On the Adaptiveness of Quicksort

Gerth Stølting Brodal[*,†] Rolf Fagerberg[‡,§] Gabriel Moruz[*]

Abstract

Quicksort was first introduced in 1961 by Hoare. Many variants have been developed, the best of which are among the fastest generic sorting algorithms available, as testified by the choice of Quicksort as the default sorting algorithm in most programming libraries. Some sorting algorithms are adaptive, i.e. they have a complexity analysis which is better for inputs which are nearly sorted, according to some specified measure of presortedness. Quicksort is not among these, as it uses $\Omega(n \log n)$ comparisons even when the input is already sorted. However, in this paper we demonstrate empirically that the actual running time of Quicksort *is* adaptive with respect to the presortedness measure Inv. Differences close to a factor of two are observed between instances with low and high Inv value. We then show that for the randomized version of Quicksort, the number of element *swaps* performed is *provably* adaptive with respect to the measure Inv. More precisely, we prove that randomized Quicksort performs expected $O(n(1 + \log(1 + \text{Inv}/n)))$ element swaps, where Inv denotes the number of inversions in the input sequence. This result provides a theoretical explanation for the observed behavior, and gives new insights on the behavior of the Quicksort algorithm. We also give some empirical results on the adaptive behavior of Heapsort and Mergesort.

1 Introduction

Quicksort was introduced by Hoare in 1961 as a simple randomized sorting algorithm [8, 9]. Hoare proved that the expected number of comparisons performed by Quicksort for a sequence of n elements is essentially $2n \ln n \approx 1.4n \log_2 n$ [10]. Many variants and analysis of the algorithm have later been given, including [1, 12, 19, 20, 21]. In practice, tuned versions of Quicksort have turned out to be very competitive, and are used as standard sorting algorithms in many software libraries, e.g. C glibc, C++ STL-library, Java JDK, and the .NET Framework.

A sorting algorithm is called adaptive with respect to some measure of presortedness if, for some given input size, the running time of the algorithm is provably better for inputs with low value of the measure. Perhaps the most well-known measure is Inv, the number of inversions (i.e. pairs of elements that are in the wrong order) in the input. Other measures of presortedness include Rem, the minimum number of elements that must be removed for the remaining elements to be sorted, and Runs, the number of consecutive ascending runs. More examples of measures can be found in [5]. An example of an adaptive sorting algorithm is insertion sort using level-linked B-trees with finger searches for locating each new insertion point [13], which sorts in $O(n(1 + \log(1 + \text{Inv}/n)))$ time. In the comparison model, this is known to be optimal with respect to the measure Inv [5].

Most classic sorting algorithms, such as Quicksort, Heapsort [6, 23], and Mergesort [11], are not adaptive: their time complexity is $\Theta(n \log n)$ irrespectively of the input. However, a large body of adaptive sorting algorithms, such as the one in [13], has been developed over the last three decades. For an overview of this area, we refer the reader to the 1992 survey [5]. Later work on adaptive sorting includes [2, 3, 4, 15, 17].

Most of these results are of theoretical nature, and few practical gains in running time have been demonstrated for adaptive sorting algorithms compared to good non-adaptive algorithms.

Our starting point is the converse observation: the actual running time of a sorting algorithm could well be adaptive even if no worst case adaptive analysis (showing asymptotical improved time complexity for input instances with low presortedness) can be given.

In this paper, we study such practical adaptability and demonstrate empirically that significant gains can be found for the classic non-adaptive algorithms Quick-

[*]BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. E-mail: `{gerth, gabi}@brics.dk`.

[†]Supported by the Carlsberg Foundation (contract number ANS-0257/20) and the Danish Natural Science Research Council (SNF).

[‡]Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark. E-mail: `rolf@imada.sdu.dk`.

[§]Supported in part by the Danish Natural Science Research Council (SNF).

sort, Mergesort, and Heapsort, under the measure of presortedness Inv. Gains of more than a factor of three are observed.

Furthermore, in the case of Quicksort, we give theoretical backing for why this should be the case. Specifically, we prove that randomized Quicksort performs expected $O(n(1 + \log(1 + \text{Inv}/n)))$ element swaps. This not only provides new insight on the Quicksort algorithm, but it also gives a theoretical explanation for the observed behavior of Quicksort.

The reason that element swaps in Quicksort should be correlated with running time is (at least) two-fold: element swaps incur not only read accesses but also write accesses (thereby making them more expensive than read-only operations like comparisons), and element swaps in Quicksort are correlated with branch mispredictions during the partition procedure of the algorithm.

For Quicksort and Mergesort we show empirically the strong influence of branch mispredictions on the running time. This is in line with recent findings of Sanders and Winkel [18], who demonstrate the practical importance of avoiding branch mispredictions in the design of sorting algorithms for current CPU architectures. For Heapsort, our experiments indicate that data cache misses are the dominant factor for the running time.

The observed behavior of Mergesort can be explained using existing results (see Section 4.2), while we leave open the problem of a theoretical analysis of the observed behavior of Heapsort. Since our theoretical contributions regard Quicksort, we concentrate our experiments on this algorithm, while mostly indicating that similar gains can be found empirically also for Mergesort and Heapsort.

The main result of this paper is Theorem 1.1 below stating a dependence between the expected number of swaps performed by randomized Quicksort and the number of inversions in the input. In Section 4, the theorem is shown to correlate very well with empirical results.

Theorem 1.1. *The expected number of element swaps performed by randomized Quicksort is at most* $n + n \ln\left(\frac{2\text{Inv}}{n} + 1\right)$.

We note that the bound on the number of element swaps in Theorem 1.1 is not optimal for sorting algorithms. Straightforward in-place selection sort uses $O(n^2)$ comparisons but performs at most $n - 1$ element swaps for any input. An optimal in-place sorting algorithm performing $O(n)$ swaps and $O(n \log n)$ comparisons was recently presented in [7].

This paper is organized as follows: In Section 2 we prove Theorem 1.1. In Section 3 we describe our experimental setup, and in Section 4 we describe and discuss our experimental results. Parts of our proof of Theorem 1.1 were inspired by the proof by Seidel [22, Section 5] of the expected number of comparisons performed by randomized Quicksort.

```
#define Item int
#define random(l,r) (l+rand() % (r-l+1))
#define swap(A, B) { Item t = A; A = B; B = t; }

void quicksort(Item a[], int l, int r)
   { int i;
     if (r <= l) return;
     i = partition(a, l, r);
     quicksort(a, l, i-1);
     quicksort(a, i+1, r);
   }

int partition(Item a[], int l, int r)
  { int i = l-1, j = r+1, p = random(l,r);
    Item v = a[p];
    for (;;) {
        while (++i < j && a[i] <= v);
        while (--j > i && v <= a[j]);
        if (j <= i) break;
        swap(a[i], a[j]);
      }
    if (p < i) i--;
    swap(a[i], a[p]);
    return i;
  }
```

Figure 1: Randomized Quicksort.

2 Expected number of swaps by randomized Quicksort

In this section we analyze the expected number of element swaps performed by the classic version of randomized Quicksort where in each recursive call a random pivot is selected. The C code for the specific algorithm considered is given in Figure 1. The parameters `l` and `r` are the first and last element, respectively, of the segment of the array `a` to be sorted.

We assume that the n input elements are distinct. In the following, let $(x_1, \ldots, x_n)$ denote the input sequence, and let π_i be the rank of x_i in the sorted sequence. The number of inversions in the input sequence is denoted by Inv. The main observation used in the proof of Theorem 1.1 is that an element x_i that has not yet been moved from its input position i is swapped during a partitioning step if and only if the selected pivot x_j satisfies $i \leq \pi_j < \pi_i$ or $\pi_i < \pi_j \leq i$, or x_i is itself the pivot element (this is seen by inspection of the code, noting that after a partitioning step, the pivot element x_j resides at its final position π_j). We shall only need the "only if" part.

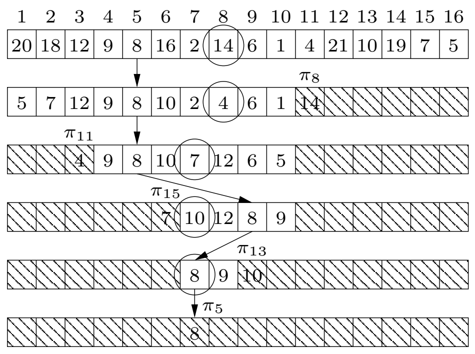

Figure 2: The partitions involving element 8.

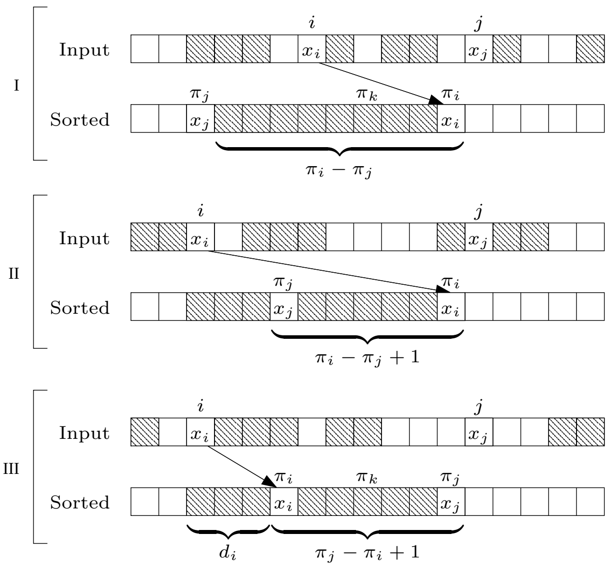

Figure 3: The three different cases of Lemma 2.2.

Fact 2.1. *When x_i is swapped the first time, the pivot x_j of the current partitioning step satisfies $i \leq \pi_j < \pi_i$ or $\pi_i < \pi_j \leq i$, or x_i is itself the pivot element.*

Figure 2 illustrates how the element $x_5 = 8$ is moved during the execution of randomized Quicksort. Circled elements are the selected pivots. The first two selected pivots 14 and 4 do not cause 8 to be swapped, since 8 is already correctly located with respect to the final positions of of the pivots 14 and 4. The first pivot causing 8 to be swapped is $x_{15} = 7$, since $\pi_5 = 7$, $\pi_{15} = 6$, and $5 \leq \pi_{15} < \pi_5$.

In the succeeding recursive calls after the first swap of an element x_i, the positions of x_i in the array are unrelated to i and π_i. Eventually, x_i is either picked as a pivot or becomes a single element input to a recursive call (the base case is reached), after which x_i does not move further.

In the following we let $d_i = |\pi_i - i|$, i.e. the distance of x_i from its correct position in the sorted output. The correlation between Inv and the d_i values is captured by the following lemma:

Lemma 2.1. Inv $\leq \sum_{i=1}^n d_i \leq 2$Inv.

Proof. For the left inequality, Inv $\leq \sum_{i=1}^n d_i$, we consider the following algorithm: If there is an element x_i not at its correct position, move x_i to position π_i, such that position π_i temporarily contains both x_i and x_{π_i} in sorted order. Next move x_{π_i} to its correct position, and repeat moving an element from the position temporarily containing two elements to its correct position, until we move an element to position i. Repeat until the sequence is sorted. By moving element x_i from position i to its correct position π_i, we move x_i over the $d_i - 1$ elements at positions between i and π_i and possibly the current element at position π_i. This decreases the number of inversions in the sequence by at most d_i, namely any inversions between x_i and each of the at most d_i elements moved over. In the final sorted sequence there are no inversions, hence we have Inv $\leq \sum_{i=1}^n d_i$.

For the right inequality, $\sum_{i=1}^n d_i \leq 2$Inv, consider some x_i with $\pi_i \geq i$. In the input sequence there are at least d_i inversions between x_i and other input elements, since there are at least d_i elements less than x_i with indices greater than i in the input sequence. A similar argument holds for the case when $\pi_i < i$. Taking into account that we may count the same inversion twice, we obtain $\sum_{i=1}^n d_i \leq 2$Inv. □

The constants in Lemma 2.1 are the best possible. For even n, the sequence $(2, 1, 4, 3, 6, 5, \ldots, n, n-1)$ has Inv $= n/2$ and $\sum_{i=1}^n d_i = n$, i.e. $(\sum_{i=1}^n d_i)/\text{Inv} = 2$, whereas the sequence $(n, n-1, n-2, n-3, \ldots, 3, 2, 1)$ has Inv $= n(n-1)/2$ and $\sum_{i=1}^n d_i = n^2/2$, i.e. $(\sum_{i=1}^n d_i)/\text{Inv} = 1 + \frac{1}{n-1}$ which converges to one for increasing n.

For the proof of Theorem 1.1 we make the following definition:

Definition 2.1. *For $i \neq j$ let X_{ij} denote the indicator variable that is one if and only if there is a recursive call to* `quicksort` *where x_j is selected as the pivot in the partition step and x_i is swapped during this partition step.*

Note that x_j can at most once become a pivot, since after a partition with pivot x_j the input to the recursive calls do not contain x_j. Furthermore note that the elements swapped in a partition step with pivot x_j are the elements in the input to the partition which are placed incorrectly relatively to the final position π_j of x_j.

There are three cases where $X_{ij} = 0$: (*i*) x_j is never selected as a pivot, i.e. there exists a recursive call where x_j is the only element to be sorted; (*ii*) x_j is selected as a pivot in a recursive call and x_i is not in the input to this recursive call; and (*iii*) x_j is selected as a pivot in a recursive call and x_i is in the input to this recursive call, but x_i is not swapped because it is placed correctly relatively to the final position π_j of x_j.

LEMMA 2.2. $\Pr[X_{ij} = 1] \leq$

$$\begin{cases} 0 & \textit{if } \pi_j < i \leq \pi_i \textit{ or } \pi_i \leq i < \pi_j \ , \\ \frac{1}{|\pi_i - \pi_j|+1} & \textit{if } i \leq \pi_j < \pi_i \textit{ or } \pi_i < \pi_j \leq i \ , \\ \frac{1}{|\pi_i - \pi_j|+1} - \frac{1}{|\pi_i - \pi_j|+1+d_i} & \textit{otherwise.} \end{cases}$$

Proof. For the case (*i*) where x_j is never selected as a pivot for a partition, we in the following adopt the convention that x_j is considered the pivot for the recursive call where the input consists of x_j only. This ensures that each element becomes a pivot exactly once.

We first note that the probability that x_i is in the input to the recursive call with pivot x_j is $\frac{1}{|\pi_i - \pi_j|+1}$, since this is the probability that x_j is the first element chosen as a pivot among the $|\pi_i - \pi_j| + 1$ elements x_k with $\pi_i \leq \pi_k \leq \pi_j$ or $\pi_j \leq \pi_k \leq \pi_i$ (if the first pivot x_k among the $|\pi_i - \pi_j| + 1$ elements is not x_j, then the selected pivot x_k will cause x_i and x_j to not appear together in any input to succeeding recursive calls).

To prove the lemma we consider the three different cases depending on the relative order of i, π_i, and π_j. In the following we assume $i \leq \pi_i$. The cases where $\pi_i < i$ are symmetric. The three possible scenarios are shown in Figure 3.

First consider the case where $\pi_j < i \leq \pi_i$, see Figure 3 (I). If a pivot x_k is selected with $\pi_j < \pi_k \leq \pi_i$ before x_j becomes a pivot, then x_i and x_j do not appear together in any input to succeeding recursive calls, so x_i cannot be involved in the partition with pivot x_j. The only other possibility is that x_j is a pivot before any element x_k with $\pi_j < \pi_k \leq \pi_i$ becomes a pivot, but then by Fact 2.1 x_i has not been moved when x_j becomes a pivot, and the partitioning with pivot x_j does not swap x_i.

For the second case, where $i \leq \pi_j < \pi_i$, see Figure 3 (II), we bound the probability that X_{ij} equals one by the probability that x_i is in the input to the recursive call with pivot x_j. As argued above, this probability is $\frac{1}{|\pi_i - \pi_j|+1}$.

For the last case where $i \leq \pi_i < \pi_j$, see Figure 3 (III), we consider the probability that x_i is in the input to the recursive call with pivot x_j and x_i is not swapped. This is at least the probability that x_j is the first element chosen as a pivot among the $|\pi_i - \pi_j| + 1 + d_i$ elements x_k with $i \leq \pi_k \leq \pi_j$, since then by Fact 2.1 x_i has not been moved yet when x_j becomes the pivot, and the partitioning with pivot x_j does not swap x_i. It follows that the probability that x_i is in the input to the recursive call with pivot x_j and x_i is not swapped, is at least $\frac{1}{|\pi_i - \pi_j|+1+d_i}$. Since the probability that x_i is in the input to the recursive call with pivot x_j is $\frac{1}{|\pi_i - \pi_j|+1}$, the lemma follows. □

Using Lemma 2.1 and Lemma 2.2 we now have the following proof of Theorem 1.1.

Proof (Theorem 1.1). The `for`-loop in the partitioning procedure in Figure 1 only swaps non-pivot elements and each element is swapped at most once in the loop. The loop is followed by one swap involving the pivot. Since a swap of two elements x_i and x_k not involving the pivot x_j are counted by the two indicator variables X_{ij} and X_{kj}, the expected number of swaps is at most

$$\begin{aligned} & \mathrm{E}\left[\sum_{j=1}^{n}\left(1 + \frac{1}{2}\sum_{i=1, i\neq j}^{n} X_{ij}\right)\right] \\ & = n + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1, i\neq j}^{n} \Pr(X_{ij} = 1) \\ (2.1) \quad & \leq n + \frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{d_i}\frac{1}{k+1} + \right. \\ & \qquad \left. \sum_{k=1}^{n}\left(\frac{1}{k+1} - \frac{1}{k+1+d_i}\right)\right) \\ & \leq n + \frac{1}{2}\sum_{i=1}^{n}\left(2\sum_{k=1}^{d_i}\frac{1}{k+1}\right) \\ & = \sum_{i=1}^{n}\sum_{k=1}^{d_i+1}\frac{1}{k} \\ (2.2) \quad & \leq \sum_{i=1}^{n}(1 + \ln(d_i + 1)) \\ (2.3) \quad & \leq n + n\ln\frac{\sum_{i=1}^{n}(d_i + 1)}{n} \\ (2.4) \quad & \leq n + n\ln\left(\frac{2\mathrm{Inv}}{n} + 1\right) \end{aligned}$$

where (2.1) follows from Lemma 2.2, (2.2) follows from $\sum_{i=1}^{n}\frac{1}{i} \leq 1 + \ln n$, (2.3) follows from the concavity of the logarithm function, and (2.4) follows from Lemma 2.1. □

It should be noted that the upper bound achieved in (2.3) using the concavity of the logarithm function can be much larger than the value (2.2). As an example, if there are $\Theta(n/\log n)$ d_i values of size $\Theta(n)$ and the rest of the d_i values are zero, then the difference between (2.2) and (2.3) is a factor $\Theta(\log n)$, i.e. the upper bound on the expected number of swaps stated in Theorem 1.1 can be a factor of $\log n$ from the actual bound.

3 Experimental setup

In the remainder of this paper, we investigate whether classic, theoretically non-adaptive sorting algorithms can show adaptive behavior in practice. We find that this indeed is the case—the running times for Quicksort, Mergesort, and Heapsort are observed to improve by factors between 1.5 and 3.5 when the Inv value of the input goes from high to low. Furthermore, the improvements for Quicksort are in very good concordance with Theorem 1.1, which shows this result to be a likely explanation for the observed behavior.

In more detail, we study how the number of inversions in the input sequence affects the number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of data cache misses of the version of Quicksort shown in Figure 1. We also study the behavior of two variants of Quicksort, namely the randomized version that chooses the median of three random elements as a pivot, and the deterministic version that chooses the middle element as a pivot. Finally, we study the same questions for the classic sorting algorithms Heapsort and Mergesort.

The input elements are 4 byte integers. We generate two types of input, having small d_i's and large d_i's, respectively. We generate a sequence with small d_i's by choosing each element x_i randomly in $[i-d, \ldots, i+d]$ for some parameter d, whereas the sequence with large d_i's is generated by letting $x_i = i$ with the exception of d random i's for which x_i is chosen randomly in $[1, \ldots, n]$. We perform our experiments by varying the disorder (by varying d) while keeping the size n of the input sequence constant. For most experiments, the input size is 2×10^6, but we also investigate larger and smaller input sizes.

Our experiments are conducted on two different machines. One of the machines has an Intel P4 2.4 GHz CPU with 512 MB RAM, running linux 2.4.20, while the other has an AMD Athlon XP 2400+ 2.0 GHz CPU with 256 MB RAM, running linux 2.4.22. On both machines the C source code was compiled using gcc-3.3.2 with optimization level -O3. The number of branch mispredictions and L2 data cache misses was obtained using the PAPI library [16] version 3.0.

Source code and the plotted data are available at `ftp://ftp.brics.dk/RS/04/47/Experiments`.

4 Experimental results

4.1 Quicksort. We first analyze the dependence of the version of Quicksort shown in Figure 1 on the number of inversions in the input.

Figure 4 shows our data for the AMD Athlon architecture. The number of comparisons is independent of the number of inversions in the input, as expected. For the number of element swaps, the plot is very close to linear when considering the input sequence with small d_i's. Since the x-axis shows $\log(\text{Inv})$, this is in very good correspondence with the bound $O(n \log(\text{Inv}/n))$ of Theorem 1.1 (recall that n is fixed in the plot). For the input sequence with large d_i's, the plot is different. This is a sign of the slack in the analysis (for this type of input) noted after the proof of Theorem 1.1. We will demonstrate below that this curve is in very good correspondence with the version of the bound given by Equation (2.2). The plots for the number of branch mispredictions and for the running time clearly show that they are correlated with the number of element swaps. For the number of branch mispredictions, this is explained by the fact that an element swap is performed after the two while loops stop, and hence corresponds to two branch mispredictions. For the running time, it seems reasonable to infer that branch mispredictions are a dominant part of the running time of Quicksort on this type of architecture. Finally, the number of data cache misses seems independent of the presortedness of the input sequence, in correspondence with the fact that for all element swaps, the data to be manipulated is already in the cache and therefore the element swaps do not generate additional cache misses.

Figure 5 show the same plots for the P4 architecture, except that we were not able to obtain data for L2 data cache misses. We note that the plots follow the same trends as in Figure 4. The number of comparisons and the number of element swaps are approximately the same, but the running time is affected by up to a factor of 1.8 on the P4, while only by up to a factor of 1.42 on the Athlon. One reason for this behavior is the number of branch mispredictions, which is slightly smaller for the Athlon. Also, the length of the pipeline, shorter for Athlon, makes the branch mispredictions more costly on a P4 than on an Athlon.

Similar observations on the resemblance between the data for the two architectures apply to all our experiments. For this reason, and because of the extra data for L2 that we have for Athlon, we for the remaining plots restrict ourselves to the Athlon architecture.

We now turn to the variants of Quicksort. Figure 6 shows the number comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the L2 data cache misses for the version of Quicksort that chooses as pivot the median of three random elements in the input sequence. We note that the plots have a behavior similar to the ones for the version of Quicksort shown in Figure 4. However, some improvements are noticed. The three-median pivot Quicksort performs around 25% less comparisons, due to the better choice of the pivot. This immediately triggers a slight improvement in the number of data cache misses. However, the number of branch mispredictions increases due to the extra branches required to compute the median of three elements. The number of element swaps remains approximately the same.

Figure 7 shows the same plots for the deterministic version of Quicksort that chooses the middle element as pivot. In this case we note that the number of comparisons does depend on the presortedness of the input. This is because for small disorder, the middle element is very close to the median and therefore the number of comparisons is close to $n \log n$, as opposed to $\approx 1.4n \log n$ expected for the randomized Quicksort [10]. The good pivot choice for small disorder in the input also triggers a smaller number of comparisons and branch mispredictions. However, for large disorder, the number of comparisons is larger compared to randomized median-of-three Quicksort due to bad pivot choices. Also, the running time is affected by up to a factor of two by the disorder in the input.

Figure 8 and Figure 9 show that when varying the input size n, the behavior of the plots remains the same for randomized Quicksort. Hence, our findings do not seem to be tied to the particular choice of $n = 2 \times 10^6$.

Finally, in Figure 10 we demonstrate that the number of element swaps is very closely related to $\sum_{i=1}^{n} \log d_i$, cf. the comment after the proof of Theorem 1.1. Hence the reason for the non-linear shape of the previous plots for input sequences with large d_i's seems to be the slack introduced (for this type of input) after Equation (2.2) in the proof of Theorem 1.1. As in the other cases, the running time and the number of branch mispredictions follow the same trend as the number of swaps.

4.2 Heapsort and Mergesort. We briefly demonstrate that also for Heapsort and Mergesort, the actual running time varies with the presortedness of the input.

For Heapsort, Figure 11 shows the way the number of inversions in the input affects the number of comparisons, the number of elements swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses for input sequences of constant length $n = 2 \times 10^6$. The number of comparisons and the number of element swaps performed by Heapsort is affected slightly, while the number of branch mispredictions is affected somewhat more. However, the number of L2 data cache misses is greatly affected, and varies by more than a factor of ten. The running time shows a virtually identical behavior, except the increase is by a factor close to four. This suggests that data cache misses are the dominant factor for the running time for Heapsort on this architecture. We leave open the question of a theoretical analysis of the number of cache misses of Heapsort as a function of Inv.

For Mergesort, we focus on the binary merge process, and count the number of times there is an alternation in which of the two input subsequences provides the next element output. It is easy to verify that the number of such alternations is dominated by the running time of the Mergesort algorithm by Moffat [14] based on merging by finger search trees, which was proved to have a running time of $O(n \log \frac{\text{Inv}}{n})$, i.e. the number of alternations by standard Mergesort is $O(n \log \frac{\text{Inv}}{n})$. The plots in Figure 12 show a very similar behavior for the number of alternations, the number of branch mispredictions, and the running time. The number of alternations is clearly correlated to the number of branch mispredictions, and these appear to be a dominant factor for the running time of Mergesort. The number of data cache misses increases only slightly for large disorder in the input.

References

[1] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software—Practice and Experience*, 23(11):1249–1265, Nov. 1993.

[2] A. Elmasry. Priority queues, pairing, and adaptive sorting. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2002.

[3] A. Elmasry. Adaptive sorting with AVL trees. Technical Report 2003-46, DIMACS, Feb. 2004.

[4] A. Elmasry and M. L. Fredman. Adaptive sorting and the information theoretic lower bound. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

[5] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *Computing Surveys*, 24:441–476, 1992.

[6] R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[7] G. Franceschini and V. Geffert. An In-Place Sorting with $O(n \log n)$ Comparisons and $O(n)$ Moves. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 242–250, 2003.

[8] C. A. R. Hoare. Algorithm 63: Partition. *Commun. ACM*, 4(7):321, 1961.

[9] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[10] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, April 1962.

[11] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[12] C. Martínez and S. Roura. Optimal sampling strategies in Quicksort and Quickselect. *SIAM Journal on Computing*, 31(3):683–705, June 2002.

[13] K. Mehlhorn. *Sorting and Searching*. Springer Verlag, Berlin, 1984.

[14] A. Moffat, O. Petersson, and N. C. Wormald. Sorting and/by merging finger trees. In *Algorithms and Computation: Third International Symposium, ISAAC '92*, volume 650 of *Lecture Notes in Computer Science*, pages 499–508. Springer Verlag, Berlin, 1992.

[15] A. Pagh, R. Pagh, and M. Thorup. On adaptive integer sorting. In *12th Annual European Symposium on Algorithms, ESA 2004*, volume 3221 of *Lecture Notes in Computer Science*, pages 556–567. Springer Verlag, Berlin, 2004.

[16] PAPI (Performance Application Programming Interface). Software library found at `http://icl.cs.utk.edu/papi/`, 2004.

[17] O. Petersson and A. Moffat. A framework for adaptive sorting. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 59, 1995.

[18] P. Sanders and S. Winkel. Super scalar sample sort. In *12th Annual European Symposium on Algorithms, ESA 2004*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Verlag, Berlin, 2004.

[19] R. Sedgewick. *Quicksort*. PhD thesis, Stanford University, Stanford, CA, May 1975. Stanford Computer Science Report STAN-CS-75-492.

[20] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[21] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21:847–857, 1978.

[22] R. Seidel. Backwards analysis of randomized geometric algorithms. Technical Report TR-92-014, International Computer Science Institute, Univeristy of Calfornia at Berkeley, February 1992.

[23] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

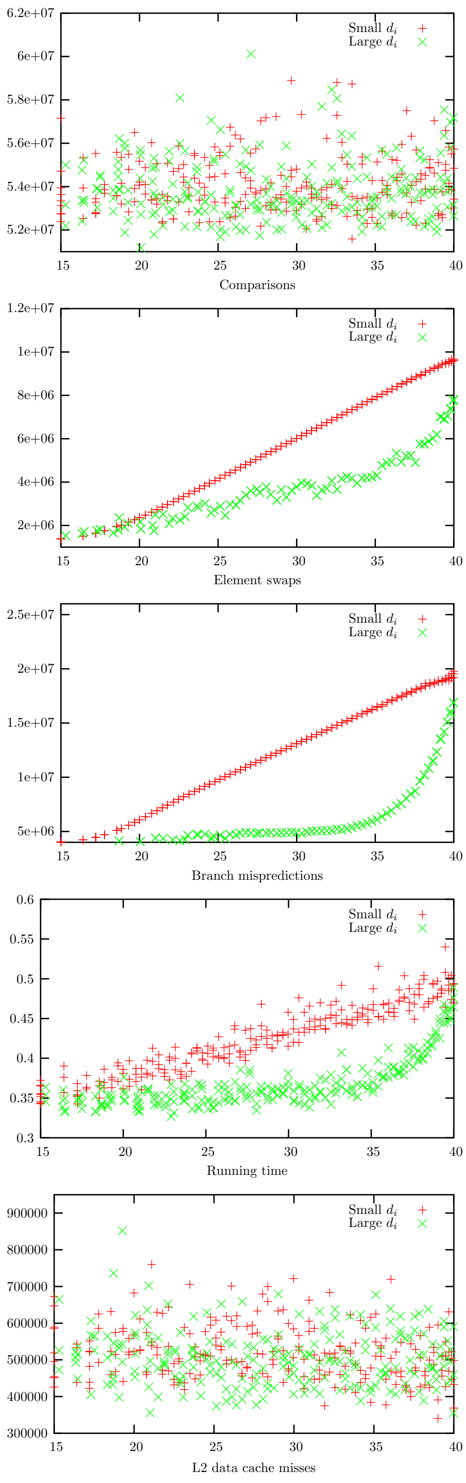

Figure 4: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 2 \times 10^6$. The x-axis shows log(Inv).

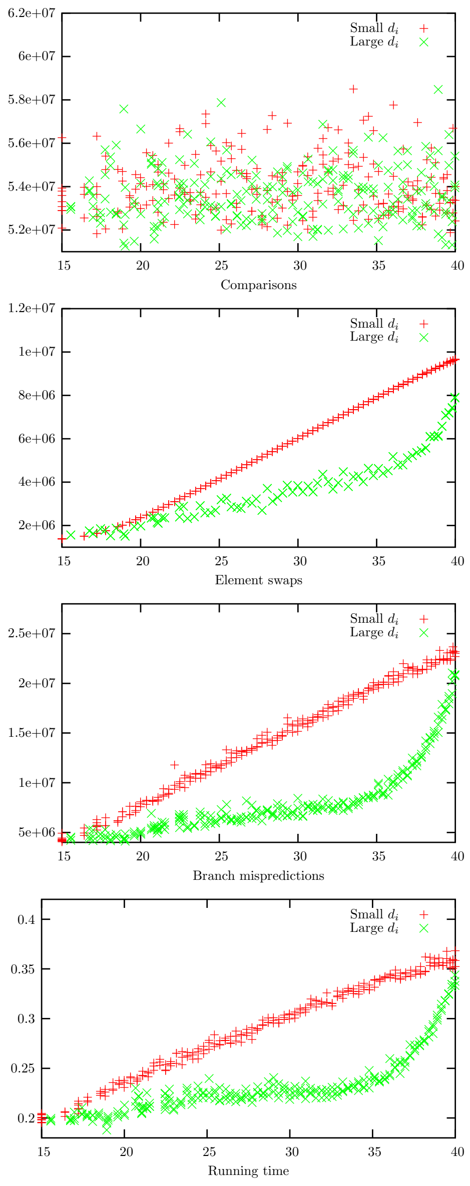

Figure 5: The number of comparisons, the number of element swaps, the number of branch mispredictions, and the running time of randomized Quicksort on P4, for $n = 2 \times 10^6$. The x-axis shows log(Inv).

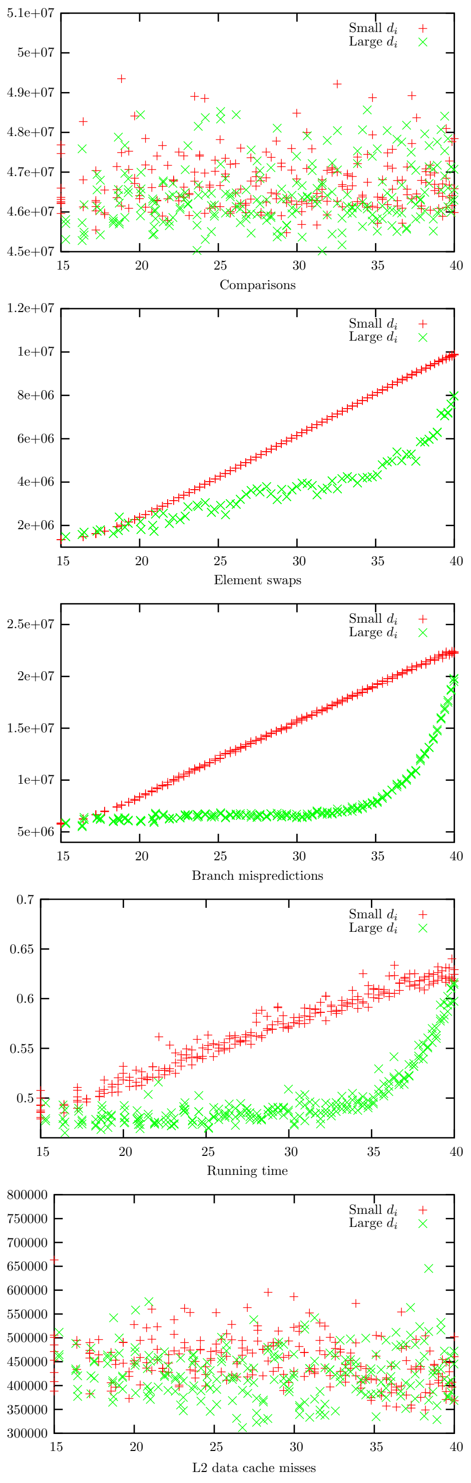

Figure 6: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized median-of-three Quicksort on Athlon, for $n = 2 \times 10^6$. The x-axis shows log(Inv).

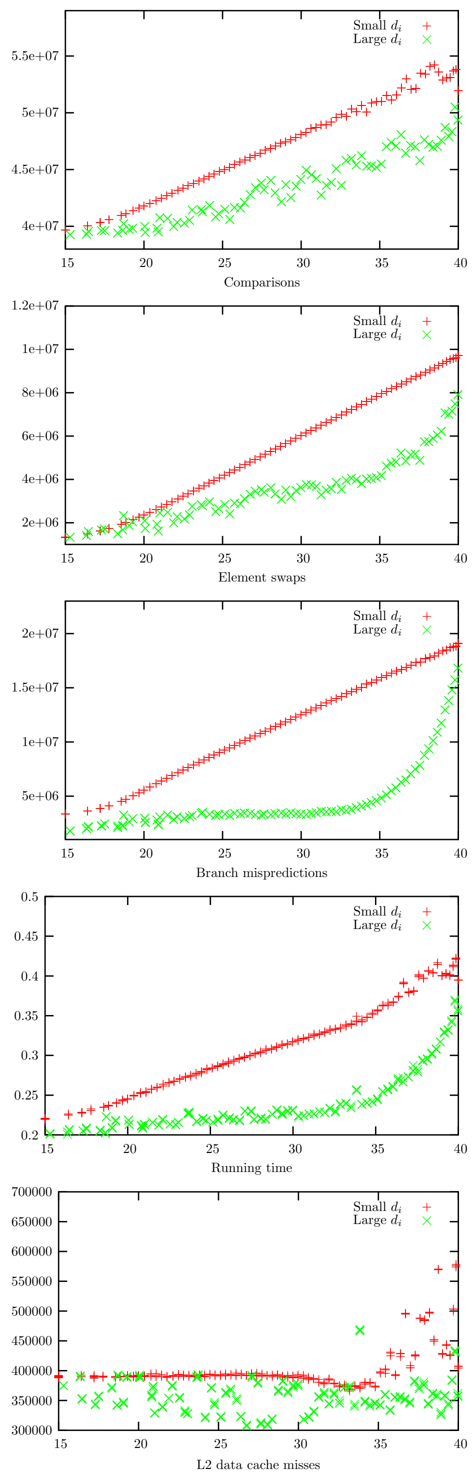

Figure 7: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by deterministic Quicksort on Athlon, for $n = 2 \times 10^6$. The x-axis shows log(Inv).

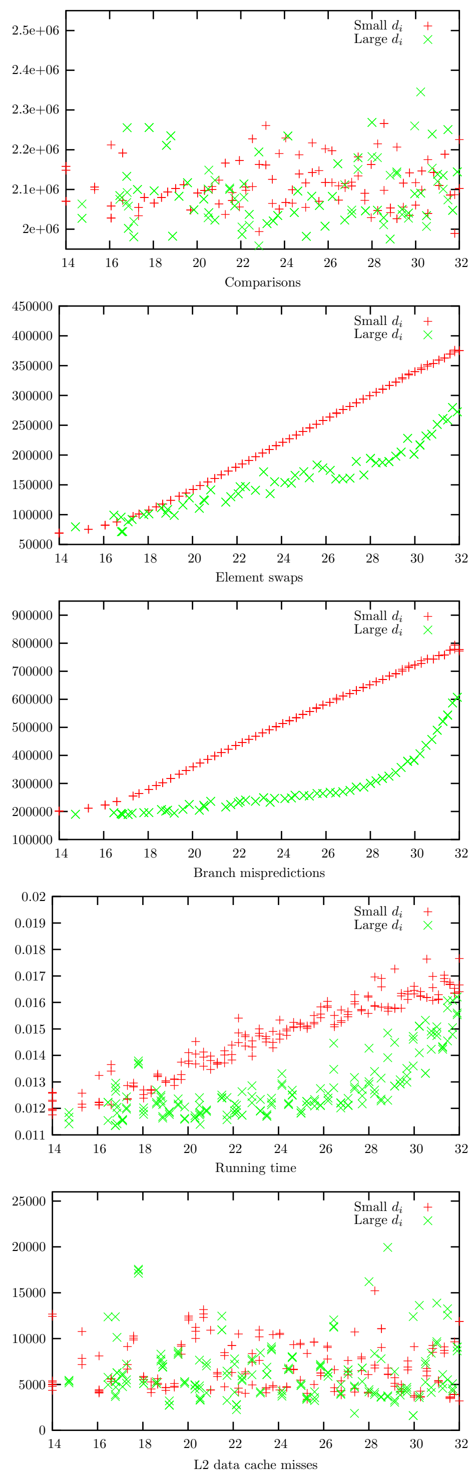

Figure 8: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 6 \times 10^4$. The x-axis shows log(Inv).

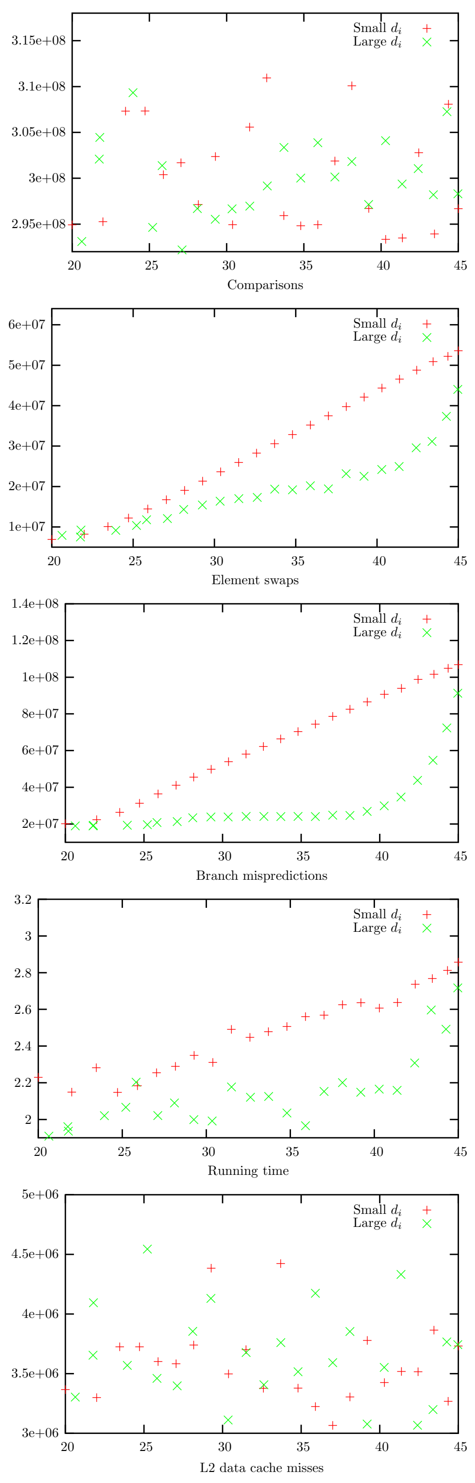

Figure 9: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 10^7$. The x-axis shows log(Inv).

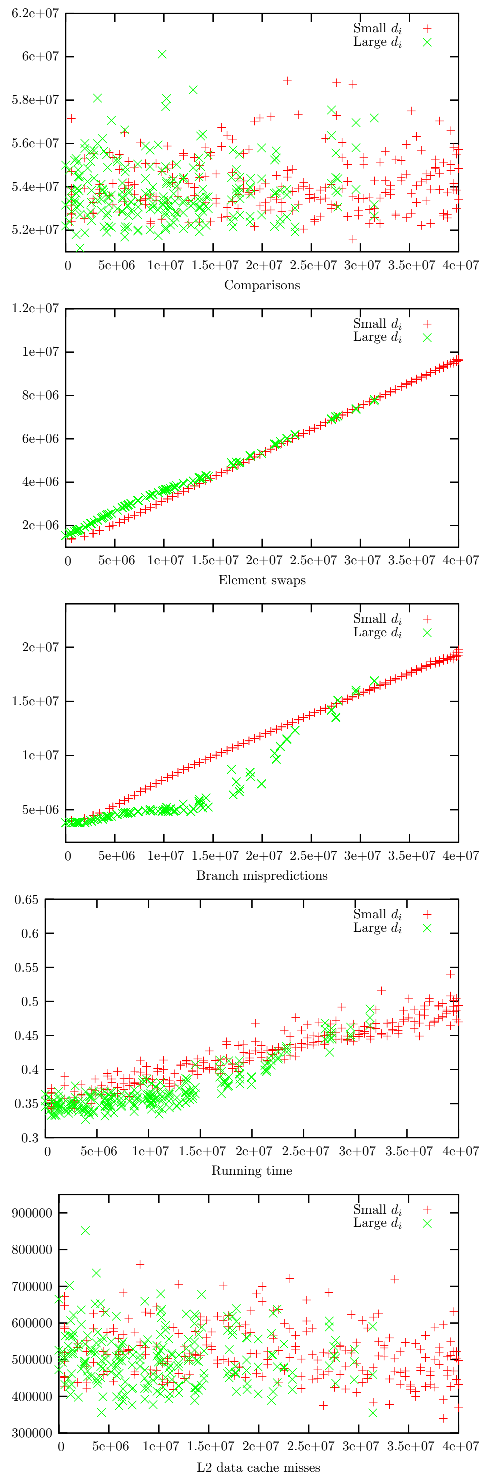

Figure 10: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for the input size $n = 2 \times 10^6$. The x-axis shows $\sum_{i=1}^{n} \log(d_i + 1)$.

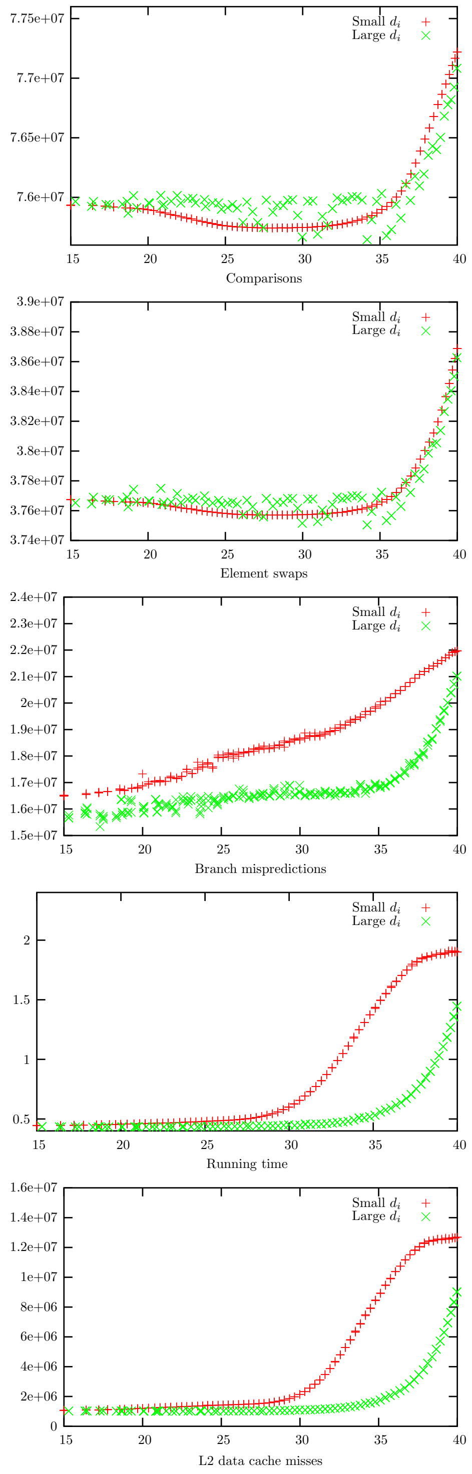

Figure 11: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by Heapsort on Athlon, for $n = 2 \times 10^6$. The x-axis shows log(Inv).

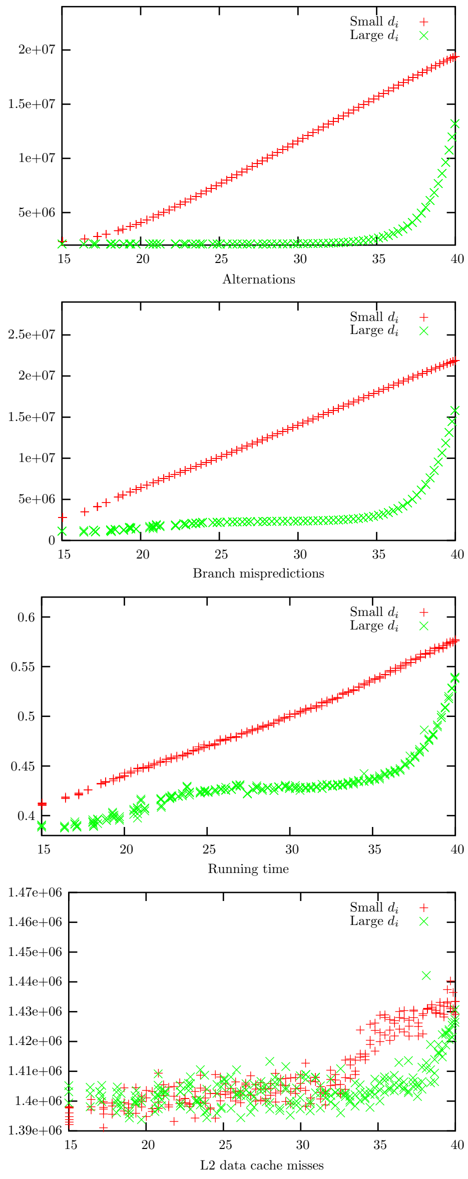

Figure 12: The number of alternations, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by Mergesort on Athlon, for $n = 2 \times 10^6$. The x-axis shows log(Inv).