This property is satisfied by most sorting algorithms.

Theorem 1 introduces a worst case tradeoff between the number of comparisons and the number of branch mispredictions performed by sorting algorithms.

Theorem 1. *Consider a deterministic comparison based sorting algorithm A that sorts input sequences of size n using $O(dn \log n)$ comparisons, $d > 1$. The number of branch mispredictions performed by A is $\Omega(n \log_d n)$.*

Proof. Let T be the decision tree corresponding to A (for a definition of decision trees see e.g. [4, Section 9.1]). By assumption, each node in the tree corresponds to a branch that can be either predicted or mispredicted. We label the edges corresponding to mispredicted branches with 1 and the edges corresponding to correctly predicted branches with 0. Each leaf is uniquely labeled with the labels on the path from the root to the given leaf. Assuming the depth of the decision tree is at most D and the number of branch mispredictions allowed is k, each leaf is labeled by a sequence of at most D 0's and 1's, containing at most k 1's. By padding the label with 0's and 1's we can assume all leaf labels have length exactly $D + k$ and contain exactly k 1's. It follows that the number of labelings is at most the binomial coefficient $\binom{D+k}{k}$ and therefore the number of leaves is at most $\binom{D+k}{k}$.

Denoting the number of leaves by $N \geq n!$, we obtain that $\binom{D+k}{k} \geq N$, which implies $\log \binom{D+k}{k} \geq \log N$. Using $\log \binom{D+k}{k} \leq k(O(1) + \log \frac{D}{k})$ we obtain that:

$$k\left(O(1) + \log \frac{D}{k}\right) \geq \log N \ . \tag{1}$$

Consider $D = \delta \log N$ and $k = \varepsilon \log N$, where $\delta \geq 1$ and $\varepsilon \geq 0$. We obtain:

$$\varepsilon \log N \left(O(1) + \log \frac{\delta}{\varepsilon}\right) \geq \log N \ ,$$

and therefore $\varepsilon \left(O(1) + \log \frac{\delta}{\varepsilon}\right) \geq 1$. Using $\delta = O(d)$ we obtain $\varepsilon = \Omega(1/\log d)$. Taking into account that $\log N \geq \log(n!) = n \log n - O(n)$ we obtain $k = \Omega(n \log_d n)$. □

Manilla [12] introduced the concept of optimal adaptive sorting algorithms. Given an input sequence X and some measure of presortedness M, consider the set $\mathrm{below}(X, M)$ of all permutations Y of X such that $M(Y) \leq M(X)$. Considering only inputs in $\mathrm{below}(X, M)$, a comparison based sorting algorithm performs at least $\log|\mathrm{below}(X, M)|$ comparisons in the worst case. In particular, an adaptive sorting algorithm that is optimal with respect to measure Inv performs $O(n(1 + \log(1 + \mathrm{Inv}/n)))$ comparisons [6].

Theorem 2 introduces a worst case tradeoff between the number of comparisons and the number of branch mispredictions for comparison based sorting algorithms that are adaptive with respect to measure Inv.

Theorem 2. *Consider a deterministic comparison based sorting algorithm A that sorts an input sequence of size n using $O(dn(1 + \log(1 + \mathrm{Inv}/n)))$ comparisons, where* Inv *denotes the number of inversions in the input. The number of branch mispredictions performed by A is $\Omega(n \log_d(1 + \mathrm{Inv}/n))$.*

Proof. We reuse the proof of Theorem 1 by letting $N = |\mathrm{below}(X, M)|$, for an input sequence X.

Using (1), with the decision tree depth $D = \delta n(1 + \log(1 + \mathrm{Inv}/n))$ when restricted to inputs in $\mathrm{below}(X, M)$, $k = \varepsilon n(1 + \log(1 + \mathrm{Inv}/n))$ branch mispredictions, and $\log N = \Omega(n(1 + \log(1 + \mathrm{Inv}/n)))$ [8], we obtain:

$$\varepsilon n \left(1 + \log\left(1 + \frac{\mathrm{Inv}}{n}\right)\right)\left(O(1) + \log\frac{\delta}{\varepsilon}\right) = \Omega\left(n\left(1 + \log\left(1 + \frac{\mathrm{Inv}}{n}\right)\right)\right) .$$

This leads to:

$$\varepsilon\left(O(1) + \log\frac{\delta}{\varepsilon}\right) = \Omega(1) ,$$

and therefore $\varepsilon = \Omega(1/\log\delta)$. Taking into account that $\delta = O(d)$ we obtain that $\varepsilon = \Omega(1/\log d)$, which leads to $k = \Omega(n \log_d(1 + \mathrm{Inv}/n))$. □

Using a similar technique, lower bounds for other measures of presortedness can be obtained. For comparison based adaptive sorting algorithms, Figure 4 states lower bounds on the number of branch mispredictions performed in the worst case, assuming the given upper bounds on the number of comparisons. For definitions of different measures of presortedness, refer to [6].

4 An optimal sorting algorithm

In this section we introduce *Insertion d-way MergeSort*. It is a variant of d-way MergeSort that achieves the tradeoff stated in Theorem 1 by using an insertion sort like procedure for implementing the d-way merger. The merger is proven to perform a linear number of branch mispredictions.

We maintain two auxiliary vectors of size d. One of them stores a permutation $\pi = (\pi_1, \ldots, \pi_d)$ of $(1, \ldots, d)$ and the other one stores the indices in the input of the current element in each subsequence $i = (i_{\pi_1}, \ldots, i_{\pi_d})$, such that the

Measure	Comparisons	Branch mispredictions
Dis	$O(dn(1+\log(1+\text{Dis})))$	$\Omega(n\log_d(1+\text{Dis}))$
Exc	$O(dn(1+\text{Exc}\log(1+\text{Exc})))$	$\Omega(n\text{Exc}\log_d(1+\text{Exc}))$
Enc	$O(dn(1+\log(1+\text{Enc})))$	$\Omega(n\log_d(1+\text{Enc}))$
Inv	$O(dn(1+\log(1+\text{Inv}/n)))$	$\Omega(n\log_d(1+\text{Inv}/n))$
Max	$O(dn(1+\log(1+\text{Max})))$	$\Omega(n\log_d(1+\text{Max}))$
Osc	$O(dn(1+\log(1+\text{Osc}/n)))$	$\Omega(n\log_d(1+\text{Osc}/n))$
Reg	$O(dn(1+\log(1+\text{Reg})))$	$\Omega(n\log_d(1+\text{Reg}))$
Rem	$O(dn(1+\text{Rem}\log(1+\text{Rem})))$	$\Omega(n\text{Rem}\log_d(1+\text{Rem}))$
Runs	$O(dn(1+\log(1+\text{Runs})))$	$\Omega(n\log_d(1+\text{Runs}))$
SMS	$O(dn(1+\log(1+\text{SMS})))$	$\Omega(n\log_d(1+\text{SMS}))$
SUS	$O(dn(1+\log(1+\text{SUS})))$	$\Omega(n\log_d(1+\text{SUS}))$

Fig. 4. Lower bounds on the number of branch mispredictions for deterministic comparison based adaptive sorting algorithms for different measures of presortedness, given the upper bounds on the number of comparisons.

sequence $(x_{i_{\pi_1}}, \ldots, x_{i_{\pi_d}})$ is sorted. During the merging, $x_{i_{\pi_1}}$ is appended to the output sequence and i_{π_1} is incremented by 1 and then inserted in the vector i in a manner that resembles insertion sort: in a scan the value $y = x_{i_{\pi_1}}$ to be inserted is compared against the smallest elements of the sorted sequence until an element larger than y is encountered. This way, the property that the elements in the input sequence having indices $i_{\pi_1}, \ldots, i_{\pi_d}$ are in sorted order holds at all times. We also note that for each insertion the merger performs $O(1)$ branch mispredictions, even using a static branch prediction scheme.

Theorem 3. *Insertion d-way MergeSort performs $O(dn\log n)$ comparisons and $O(n\log_d n)$ branch mispredictions.*

Proof. For the simplicity of the proof, we consider a static prediction scheme where for the merging phase the element to be inserted is predicted to be larger than the minimum in the indices vector.

The number of comparisons performed at each level of recursion is $O(dn)$, since in the worst case each element is in the worst case compared against $d-1$ elements at each level. Taking into account that the number of recursion levels is $\lceil\log_d n\rceil$, the total number of comparisons is $O(dn\log_d n) = O(dn\log n)$.

In what concerns the number of branch mispredictions, for each element Insertion d-way MergeSort performs $O(1)$ branch mispredictions for each recursion level. That is because each element is inserted at most once in the indices array i at a given recursion level and for insertion sort each insertion is performed by using a constant number of branch mispredictions. Therefore we conclude that Insertion d-way MergeSort performs $O(n\log_d n)$ branch mispredictions. □

We stress that Theorem 3 states an optimal tradeoff between the number of comparisons and the number of branch mispredictions. This allows tuning the parameter d, such that Insertion d-way Mergesort can achieve the best running time on different architectures depending on the CPU characteristics, i.e. the clock speed and the pipeline length.

5 Optimal adaptive sorting

In this section we describe how d-way merging introduced in Section 4 can be integrated within *GenericSort* by Estivill-Castro and Wood [5], using a greedy-like division protocol. The resulting algorithm is proved to achieve the tradeoff between the number of comparisons and the number of branch mispredictions stated in Theorem 2.

GenericSort is based on MergeSort and works as follows: if the input is small, it is sorted using some alternate sorting algorithm; if the input is already sorted, the algorithm returns. Otherwise, it splits the input sequence into d subsequences of roughly equal sizes according to some *division protocol*, after which the subsequences are recursively sorted and finally merged to provide the sorted output.

The division protocol that we use, *GreedySplit*, is a generalization of the binary division protocol introduced in [1]. It partitions the input in $d+1$ subsequences $S_0, \ldots, S_d$, where S_0 is sorted and $S_1, \ldots, S_d$ have balanced sizes. In a single scan from left to right we build S_0 in a greedy manner while distributing the other elements to subsequences $S_1, \ldots, S_d$ as follows: each element is compared to the last element of S_0, if it is larger, it is appended to S_0; if not, it is distributed to an S_j such that at all times the i^{th} element in the input that is not in S_0 is distributed to $S_{1+i \bmod d}$. It is easy to see that S_0 is sorted and $S_1, \ldots, S_d$ have balanced sizes. For merging we use the insertion sort based merger introduced in Section 4.

Lemma 1 generalizes Lemma 3 in [1] to the case of d-way splitting.

Lemma 1. *If GreedySplit splits an input sequence X in $d+1$ subsequences $S_0, \ldots, S_d$, where S_0 is sorted and $d \geq 2$, then*

$$\mathrm{Inv}(X) \geq \mathrm{Inv}(S_1) + \cdots + \mathrm{Inv}(S_d) + \frac{d-1}{4}(\mathrm{Inv}(S_1) + \cdots + \mathrm{Inv}(S_d)) \, .$$

Proof. Let $X = (x_1, \ldots, x_n)$ and $S_i = (s_{i1}, \ldots, s_{it})$, for $1 \leq i \leq d$. For each s_{ij} denote by δ_{ij} its index in the input. By construction, S_i is a subsequence of X.

For some subsequence S_i consider an inversion $s_{ii_1} > s_{ii_2}$, with $i_1 < i_2$. By construction we know that for each subsequence S_k, with $k \neq i$, there exists some $s_{k\ell} \in S_k$ such that in the input sequence we have $\delta_{ii_1} < \delta_{k\ell} < \delta_{ii_2}$, see Figure 5. We prove that there exists at least an inversion between $s_{k\ell}$ and s_{ii_1} or s_{ii_2} in X. If $s_{k\ell} < s_{ii_2} < s_{ii_1}$ then there is an inversion between $s_{k\ell}$ and s_{ii_1}; if $s_{ii_2} < s_{k\ell} < s_{ii_1}$ then there are inversions in the input between $s_{k\ell}$ and both s_{ii_1} and s_{ii_2}; finally, if $s_{ii_2} < s_{ii_1} < s_{k\ell}$, there is an inversion between $s_{k\ell}$ and s_{ii_2}. Let $s_{k\ell_1}, \ldots, s_{k\ell_z}$ be all the elements in S_k such that $i_1 < \delta_{k\ell_1} < \cdots < \delta_{k\ell_z} < i_2$, i.e. all the elements from S_k that appear in the input between ranks δ_{ii_1} and δ_{ii_2}.

We proved that there is an inversion between $s_{k\ell_{\lfloor(1+z)/2\rfloor}}$ and at least one of s_{ii_1} and s_{ii_2}. Therefore, for the inversion (s_{ii_1}, s_{ii_2}) in S_i we have identified an inversion in X between an element in S_k and an element in S_i that is not present in any of $S_1, \ldots, S_d$. But this inversion can be counted for at most two different pairs in S_i, namely (s_{ii_1}, s_{ii_2}) and $(s_{ii_1}, s_{i(i_2+1)})$ if there is an inversion between

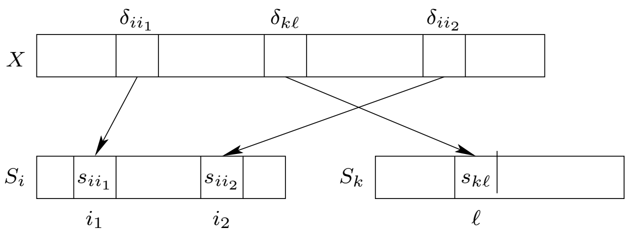

Fig. 5. Greedy division protocol. Between any two elements in S_i there is at least one element in S_k in the input sequence.

s_{ii_1} and $s_{k\ell_{\lfloor(1+z)/2\rfloor}}$ or (s_{ii_1}, s_{ii_2}) and $(s_{i(i_1-1)}, s_{ii_2})$ otherwise. In a similar manner in S_k the same inversion can be counted two times. Therefore, we obtain that for each inversion in S_i there is an inversion between S_i and S_k that can be counted four times. Taking into account that all the inversions in $S_1, \ldots, S_d$ are also in X, we obtain:

$$\mathrm{Inv}(X) \geq \mathrm{Inv}(S_1) + \cdots + \mathrm{Inv}(S_d) + \frac{d-1}{4}(\mathrm{Inv}(S_1) + \cdots + \mathrm{Inv}(S_d)) \ .$$

□

Theorem 4. *GreedySort performs* $O(dn(1 + \log(1 + \mathrm{Inv}/n)))$ *comparisons and* $O(n \log_d(1 + \mathrm{Inv}/n))$ *branch mispredictions.*

Proof. We assume a static branch prediction scheme. For the division protocol we assume that at all times the elements are smaller than the maximum of S_0, meaning that branch mispredictions occur when elements are appended to the sorted sequences. This leads to a total of $O(1)$ branch mispredictions per element for the division protocol, because the sorted sequences are not sorted recursively. For the merger, the element to be inserted is predicted to be larger than the minimum in the indices vector at all times. Following the proof of Theorem 3, we obtain that splitting and merging take $O(1)$ branch mispredictions per element for each level of recursion.

We follow the proof in [10]. First we show that at the first levels of recursion, until the number of inversions gets under n/d, GreedySort performs $O(dn(1 + \log(1+\mathrm{Inv}/n))$ comparisons and $O(n(1+\log_d(1+\mathrm{Inv}/n))$ branch mispredictions. Afterwards, we show that the remaining levels consume a linear number of branch mispredictions and comparisons.

We first find the level ℓ for which the number of inversions gets below n/d. Denote by Inv_i the total number of inversions in the subsequences at level i. Using the result in Lemma 1, we obtain $\mathrm{Inv}_i \leq \left(\frac{4}{d+3}\right)^i \mathrm{Inv}$. The level ℓ should therefore satisfy:

$$\left(\frac{4}{d+3}\right)^\ell \mathrm{Inv} \leq \frac{n}{d} \ ,$$

implying $\ell \geq \log_{\frac{d+3}{4}} \frac{\mathrm{Inv}\cdot d}{n}$.

Taking into account that at each level of recursion the algorithm performs $O(dn)$ comparisons and $O(n)$ branch mispredictions, we obtain that for the first $\ell = \lceil \log_{\frac{d+3}{4}} \frac{\mathrm{Inv} \cdot d}{n} \rceil$ levels we perform $O(dn \log_d(\mathrm{Inv}/n)) = O(dn \log(\mathrm{Inv}/n))$ comparisons and $O(n \log_d(\mathrm{Inv}/n))$ branch mispredictions.

We prove that for the remaining levels we perform a linear number of comparisons and branch mispredictions.

Let $L(x)$ be the recursion level where some element x is placed in a sorted sequence and $L(x) \geq \ell$. For each level of recursion j, where $\ell \leq j < L(x)$, x is smaller than the maximum in the sorted subsequence S_0 and therefore there is an inversion between x and the maximum in S_0 that does not exist in the recursive levels $j+1, j+2, \ldots$. It follows that $L(x) - \ell$ is bounded by the number of inversions with x at level ℓ.

Taking into account that the total number of inversions at level ℓ is at most n/d and that for each element at a level we perform $O(d)$ comparisons, we obtain that the total number of comparisons performed at the levels $\ell{+}1, \ell{+}2, \ldots$ is $O(n)$. Similarly, using the fact that for each element at each level $O(1)$ mispredictions are performed, we obtain that the total number of branch mispredictions performed for the levels below ℓ is $O(n/d)$. $\square$

Acknowledgment

We would like to thank Peter Bro Miltersen for very helpful discussions.

References

1. G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science. Springer Verlag, 2005.
2. G. S. Brodal, R. Fagerberg, and G. Moruz. On the adaptiveness of quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*. SIAM, 2005.
3. A. Colin and I. Puaut. Worst case execution time for a processor with branch prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, april 2000.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.
5. V. Estivill-Castro and D. Wood. Practical adaptive sorting. In *International Conference on Computing and Information - ICCI*, pages 47–54. Springer Verlag, 1991.
6. V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surverys*, 24(4):441–475, 1992.
7. R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.
8. L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation of linear lists. In *Proc. 9th Ann. ACM Symposium on Theory of Computing*, pages 49–60, 1977.
9. J. L. Hennesy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 1996.

10. C. Levcopoulos and O. Petersson. Splitsort – an adaptive sorting algorithm. *Information Processing Letters*, 39(1):205–211, 1991.
11. X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems Journal*, 29(1), January 2005.
12. H. Manilla. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Comput.*, 34:318–325, 1985.
13. S. McFarling. Combining branch predictors. Technical report, Western Research Laboratory, 1993.
14. K. Mehlhorn. *Data structures and algorithms. Vol. 1, Sorting and searching.* Springer Verlag, 1984.
15. P. Sanders and S. Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms (ESA)*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Verlag, 2004.
16. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.
17. Y.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM International Symposium on Computer Architecture (ISCA)*, 1992.