On the Adaptiveness of Quicksort

Gerth Stølting Brodal
MADALGO, University of Aarhus, Denmark
Rolf Fagerberg
University of Southern Denmark, Denmark
Gabriel Moruz
J. W. Goethe University, Germany

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 that 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 for sorted inputs. 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 Quicksort. We also give some empirical results on the adaptive behavior of Heapsort and Mergesort.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms

Additional Key Words and Phrases: adaptive sorting, Quicksort, branch mispredictions

---

Gerth Stølting Brodal, MADALGO (Center for massive data algorithmics, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, IT Parken, Åbogade 34, DK-8200 Århus N, Denmark. Supported by the Carlsberg Foundation (contract number ANS-0257/20) and the Danish Natural Science Research Council (SNF).
Rolf Fagerberg, Department of Mathematics and Computer Science, University of Southern Denmark, Campusvej 55, DK-5230 Odense M, Denmark. Supported in part by the Danish Natural Science Research Council (SNF).
Gabriel Moruz, Institute for Computer Science, J. W. Goethe University, 60325 Frankfurt/Main, Germany. Partially supported by MADALGO (Center for massive data algorithmics, funded by the Danish National Research Foundation). Work done while at University of Aarhus.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

Quicksort was introduced by Hoare in 1961 as a simple randomized sorting algorithm [Hoare 1961a; 1961b]. Hoare proved that the expected number of comparisons performed by randomized Quicksort for a sequence of n elements is essentially $2n \ln n \approx 1.4n \log_2 n$ [Hoare 1962]. Many variants and analysis of the algorithm have later been given, including [Bentley and McIlroy 1993; Martínez and Roura 2002; Sedgewick 1975; 1977; 1978]. 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 [Estivill-Castro and Wood 1992]. An example of an adaptive sorting algorithm is insertion sort using level-linked B-trees with finger searches for locating each new insertion point [Mehlhorn 1984], 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 [Estivill-Castro and Wood 1992].

Most classic sorting algorithms, such as Quicksort, Mergesort [Knuth 1973], and Heapsort [Floyd 1964; Williams 1964], 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 [Mehlhorn 1984], has been developed over the last three decades. For an overview of this area, we refer the reader to the survey by Estivill-Castro and Wood [Estivill-Castro and Wood 1992]. Later work on adaptive sorting includes [Brodal et al. 2005; Brodal and Moruz 2005; Elmasry 2002; 2004; Elmasry and Fredman 2003; Pagh et al. 2004; Petersson and Moffat 1995].

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 Quicksort, 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 [Sanders and Winkel 2004], 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* $n + n \ln\left(\frac{2\mathrm{Inv}}{n} + 1\right)$.

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 [Franceschini and Geffert 2003].

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 [Seidel 1992, Section 5] concerning the expected number of comparisons performed by 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 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 stated in Fact 2.1, which states that an element x_i that has not yet been moved from its input position i is swapped during the execution of the partitioning procedure if and only if x_i is selected as pivot, or we have that $i \le \pi_j < \pi_i$ or $\pi_i < \pi_j \le i$, where x_j denotes the pivot element. This

```
#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;
  }
```

Fig. 1. C code for randomized Quicksort.

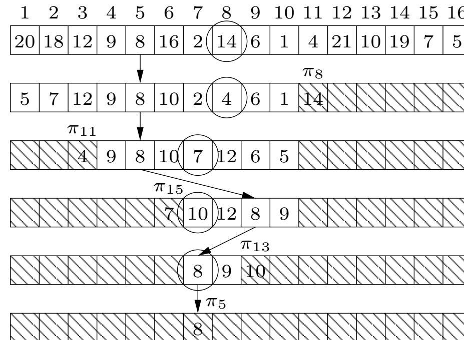


Fig. 2. The partitions involving element 8.

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. Without loss of generality assume $i < \pi_i$. If $i < \pi_i < \pi_j$ or $\pi_j \leq i < \pi_i$ then x_i is correctly placed with respect to the pivot and it is not swapped during the partitioning step.

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

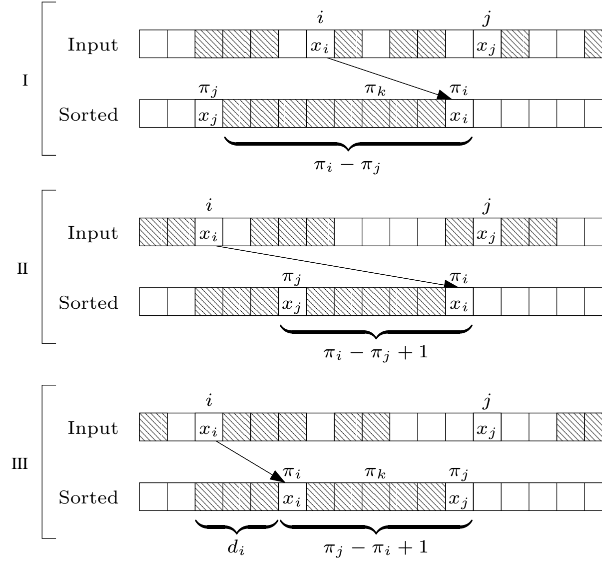


Fig. 3. The three different cases of Lemma 2.4.

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 2.2. $\text{Inv} \leq \sum_{i=1}^{n} d_i \leq 2\text{Inv}$.

PROOF. For the left inequality, $\text{Inv} \leq \sum_{i=1}^{n} d_i$, 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 $\text{Inv} \leq \sum_{i=1}^{n} d_i$.

For the right inequality, $\sum_{i=1}^{n} d_i \leq 2\text{Inv}$, 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 $\sum_{i=1}^{n} d_i \leq 2\text{Inv}$. □

The constants in Lemma 2.2 are the best possible. For even n, the sequence $(2, 1, 4, 3, 6, 5, \ldots, n, n-1)$ has $\text{Inv} = n/2$ 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 $\text{Inv} = n(n-1)/2$ 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 2.3. 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 inputs 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 2.4.

$$\Pr[X_{ij} = 1] \leq \begin{cases} 0 & \text{if } \pi_j < i \leq \pi_i \text{ or } \pi_i \leq i < \pi_j \,, \\ \frac{1}{|\pi_i - \pi_j| + 1} & \text{if } i \leq \pi_j < \pi_i \text{ or } \pi_i < \pi_j \leq i \,, \\ \frac{1}{|\pi_i - \pi_j| + 1} - \frac{1}{|\pi_i - \pi_j| + 1 + d_i} & \text{otherwise.} \end{cases}$$

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 $i \le \pi_i < \pi_j$, 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 $i \le \pi_k \le \pi_j$, 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 2.2 and Lemma 2.4 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

$$
\begin{aligned}
& \mathrm{E}\left[\sum_{j=1}^{n}\left(1 + \frac{1}{2}\sum_{i=1, i\ne j}^{n} X_{ij}\right)\right] \\
&= n + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1, i\ne j}^{n} \Pr(X_{ij} = 1) \\
&\le n + \frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{d_i}\frac{1}{k+1} + \sum_{k=1}^{\infty}\left(\frac{1}{k+1} - \frac{1}{k+1+d_i}\right)\right) && (1) \\
&\le n + \frac{1}{2}\sum_{i=1}^{n}\left(2\sum_{k=1}^{d_i}\frac{1}{k+1}\right) \\
&= \sum_{i=1}^{n}\sum_{k=1}^{d_i+1}\frac{1}{k} \\
&\le \sum_{i=1}^{n}(1 + \ln(d_i + 1)) && (2) \\
&\le n + n\ln\frac{\sum_{i=1}^{n}(d_i + 1)}{n} && (3) \\
&\le n + n\ln\left(\frac{2\mathrm{Inv}}{n} + 1\right) && (4)
\end{aligned}
$$

where (1) follows from Lemma 2.4, (2) follows from $\sum_{i=1}^{n}\frac{1}{i} \le 1 + \ln n$, (3) follows from the concavity of the logarithm function, and (4) follows from Lemma 2.2. □

It should be noted that the upper bound achieved in (3) using the concavity of the logarithm function can be much larger than the value (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) and (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. It should also be noted that in the case of many equal keys the running time of the code in Figure 1 can rise to as much as $O(n^2)$ and thus the assumption the the input consists of distinct keys is essential for our analysis.

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 and Mergesort are observed to improve by factors between 1.5 and 2 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 L1 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. We furthermore investigate different experimental setups for the Quicksort in Figure 1. We reduce the number of branch mispredictions by unrolling the inner loops three times and we study expensive comparisons by making the elements real and using a comparison function. Finally, we study the behavior of the classic sorting algorithm Mergesort, which also has an adaptive behavior.

The input elements are distinct 4 byte integers. We generate two types of input sequences consisting of distinct elements, having small d_i's and large d_i's, respectively. We generate the sequence with small d_i's by choosing each element x_i randomly in $[i-d,\ldots,i+d]$ for some parameter d, making sure it is different than its predecessors. The sequence with large d_i's is generated by letting $x_i = i$ with the exception of d random i's which are permuted in the input. 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 10^6, but we also investigate larger and smaller input sizes.

Our experiments are conducted on two different machines. The first machine has a Dual Core Intel P4 3.4 GHz CPU with 1 GB RAM, running linux 2.6.8, while the other has an AMD Athlon XP 2400+ 2.0 GHz CPU with 1 GB RAM, running linux 2.6.8. On the P4 the C source code was compiled using gcc-3.4.2, while on the AMD we used gcc-4.2.1, and in both cases we used optimization level -O3. The number of branch mispredictions and L1 data cache misses was obtained using the PAPI library [papi 2004].

Source code and the plotted data are available at `http://www.daimi.au.dk/~gabi/JEA05`.

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 $O(n(1+\log(1+\frac{\text{Inv}}{n})))$ 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). 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 the plot for the L1 data cache misses, where surprisingly the number of data cache misses decreases when the number of inversions in the input increases, by a factor of two. We conjecture that this behavior is given by the hardware prefetcher included in recent Pentium processors. The hardware prefetcher is a hardware optimization which prefectches the data required by the processor into the caches, thus without incurring cache misses. For Quicksort, we first note that regardless of the direction of the branches in the inner loop of the partitioning step, the data to be perfetched is the same. When the number of inversions is small, the number of branch mispredictions is also small and the prefetcher can not bring data into cache as fast as it is being processed, hence the number of L1 data cache misses is roughly the same as for the AMD. However, when the number of inversions increases, the number of branch mispredictions increases too, and this triggers a significant number of CPU cycles to be used for filling the instruction pipeline emptied for each branch misprediction, allowing more time for the hardware prefetcher to load the data, thus preventing the occurrence of a number of data cache misses.

We note that the remaining plots follow the same trends as in Figure 4. On the P4, the number of comparisons and the number of element swaps are approximately the same as on the Athlon, but the running time varies by more than 200% on P4 and only 45% on Athlon. A likely reason for this behavior is that the length of the pipeline is shorter for Athlon, and thus emptying it upon branch mispredictions is less costly than on P4.

Similar observations on the resemblance between the data for the two architec-

tures apply to all our experiments. For this reason, and because the effects of branch mispredictions over the running time are less obvious, 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 L1 data cache misses for the version of Quicksort that chooses as a 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 15% less comparisons, due to the better choice of the pivot. This immediately triggers a slight improvement in the number of data cache misses. Although the number of element swaps remains approximately the same, the number of branch mispredictions increases due to the extra branches used for computing the median of three elements. Also, the running time increases because of the increased number of branch mispredictions and random number generations.

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 [Hoare 1962]. The good pivot choice for small disorder in the input also triggers a smaller number of 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 almost 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 $n = 10^6$.

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) 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.

We now study different variants of the randomized version of Quicksort in Figure 1. We first enforce expensive comparisons in two ways, by sorting real elements and using a comparison function respectively. To attempt reducing the number of branch mispredictions, we also unroll the inner loops three times. We first show in Figures 13 and 14 that, for both architectures considered, loop unrolling does not help in reducing the number of branch mispredictions for inputs having small d_i's as well as for inputs having large d_i's. This happens because the branches comparing input elements in the inner loop of the partitioning step are virtually impossible to predict since their outputs does not exhibit any particular pattern. Therefore, standard techniques for reducing the number of branch mispredictions, such as loop unrolling, do not yield any noticeable results. In what concerns the running time, we show in Figures 15 and 16 that for both architectures the running

time increases significantly when sorting real elements, but is approximately the same in the case of sorting integers, using a comparison function, and unrolling the inner loops. Also, the shape of all these charts is similar and is consistent with the result in Theorem 1.1.

4.2 Mergesort.

We briefly demonstrate that also for Mergesort the actual running time varies with the number of inversions in the input. 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 [Moffat et al. 1992] based on merging by finger search trees, which was proved to have a running time of $O(n(1 + \log(1 + \frac{\text{Inv}}{n})))$, i.e. the number of alternations by standard Mergesort is $O(n(1 + \log(1 + \frac{\text{Inv}}{n})))$. 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 does not exhibit any clear variation when the number of inversions in the input changes, but for large d_i's there seem to occur slightly more misses.

5. CONCLUSIONS AND RELATED WORK

In this paper we demonstrate that, in spite of common knowledge, the running time of the randomized version of Quicksort is adaptive with respect to measure Inv. Even though the expected number of comparisons is $O(n \log n)$, we prove that the expected number of element swaps is $O(n(1+\log(1+\text{Inv}/n)))$. Furthermore, we demonstrate experimentally that the number of element swaps performed follows closely the number of branch mispredictions, which are an important factor affecting the running time when computation takes place in internal memory. We observe that Mergesort has an adaptive behavior too.

Elmasry and Hammad [Elmasry and Hammad 2005] gave an empirical study for optimal algorithms with respect to Inv, and compare these algorithms against Quicksort. For Quicksort they measure the number of comparisons and the running time, obtaining results that are consistent to ours. They demonstrate that, for a low number of inversions, Quicksort is outperformed by some other algorithms, but its running time is still competitive. On the other hand, when the input sequence has a high Inv value, Quicksort outperforms all the Inv optimal algorithms considered.

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 L1 data cache misses for input sequences of constant length $n = 10^6$, on the Athlon architecture. The number of comparisons and the number of element swaps performed by Heapsort is affected slightly, while the number of branch mispredictions is affected in a more significant way, by almost 40%. The number of L1 data cache misses is also greatly affected, and varies by more than a factor of eight. The running time shows a virtually identical behavior with the data cache misses, 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.

An interesting sorting algorithm to be considered for study is Shellsort, introduced by Shell in [Shell 1959] and improved over the years (see [Sedgewick 1996] for a comprehensive survey). Since it is based on Insertionsort, we expect Shellsort to outperform some optimal sorting algorithms for a very small number of inversions, because of a very small number of comparisons and branch mispredictions. Intuitively, Insertionsort performs $O(n)$ branch mispredictions, because the branch testing the element to be inserted against some element in the sequence should be correctly predicted with one exception, when the element gets inserted.

REFERENCES

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

Brodal, G. S., Fagerberg, R., and Moruz, G. 2005. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming.* Lecture Notes in Computer Science, vol. 3580. Springer Verlag, Berlin, 576–588.

Brodal, G. S. and Moruz, G. 2005. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *Proc. 9th International Workshop on Algorithms and Data Structures.* Lecture Notes in Computer Science, vol. 3608. Springer Verlag, Berlin, 385–395.

Elmasry, A. 2002. Priority queues, pairing, and adaptive sorting. In *29th Annual International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 2380. Springer Verlag, Berlin, 183–194.

Elmasry, A. 2004. Adaptive sorting with avl trees. In *3rd IFIP International Conference on Theoretical Computer Science.* 307–316.

Elmasry, A. and Fredman, M. L. 2003. Adaptive sorting and the information theoretic lower bound. In *20th Annual Symposium on Theoretical Aspects of Computer Science.* Lecture Notes in Computer Science, vol. 2607. Springer Verlag, Berlin, 654–662.

Elmasry, A. and Hammad, A. 2005. An empirical study for inversions-sensitive sorting algorithms. In *4th International Workshop on Experimental and Efficient Algorithms.* 597–601.

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

Floyd, R. W. 1964. Algorithm 245: Treesort3. *Communications of the ACM 7,* 12, 701.

Franceschini, G. and Geffert, V. 2003. 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.* 242–250.

Hoare, C. A. R. 1961a. Algorithm 63: Partition. *Commun. ACM 4,* 7, 321.

Hoare, C. A. R. 1961b. Algorithm 64: Quicksort. *Commun. ACM 4,* 7, 321.

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

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

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

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

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

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

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

Petersson, O. and Moffat, A. 1995. A framework for adaptive sorting. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science 59*, 152–179.

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

Sedgewick, R. 1975. Quicksort. Ph.D. thesis, Stanford University, Stanford, CA. Stanford Computer Science Report STAN-CS-75-492.

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

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

Sedgewick, R. 1996. Analysis of shellsort and related algorithms. In *Proc. 4th European Symposium on Algorithms*. 1–11.

Seidel, R. 1992. Backwards analysis of randomized geometric algorithms. Tech. Rep. TR-92-014, International Computer Science Institute, Univeristy of Calfornia at Berkeley. February.

Shell, D. L. 1959. A high-speed sorting procedure. *Communications of the ACM 2,* 7, 30–32.

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

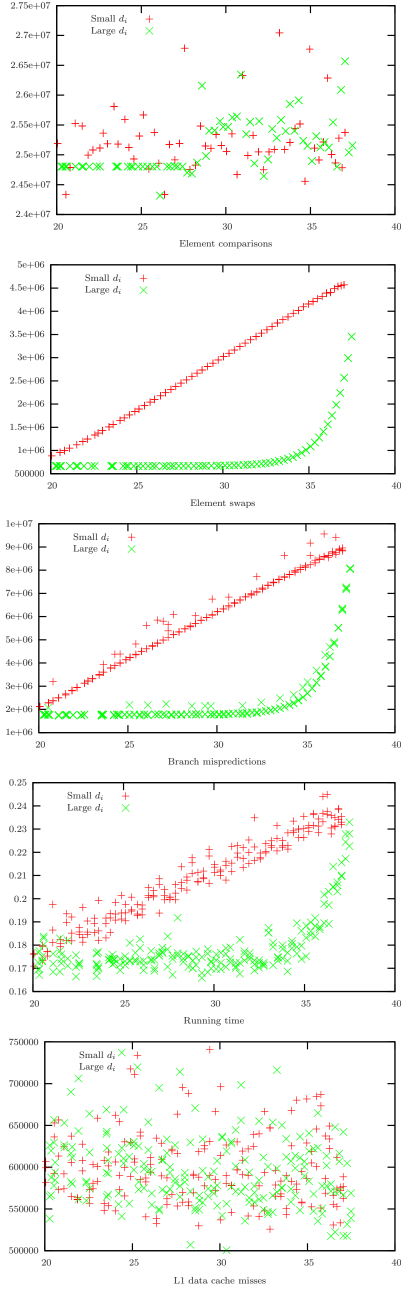


Fig. 4. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by randomized Quicksort on Athlon, for $n = 10^6$. The x-axis shows log(Inv).

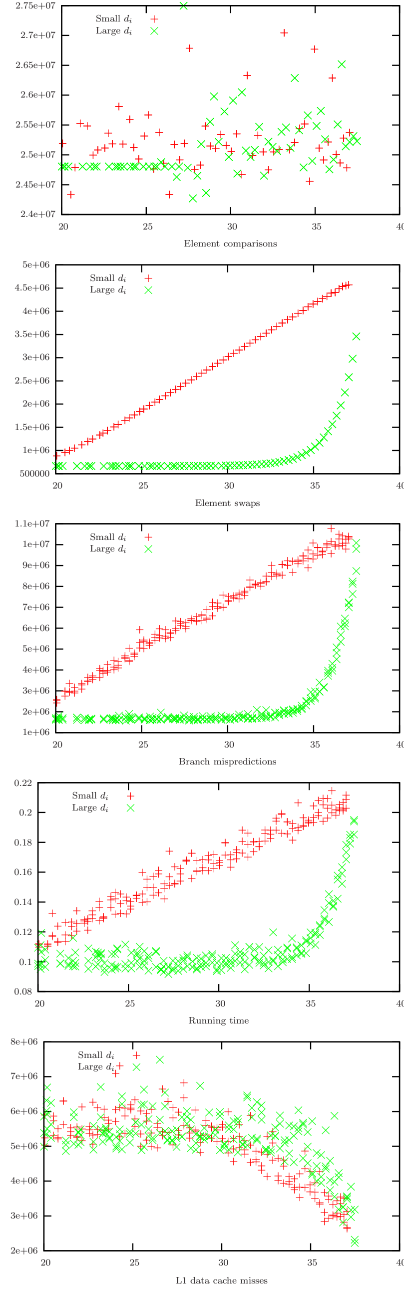
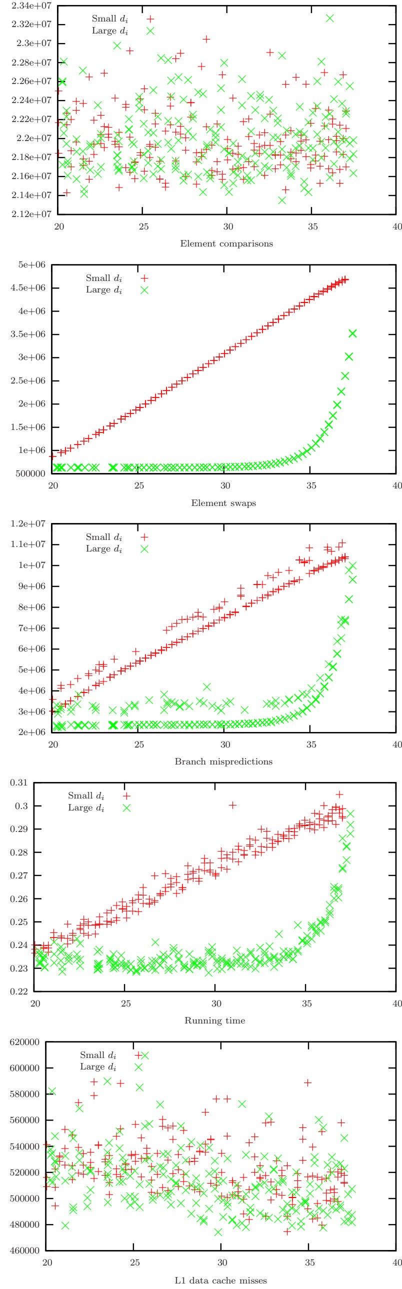


Fig. 5. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses of randomized Quicksort on P4, for $n = 10^6$. The x-axis shows log(Inv).



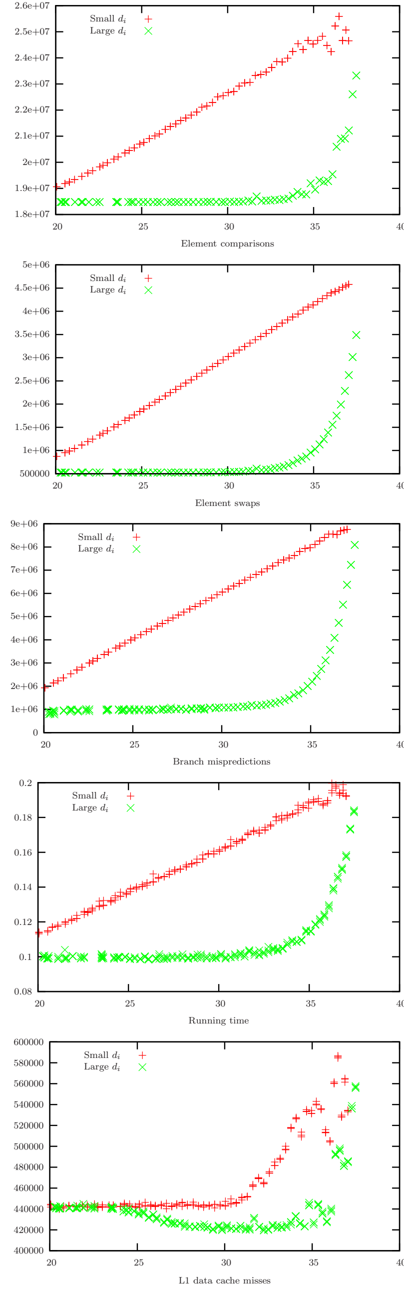


Fig. 6. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by randomized median-of-three Quicksort on Athlon, for $n = 10^6$. The x-axis shows log(Inv).

Fig. 7. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by deterministic Quicksort on Athlon, for $n = 10^6$. The x-axis shows log(Inv).

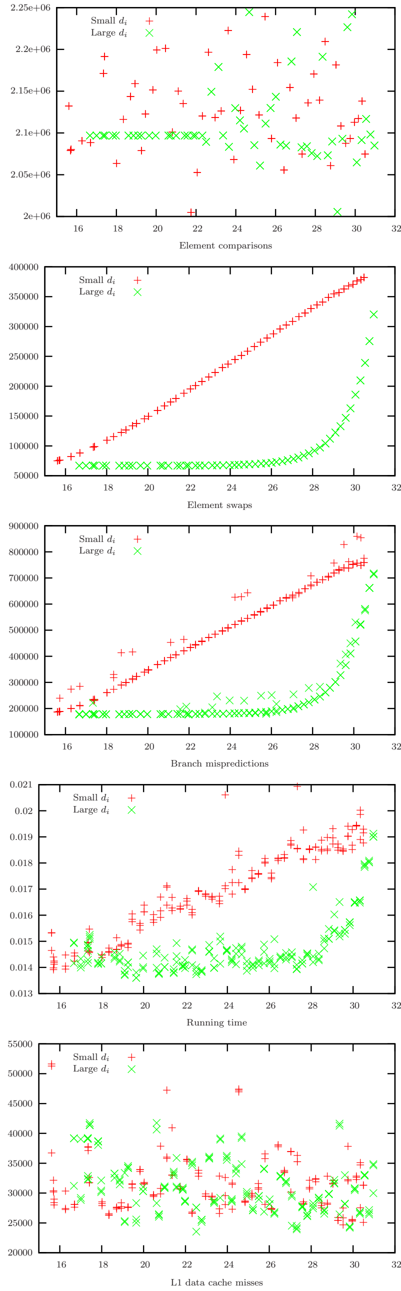


Fig. 8. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by randomized Quicksort on Athlon, for $n = 10^5$. The x-axis shows log(Inv).

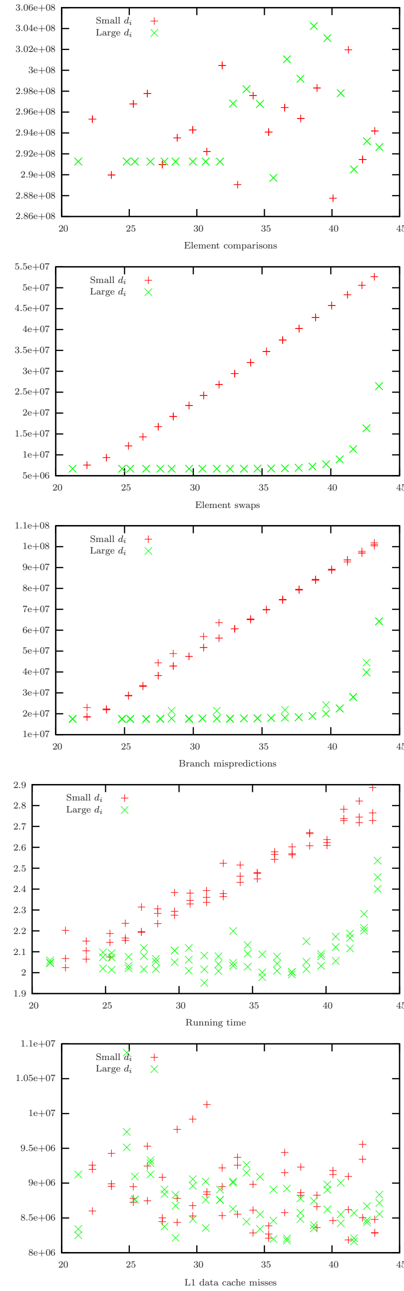


Fig. 9. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by randomized Quicksort on Athlon, for $n = 10^7$. The x-axis shows log(Inv).

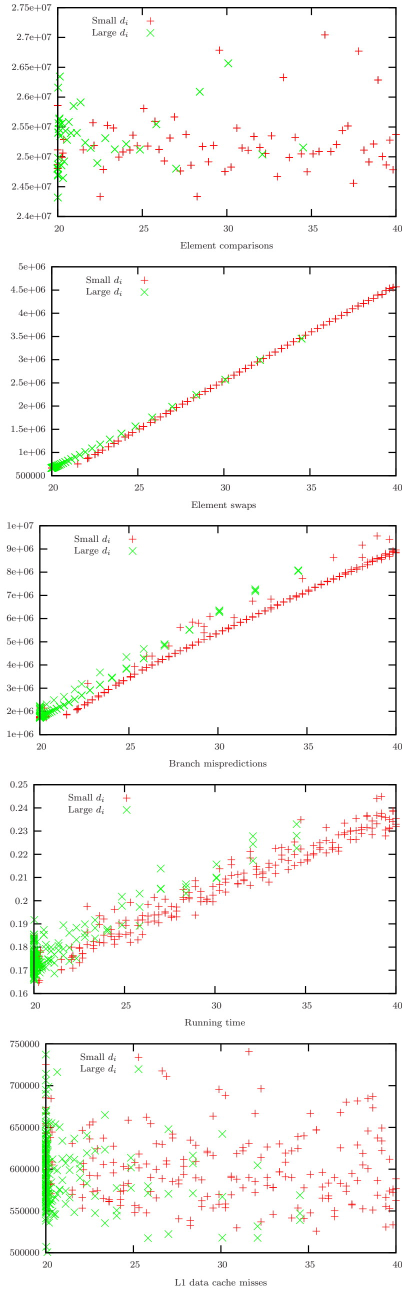


Fig. 10. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for the input size $n = 10^6$. The x-axis shows $\sum_{i=1}^{n} \log(d_i + 1)$.

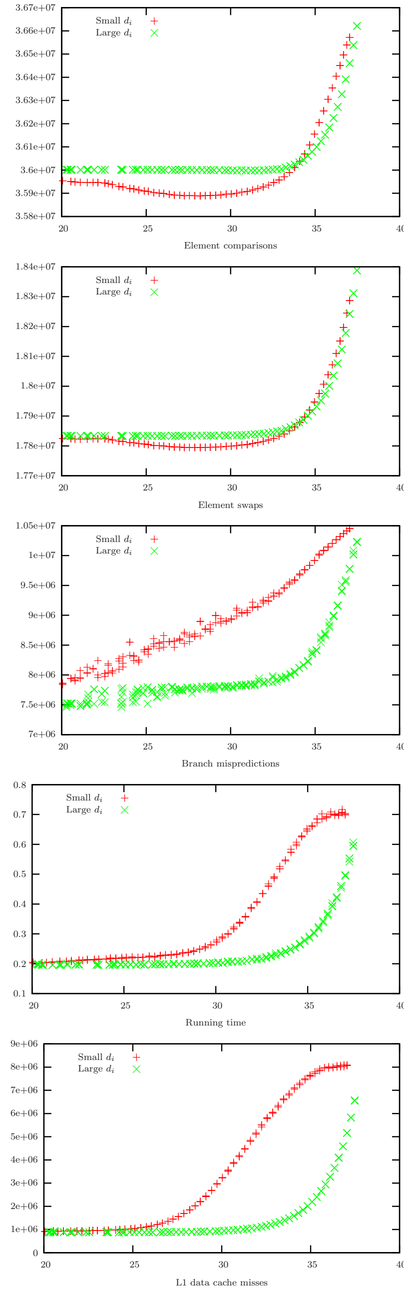


Fig. 11. The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by Heapsort on Athlon, for $n = 10^6$. The x-axis shows log(Inv).

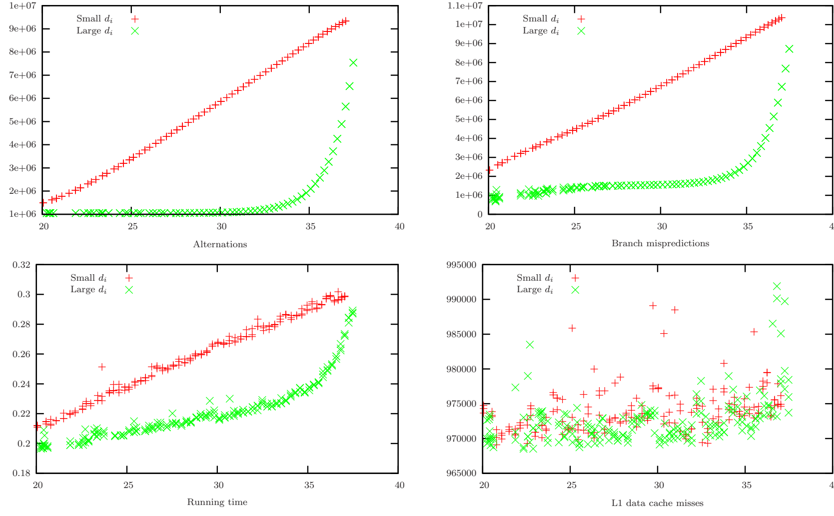


Fig. 12. The number of alternations, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by Mergesort on Athlon, for $n = 10^6$. The x-axis shows log(Inv).

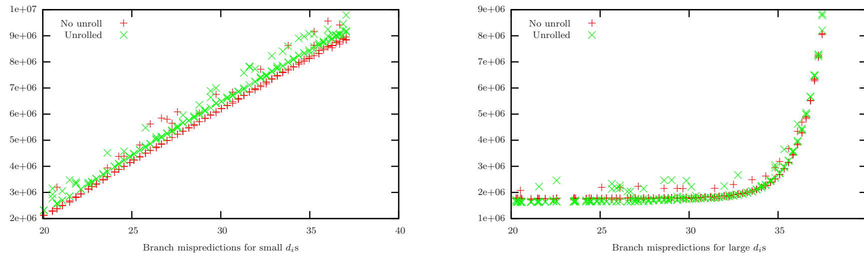


Fig. 13. The number of branch mispredictions performed by Quicksort on Athlon for small d_i's (left) and large d_i's (right), with no loop unrolling and with three unrolls of the inner loops. The input size is $n = 10^6$, and the x-axis shows log(Inv).

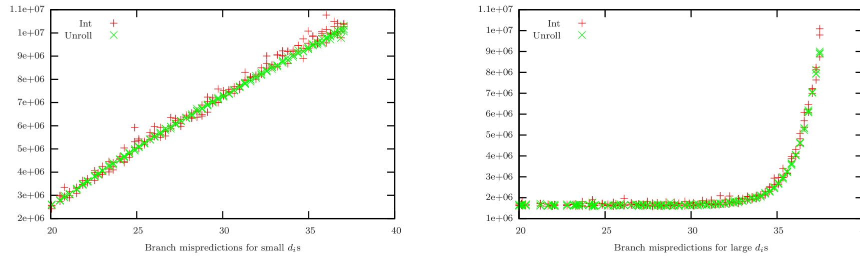


Fig. 14. The number of branch mispredictions performed by Quicksort on P4 for small d_i's (left) and large d_i's (right), with no loop unrolling and with three unrolls of the inner loops. The input size is $n = 10^6$, and the x-axis shows log(Inv).

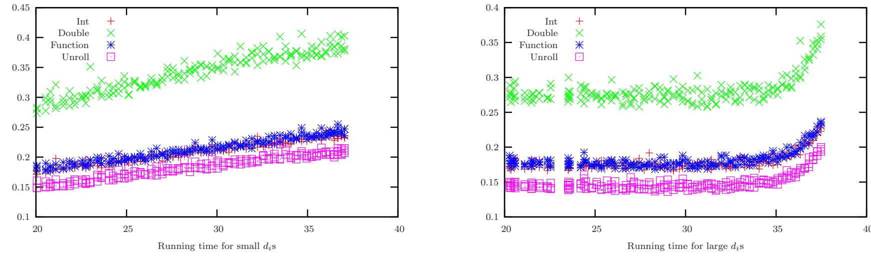


Fig. 15. The running time performed by several variants of Quicksort on Athlon for small d_i's (left) and large d_i's (right). The input size is $n = 10^6$, and the x-axis shows log(Inv).

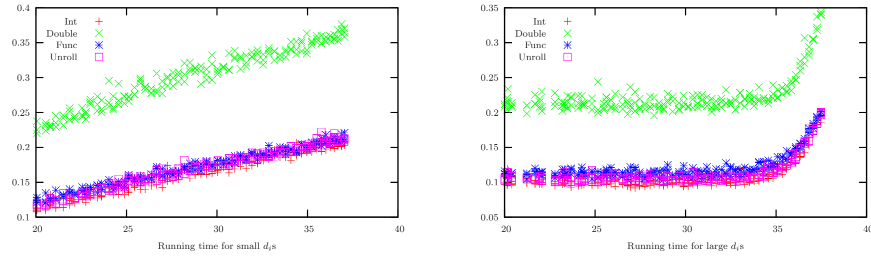


Fig. 16. The running time performed by several variants of Quicksort on P4 for small d_i's (left) and large d_i's (right). The input size is $n = 10^6$, and the x-axis shows log(Inv).