

An Experimental Study of Sorting and Branch Prediction

PAUL BIGGAR¹, NICHOLAS NASH¹, KEVIN WILLIAMS² and DAVID GREGG
Trinity College Dublin

Sorting is one of the most important and well studied problems in Computer Science. Many good algorithms are known which offer various trade-offs in efficiency, simplicity, memory use, and other factors. However, these algorithms do not take into account features of modern computer architectures that significantly influence performance. Caches and branch predictors are two such features, and while there has been a significant amount of research into the cache performance of general purpose sorting algorithms, there has been little research on their branch prediction properties. In this paper we empirically examine the behaviour of the branches in all the most common sorting algorithms. We also consider the interaction of cache optimization on the predictability of the branches in these algorithms. We find insertion sort to have the fewest branch mispredictions of any comparison-based sorting algorithm, that bubble and shaker sort operate in a fashion which makes their branches highly unpredictable, that the unpredictability of shellsort's branches improves its caching behaviour and that several cache optimizations have little effect on mergesort's branch mispredictions. We find also that optimizations to quicksort – for example the choice of pivot – have a strong influence on the predictability of its branches. We point out a simple way of removing branch instructions from a classic heapsort implementation, and show also that unrolling a loop in a cache optimized heapsort implementation improves the predictability of its branches. Finally, we note that when sorting random data two-level adaptive branch predictors are usually no better than simpler bimodal predictors. This is despite the fact that two-level adaptive predictors are almost always superior to bimodal predictors in general.

Categories and Subject Descriptors: E.5 [Data]: Files—*Sorting/Searching*; C.1.1 [Computer Systems Organization]: Processor Architectures, Other Architecture Styles—*Pipeline processors*

General Terms: Algorithms, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Sorting, Branch Prediction, Pipeline Architectures, Caching

1. MOTIVATION

Classical analyses of algorithms make simplifying assumptions about the cost of different machine instructions. For example, the RAM model used for establishing

¹Supported by the Irish Research Council for Science, Engineering and Technology (IRCSET).

²Supported by the Irish Research Council for Science, Engineering and Technology (IRCSET) and IBM.

Corresponding author's address: David Gregg, Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. David.Gregg@cs.tcd.ie.

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

asymptotic bounds and Knuth's MIX machine code both make drastically simplifying assumptions about the cost of machine instructions [Knuth 1997]. More recently researchers have recognized that on modern computers the cost of accessing memory can vary dramatically depending on whether the data can be found in the first-level cache, or must be fetched from a lower level of cache or even main memory. This has spawned a great deal of research on cache-efficient searching and sorting [Nyberg et al. 1994; Agarwal 1996; LaMarca and Ladner 1996; LaMarca 1996; LaMarca and Ladner 1997; Xiao et al. 2000; Rahman and Raman 2001; Wickremesinghe et al. 2002].

Another type of instruction whose cost can vary dramatically is the conditional branch. Modern pipelined processors depend on *branch prediction* for much of their performance. If the direction of a conditional branch is correctly predicted ahead of time, the cost of the conditional branch may be as little as the cost of, say, an integer add instruction. If, on the other hand, the branch is mispredicted the processor must flush its pipeline, and restart from the correct target of the branch. This cost is typically a large multiple of the cost of executing a correctly-predicted branch. For example, Intel Pentium 4 processors [Intel 2004; 2001] have pipelines of up to 31 stages, meaning that a branch misprediction will cost around 30 cycles. Fortunately, the branches in most programs are very predictable, so branch mispredictions are usually rare. Indeed, prediction accuracies of greater than 90% are typical [Uht et al. 1997] with the best predictors.

The cost of executing branches is particularly important for sorting because the inner-loops of most sorting algorithms consist of comparisons of items to be sorted. Thus, the predictability of these comparison branches is critical to the performance of sorting algorithms. In this paper we study the predictability of branches in most of the major sorting algorithms. We focus on the behaviour of the branches whose outcome depends on a comparison of keys presented to the sorting algorithm in its input. Throughout this paper we refer to such branches as *comparison branches*. Branches associated with controlling simpler aspects of the control flow of the algorithms are of much less interest because they are generally almost perfectly predictable.

We also examine cache conscious variations of many of the algorithms, and show how optimizations for the cache influence the level of branch mispredictions. The cache conscious algorithms whose branch misprediction characteristics we examine are all cache aware algorithms. We do not consider sorting algorithms designed in the cache oblivious model of Frigo *et al* [1999], nor sorting algorithms designed in the external-memory model of Aggarwal and Vitter [1988].

2. BRANCH PREDICTION

Branches are a type of instruction used for defining the flow control of programs. In high level languages they result from the use of statements like `if`, `while` and their variants. Branches can be either *taken*, indicating that the address they provide is the new value for the program counter, or they can be *not-taken*, in which case sequential execution continues as though the branch had not been present.

Branch instructions pose a difficulty for pipelined processors. In a pipelined processor, while the program counter points to a particular instruction, a potentially

large number of subsequent instructions are in a partially completed state. Until it is known whether a branch is taken or not-taken, it is not possible to know what the next instructions to be executed are. If the processor cannot tell what these next instructions are, it cannot fill its pipeline. To keep the utilization of the pipeline at a reasonable level, modern processors attempt to anticipate the outcome of branch instructions by employing *branch predictors*. All modern general purpose processors that we are aware of employ some form of branch prediction technique. A good discussion of pipelining and its associated issues, including the branch prediction techniques we now describe, can be found in Hennessy and Patterson [1990].

There are several kinds of branch predictors: static, semi-static and dynamic. Static branch predictors always predict the same direction for a branch each time it is executed. A typical heuristic is to predict forward branches not-taken and backwards branches taken. Semi-static branch predictors support a *hint* bit which allows the compiler to determine the prediction direction of the branch. We use only dynamic predictors in our experiments, because they are most commonly used in real processors. Moreover, static predictors, aside from being less common in modern mainstream processors, are in most cases trivial to analyse compared to the more realistic case of dynamic predictors.

The statistics in the following paragraphs are taken from those derived by Uht *et al* [1997], over the SPECint92 benchmarking suite.

The simplest type of dynamic predictor is referred to as a *1-bit* predictor. It keeps a table recording, at each entry, whether a particular branch was taken or not-taken. Using the table, it predicts that a branch will go the same way as it went on its previous execution. 1-bit predictors achieve 77% to 79% accuracy.

We refer to a 2-bit dynamic predictor as a *bimodal* predictor. A bimodal predictor operates in the same manner as a 1-bit predictor but each table entry can be thought of as maintaining a counter from 0 to 3. The counter decrements on each taken branch (with the exception of when the count is 0), and increments on each not-taken branch (with the exception of when the count is 3). With counts equal to 0 or 1 the next branch is predicted as taken, with counts of 2 or 3 the branch is predicted not-taken. When the counter is 0, we say the predictor is in the *strongly taken* state, since the branch must be not-taken twice before the not-taken direction will be predicted again. When the counter is 1 we say the predictor is in the *taken* state. Similarly counts of 2 and 3 are referred to as *not-taken* and *strongly not-taken* respectively. Bimodal predictors achieve 78% to 89% accuracy.

Finally, some branch predictors attempt to exploit correlations in branch outcomes to improve accuracy. A *two-level adaptive* predictor maintains a *branch history register*. This register records the outcomes of a number of previous branch instructions. For example, a register contents of 110010 indicates that the 6 previous branch outcomes were taken, taken, not-taken, not-taken, taken and not-taken. For each branch, this register is used to index a table of bimodal predictors. The program counter can also be included in the index, either concatenated or XORed with the indexing register. Two-level adaptive predictors are accurate about 93% of the time.

In all the above, due to the size of the predictor table branches can collide. That is, two different branches can map to the same table location, often reducing

accuracy.

3. EXPERIMENTAL SETUP

We used truly random data provided by Haahr [2006] in all our experiments. We used ten chunks of random data containing 2^{22} (4194304) keys. Where a particular experiment used only part of the keys in a chunk, it used the left-most keys. Our results are averaged over these chunks of data.

In order to experiment with a variety of cache and branch prediction results we used the SimpleScalar PISA processor simulator version 3 [Austin et al. 2001]. We used `sim-cache` and `sim-bpred` to generate results for caching and branch prediction characteristics respectively.

We used a variety of cache configurations; generally speaking we used an 8 KB level 1 data cache, 8 KB level 1 instruction cache, and a shared 2 MB instruction and data level 2 cache, all with 32 byte cache lines. We also experimented with both direct mapped and fully associative caches. When presenting results where the cache configuration is important, we will describe the precise configuration used. We present results here only for direct mapped caches. The results for fully associative caches are similar, as we show in our technical report [Biggar and Gregg 2005], which contains complete results for a large number of branch predictor and cache configurations.

For the measurements taken using SimpleScalar, we used power of two sized sets of random keys ranging from 2^{12} to 2^{22} keys in size. This is with the exception of the quadratic time sorting algorithms, for which the maximum set size used was 2^{16} keys. This was done because of the very significant time it takes such algorithms to sort large inputs.

It takes time to fill arrays, and this can distort the relationship between the results for small set sizes, and larger set sizes, for which this time is amortised. As a result, we measured this time, and subtracted it from our results. However, this practice also distorts the results. It removes compulsory cache misses from the results, so long as the data fits in the cache. When the data does not fit in the cache, the keys from the start of the array are faulted, with the effect that capacity misses reoccur, which are not discounted.

For branch prediction we simulated both bimodal and two-level adaptive predictors. The simple bimodal predictor provides a good base-line to compare the two-level adaptive predictor against. We used power of two sized tables for the branch predictors. We used tables containing between 2^{11} and 2^{14} predictors, because these are close to the size of the 2^{12} entry table of the Pentium 4 [Hinton et al. 2001]. When presenting branch prediction results, the predictor configurations will be described precisely. SimpleScalar's `sim-bpred` provides total results over all the branches in a program. It is often desirable to examine particular branches individually however. In order to do so, we added our own simulations of branch predictors to the programs. For our own simulations we average the results of each branch over ten runs of random data containing 2^{22} keys.

Finally we used *PapiEx* [Mucci 2004] to access performance counters on a Pentium 4 1.6 Ghz, with 1 GB of memory. Note that this means our data sets always fit inside main memory, even for the out-of-place sorting algorithms we describe

below. The hardware performance counters allow cycle counts (i.e. running times) to be determined for the sorting algorithms on a real processor. Note that these cycle counts include the stall cycles resulting from branch mispredictions and cache misses. The results presented from these tests are averaged over 1024 runs and use the system's random number generator to provide data. The data ranged in size from 2^{12} to 2^{22} keys in size. The processor we used had an 8-way associative 256KB level 2 cache with 64-byte cache lines. The separate data and instruction level 1 caches are 4-way associative, 8KB in size and have 32-byte cache lines. The Pentium 4 uses an unspecified form of dynamic branch prediction. A reference manual [Intel 2004] mentions a branch history register, so it is probable that the scheme is some variety of to two-level adaptive.

4. ELEMENTARY SORTS

We begin by analysing the branch prediction behaviour of a number of common elementary sorting algorithms, each operating in $O(n^2)$ time. The main practical advantage of these algorithms is their simplicity, which generally allows them to operate faster than algorithms with $O(n \log n)$ time complexity for small inputs. Indeed, as we will see later, optimized versions of $O(n \log n)$ sorts use calls to elementary sorts for small inputs. Knuth [1998] provides a good exposition of these elementary sorting algorithms (as well as basic versions of the sorting algorithms we shall examine in later sections).

4.1 Selection Sort

Selection sort is a very simple quadratic time sorting algorithm which begins by sequentially searching for the smallest key in its input and moving it into position. The algorithm continues by repeatedly searching for and then moving the smallest key in the unsorted section of its input into the appropriate position. The inner-loop of selection sort, which is run a total of $n - 1$ times on an input `a[0..n - 1]` is shown below

```

min = i;
for(j = i + 1; j < n; j++)
    if(a[j] < a[min]) min = j;
swap(a[i], a[min]);

```

It is easy to see that selection sort performs approximately $n^2/2$ comparisons and exactly $n - 1$ exchanges. On the j^{th} iteration of the inner-loop the comparison branch is taken if `a[j]` is the minimum of `a[i..j]`, which occurs with probability $1/(j - i + 1)$ ¹. The expected number of times the branch is taken on the first iteration is thus given by $H_n - 1$ where $H_n = \sum_{i=1}^n 1/i$ is the n^{th} harmonic number [Knuth 1997]. Since $H_n \approx \ln n$, and $\sum_{i=1}^n H_i = O(n \log n)$ the comparison branch will be highly predictable. If the array is initially in sorted order, then the value of `min` will never change in the inner-loop. Since this implies the comparison branch is never taken, it will be almost perfectly predictable. Similarly, if the array is sorted in reverse order, the value of `min` must change at every element. That is, the

¹Throughout this paper, for the purpose of analysis we make the usual assumption that the input to be sorted consists of a random permutation of $\{1, \dots, n\}$.

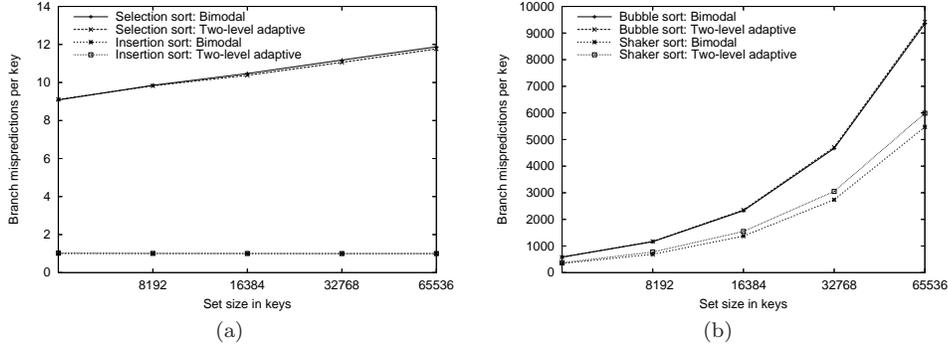


Fig. 1. (a) Shows the branch mispredictions for selection and insertion sort using bimodal and two-level adaptive predictors both having 4096 table entries. It is noteworthy the simple bimodal predictor has essentially identical performance to the two-level adaptive predictors. (b) Shows the branch mispredictions for bubble and shaker sort using bimodal and two-level adaptive predictors, again both having 4096 table entries. For bubble sort the bimodal and two-level adaptive predictors have close to identical performance. However, for shaker sort the simple bimodal predictor results in fewer mispredictions than the two-level adaptive predictor. We also note that as a result of the partial sorting performed by bubble and shaker sort they incur far more branch mispredictions per key than other $O(n^2)$ sorting algorithms. `sim-bpred` was used to simulate the branch predictors in both (a) and (b). The two-level adaptive predictors used 10-bit history registers.

comparison branch is taken at every element, and the branch is therefore almost perfectly predictable. From the point of view of branch prediction, frequent changes (preferably with no simple pattern) are the main indicator of poor performance, rather than the absolute number of times that the branch resolves in each direction. The worst case for selection sort would be an array which is partially sorted in reverse order; the value of `min` would change frequently, but not predictably so. Figure 1(a) shows the behaviour of selection sort’s branches.

4.2 Insertion Sort

Insertion sort is another simple algorithm with quadratic worst case time. It works by iterating the inner-loop below a total of $n - 1$ times over its input

```

item = a[i];
while(item < a[i - 1])
{
    a[i] = a[i - 1];
    i--;
}
a[i] = item;

```

On the i^{th} iteration `a[0..i - 1]` is sorted, allowing `a[i]` to be placed in the correct position as shown in the inner-loop above. On average, insertion sort performs approximately $n^2/4$ comparisons and $n^2/4$ assignments in total [Knuth 1998]. Thus insertion sort performs around half as many branches as selection sort.

A very pleasing property of insertion sort is that it generally causes just a single branch misprediction per key, as Figure 1(a) shows. Typically the inner `while`

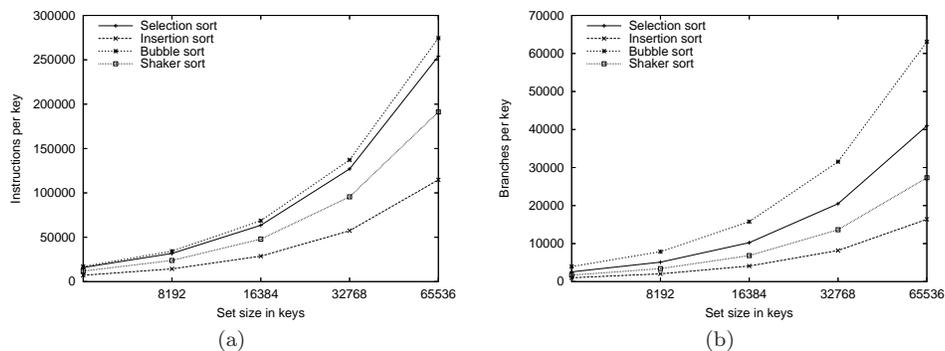


Fig. 2. (a) Shows the number of instructions executed per key by the elementary sorting algorithms, while (b) shows the number of branches executed per key by the algorithms. As with other results, these measurements were gathered using SimpleScalar. Shaker-sort executes fewer instructions in total than selection sort, as (a) shows, executes fewer branch instructions than selection sort, as (b) shows, and also has better cache performance, as Figure 4(a) shows. Despite this it is significantly slower than selection sort, as shown in Figure 4(b) due to its very high level of branch mispredictions, shown in Figure 1(b).

loop exit causes this misprediction, when the correct location for the current item has been found. In fact, a static predictor would perform just as well as the two predictors shown, since it would mispredict exactly once per key.

4.3 Bubble sort

Bubble sort is another simple sorting algorithm, although notoriously inefficient. It works by iterating the loop shown below at most $n - 1$ times on an input $a[0..n - 1]$.

```

for(j = 0; j < n - i - 1; j++)
{
    if(a[j + 1] < a[j])
        swap(a[j + 1], a[j]);
}

```

In this code i is the outer-loop iteration counter, and begins at 0 and counts towards $n - 1$. Bubble sort uses information yielded by the comparisons which selection sort simply discards. After a full iteration of the inner-loop shown larger elements have moved to the right, closer to their final location.

Shaker sort is a variation of bubble sort in which there are two inner-loops which are alternated. One is the inner-loop shown above. The other inner-loop scans right-to-left moving small elements to the left in the same manner as moving larger elements to the right has just been described. Although shaker sort has the same worst-case time as bubble sort, it is generally more efficient in practice.

Figure 1(b) shows the number of branch mispredictions per key for bubble and shaker sort. As the data set gets large, bubble sort causes nearly 10000 branch mispredictions per key. Shaker sort executes a lot fewer branches (see Figure 2(b)), and so has fewer mispredictions, but the rate of misprediction is similar.

To understand the high misprediction rate of bubble sort we define Q_l^k as the probability that l^{th} inner-loop comparison branch is taken on the k^{th} outer-loop iteration, $1 \leq k < n$. It turns out that

$$Q_l^k = \begin{cases} \frac{l}{l+k} & \text{if } l \leq n - k \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

This is a result of the fact that the l^{th} comparison branch of the inner-loop is taken only if $a[1]$ is not larger than all of $a[0..1 - 1]$. For this to be the case, the $(l + 1)^{th}$ branch in the previous outer-loop iteration must have been taken, since otherwise the previous iteration of the outer-loop left the maximum of $a[0..1 - 1]$ in $a[1]$. The probability that the $(l + 1)^{th}$ branch of the previous outer-loop iteration is taken is Q_{l+1}^{k-1} . Given that the $(l + 1)^{th}$ branch of the previous outer-loop iteration was taken, the probability that $a[j]$ is not larger than all of $a[0..1 - 1]$ is $1 - 1/(l + 1)$, thus,

$$Q_l^k = Q_{l+1}^{k-1} \left(1 - \frac{1}{l + 1}\right)$$

The bases $Q_l^1 = 1 - 1/(l + 1)$ for $1 \leq l < n$ and $Q_n^1 = 0$ give the solution shown in Equation 1. It is easy to see that in the first outer-loop iteration ($k = 1$) the comparison branch is very likely to be taken, whereas in the last outer-loop iteration ($k = n - 1$), the comparison branch is very unlikely to be taken. Moving between these two extremes causes the intermediate branches to be unpredictable. The remainder of this section examines this phenomenon in more detail.

In order to analyse the behaviour of bubble sort when a bimodal predictor is used, it is useful to make use of the *steady state predictability* function. This function generally gives a good estimate of the probability that a particular branch will be correctly predicted, given that we know the probability that the branch will be taken. For a bimodal predictor this function is given by

$$CP(p) = \frac{3p^2 - 3p + 1}{2p^2 - 2p + 1} \quad (2)$$

Where p is the probability that the branch in question will be taken. Although the derivation of this function is simple, we relegate it to the appendix because we wish to focus mainly on our experimental results. CP can be used to examine the predictability of the branches in bubble sort as the outer-loop iterates. Bubble sort executes $n - k$ comparison branches on its k^{th} outer-loop iteration. Thus, using Equation 1 the average probability that the comparison branch is taken on the k^{th} outer-loop iteration is

$$\begin{aligned} Q_{avg}^k &= \frac{1}{n - k} \sum_{l=1}^{n-k} \frac{l}{l + k} \\ &= 1 + \frac{k}{n - k} (H_k - H_n) \\ &\approx 1 + \frac{k}{n - k} \ln \frac{k}{n} \end{aligned}$$

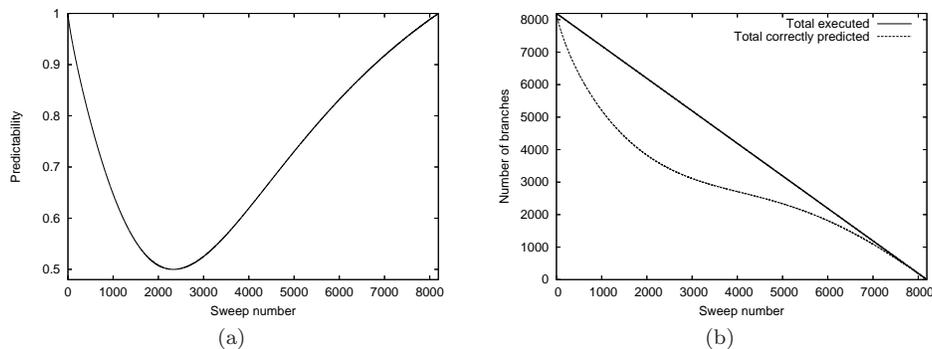


Fig. 3. (a) Shows $CP(Q_{avg}^k)$ as the sweep number, k , increases (i.e. as the outer-loop iterates). This gives a good approximation to the predictability of the comparison branch of bubble sort. (b) Shows the number of correctly predicted branches in bubble sort, as well as the number of executed branches. Initially the number of correctly predicted branches is close to the number of executed branches, however, the number of correctly predicted branches rapidly decreases as the sweep number increases. Gradually, the branches become predictable again. These trends correspond to the variation in predictability shown in (a). The results of (b) were obtained using our own software simulated bimodal predictor averaged over a large number of arrays with 8192 elements. Note also that in (b) the version of bubble sort used does not terminate early if no swaps are performed for an entire execution of the inner-loop.

Where H_n denotes the n^{th} harmonic number. An approximation to the predictability of the comparison branch of bubble sort on the k^{th} iteration is then given by $CP(Q_{avg}^k)$. Figure 3(a) plots this, showing how the predictability of the comparison branch varies with the number of outer-loop iterations. Figure 3(b) shows the measured number of correctly predicted branches as the outer-loop iterates, these results correspond approximately with what was obtained analytically in Figure 3(a).

Initially the branches are quite predictable, however their predictability reduces quite rapidly as the outer-loop iterates. The incremental movement of large keys to the right in bubble sort causes this degradation in the predictability of its branches. As the data becomes close to being fully sorted the predictability of the branches improves again. The partial sorting bubble sort performs allows it to potentially converge to fully sorted data in fewer outer-loop iterations than selection sort. We can detect that the data is sorted early if the branch shown in the inner-loop above is never taken for a whole iteration. However, this partial sorting also greatly increases the number of branch mispredictions bubble sort incurs, substantially slowing it over selection sort or insertion sort.

4.4 Remarks

Our results clearly demonstrate the importance of branch prediction for the elementary sorting algorithms. Our shaker sort implementation has fewer cache misses than our selection sort implementation, as Figure 4(a) shows. It also has a lower instruction count than selection sort, as Figure 2(a) shows. However it is slower due to branch mispredictions, as Figure 4(b) shows. Figure 1 shows the contrast between the branch prediction performance of selection sort and bubble or shaker sort.

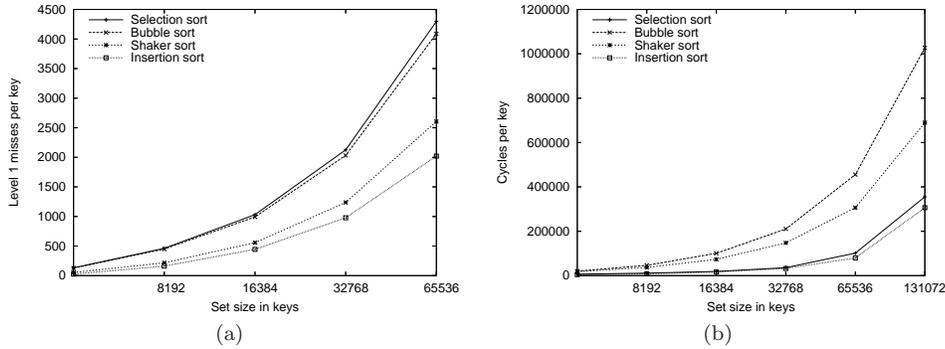


Fig. 4. (a) Shows the level 1 cache misses of the elementary sorting algorithms. These results were obtained by using `sim-cache` to simulate an 8KB direct mapped level 1 cache with 32 byte cache lines. Note that there are no level 2 cache misses since the simulated 2nd level cache is 2MB in size. (b) Shows the number of cycles per key for the elementary sorting algorithms. Despite bubble and shaker sort having better cache performance than selection sort, the vast number of branch mispredictions (see Figure 1(b)) they cause make them less efficient than selection sort in practice. These results were measured using hardware performance counters on a Pentium 4.

Where selection sort causes on average fewer than 12 mispredictions per key for a set size of 65536 keys, bubble and shaker sort cause many thousands. Insertion sort behaves even better than selection sort, causing almost a single misprediction per key. Since insertion sort also has the fewest cache misses it has much to recommend it as an elementary sorting algorithm. It is bubble sort’s partial sorting behaviour that wreaks havoc with branch prediction, adding further weight to Knuth’s [1998] point of view that it has very little to recommend it, besides a catchy name and some interesting theoretical problems it poses.

Finally our results show that over these elementary sorting algorithms two-level adaptive branch predictors are no better than simpler bimodal predictors. This is noteworthy, because two-level predictors are similar to, or significantly more accurate than bimodal predictors for almost all other types of branches [Uht et al. 1997]. In fact, a bimodal predictor out-performs a two-level adaptive predictor for shaker sort, as is shown in Figure 1(b).

5. SHELLSORT

Shellsort [Shell 1959] was the first in-place sorting algorithm with time complexity better than $O(n^2)$. Algorithms like selection and bubble sort use each comparison to resolve at most one inversion (an inversion is a pair of keys in the input which appear in the wrong order) in the input, and it is well known that this restricts their average case running time to be in $O(n^2)$ [Knuth 1998]. Shellsort uses each comparison to resolve more than one inversion, giving better performance.

Shellsort uses a sequence of decreasing integers h_1, h_2, \dots, h_k with $h_k = 1$, called increments. On the i^{th} of its k iterations, shellsort performs an insertion sort but treats $a[j]$ and $a[j + h_i]$ as though they were adjacent (rather than a normal insertion sort, which treats $a[j]$ and $a[j + 1]$ as adjacent).

The choice of increments influences shellsort’s asymptotic behaviour. We used Gonnet’s increments for our implementation, because experiments conducted by

Gonnet and Baeza-Yates [1991] have shown they perform better than other increment sequences in practise. Using Gonnet’s increments $h_i = \lfloor \frac{5}{11} h_{i-1} \rfloor$, with $h_1 = \lfloor \frac{5}{11} n \rfloor$ and the final increment being 1 as usual. With these increments shellsort operates in $O(n^{3/2})$ time.

We use a straightforward implementation of shellsort, except we perform the final iteration (i.e. the iteration in which we perform a standard insertion sort because the increment is 1) separately. For this final insertion sort we extract $\min(a[0], \dots, a[9])$ as a sentinel, removing the need for a bounds check in its inner-loop. This also simplifies the bounds checks on the outer-most loop of the other iterations.

5.1 Branch Prediction Results

The behaviour of the comparison branches in shellsort is simple to analyse. There are approximately $\lfloor \log_{\frac{11}{5}} n \rfloor$ passes, with each pass performing an insertion sort (using the current increment). As we saw in Section 4.2 insertion sort causes about one branch misprediction per key, thus we expect about $n \lfloor \log_{\frac{11}{5}} n \rfloor$ mispredictions in total, and about $\lfloor \log_{\frac{11}{5}} n \rfloor$ mispredictions per key.

Figure 5(a) shows the branch mispredictions per key on average for our shellsort implementations, with bimodal and two-level adaptive predictors. The scale on the x-axis is logarithmic, and the plots are nearly straight lines², as we would expect if the branch mispredictions per key are given by $\lfloor \log_{\frac{11}{5}} n \rfloor$. For example, the final measurement on the x-axis is for a set size of 4194304 and here we observe from the plot approximately 18 misses per key on average, and $\lfloor \log_{\frac{11}{5}} 4194304 \rfloor = 19$.

The comparison branch of the insertion sort inner-loop of shellsort are very unpredictable, unlike the nearly perfectly predictable branches of a standard insertion sort. We measured a predictability of about 50% on average for this branch using a simulated bimodal predictor. A static predictor which predicts all branches as taken would be ideal for predicting the branches of the insertion sorts performed by shellsort. We found that on average, using Gonnet’s increments, a key moves 0.9744 places per iteration (note that a move is of h positions, where h is the current increment). Thus, although the insertion sort branch will still cause just a single misprediction per key, its outcome is not nearly as predictable as a standard insertion sort, where keys on average are moved a larger number of times per inner-loop iteration.

One might expect shellsort’s cache performance to be poor, since it repeatedly accesses distantly separated keys in its input while the increments are large. Each such access could potentially cause a cache miss. However, the fact that on average keys move just 0.9744 places per iteration implies that large numbers of references in distant parts of the input (which could lead to cache misses) are not generally necessary. In fact, our shellsort implementation has better cache performance in both the level 1 and level 2 cache than for our basic heapsort implementation as Figure 5(b), and Figure 5(c) shows. Indeed, our shellsort implementation performs significantly better than our basic heapsort implementation as Figure 5(d) shows

²The flattening of the graph appearing between input sizes 32768 and 65536 is a result of the fact that these input sizes result in the same number of passes (11), and thus approximately the same number of branch mispredictions.

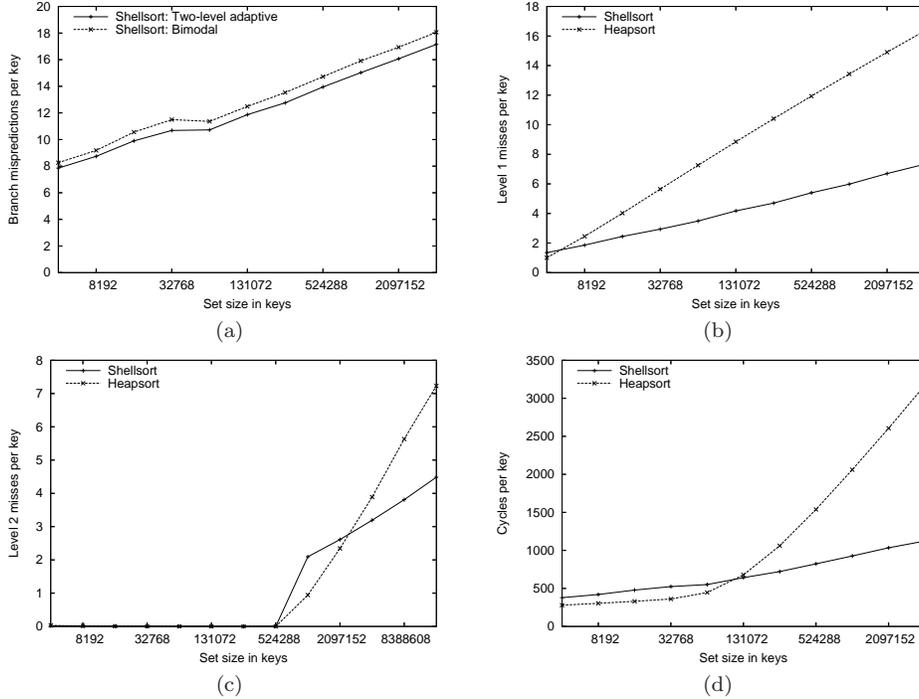


Fig. 5. (a) Shows the average branches misses per key for shellsort using a bimodal and two-level adaptive predictor, both having 4096 table entries. On this occasion, the two-level adaptive predictor out-performs the simpler bimodal predictor. `sim-bpred` was used to simulate these branch predictors. (b) and (c) show that our basic heapsort implementation has inferior cache performance at both levels of the cache compared to our shellsort implementation. Note that (c) includes two extra data set sizes of 2^{23} and 2^{24} keys, in order that the trend of level 2 cache misses is clearer. Shellsort incurs only a small number of cache misses because on average a key moves just 0.9744 places per iteration. These results use `sim-cache` to simulate an 8KB direct mapped level 1 cache and a 2MB direct mapped level 2 cache both with 32 byte cache lines. Finally (d) shows that our shellsort implementation out-performs our heapsort implementation, these results were measured using Pentium 4 hardware performance counters.

(see Section 6 for further details on our heapsort implementations).

6. HEAPSORT

Another well known general purpose sorting algorithm is heapsort [Williams 1964]. Heapsort’s running time is $O(n \log n)$. Heapsort begins by constructing a *heap*: a binary tree in which every level except possibly the deepest is entirely filled, with the deepest level filled from the left. In addition, a heap must also satisfy the *heap property*: A key is associated with each node that is always at least as large as the keys associated with all its children nodes. The sequence of keys resulting from a breadth first traversal of a heap’s nodes can be used to represent it unambiguously in an array.

Using the approach of Floyd [1964] an unordered array can be transformed into a heap in $O(n)$ time [Knuth 1998]. Given a node whose children are heaps but

who may itself violate the heap property, an iteration of Floyd’s approach swaps the node with the larger of its children, and then repeats the same procedure with the subtree rooted at that child, until the heap property is satisfied or a leaf is reached. We refer to this as “sifting down” a node. To build the entire heap, we first sift-down the deepest nodes having children, followed by their parents, and so on until we reach the root.

Once the heap is built, we gradually destroy it in order to sort. We iterate the code below $n - 1$ times.

```
tmp = a[0];
a[0] = a[--n];
a[n] = tmp;
sift_down(a, 0, n);
```

On the i^{th} iteration we swap the largest key, $a[0]$, with $a[n - i]$. After each swap, the heap property is restored by sifting down the new root of the heap (an operation with worst case time $O(\log n)$). After iteration i the heap contains $n - i$ keys, and is found in $a[0..n - i - 1]$. Meanwhile $a[n - i..n - 1]$ contains the i largest keys of the input from smallest to largest.

6.1 Heapsort Variations

For our base heapsort implementation, we used the approach described in the previous section, but removed a bounds check from the sift down procedure by introducing a sentinel at the cost of no extra instructions. The following optimization reduced the instruction count by about 10% on average compared to an implementation of sift-down which uses a bounds check. We have not seen it described elsewhere. In a heap, at most one node has a single left child, all other nodes have either two children or zero children. On every sift-down, we compare each node which has children with each of its children. As a result of the node with a single child, on every sift-down we must check that the node being compared to its children has a right child as well as a left child. To avoid this check while building the heap, we can just insert a maximal extra node at the end of the heap. When destroying the heap however, every second iteration requires the insertion of this extra node. To avoid this the code for destroying the heap shown in the previous section is changed to

```
tmp = a[0];
a[0] = a[--n];
sift_down(a, 0, n);
a[n] = tmp;
```

By not modifying $a[n - 1]$ until after sifting down, it acts as a sentinel for sift-down. This removes the need for the bounds check without introducing any extra instructions.

Aside from this optimization, we also applied caching optimizations of LaMarca and Ladner [1996] to give several cache-optimized versions of the algorithm. Firstly, the heap can be padded so that siblings always reside in the same cache line. Another cache optimization is realised by abandoning Floyd’s heap building method and simply repeatedly growing the heap one array element at a time, sifting each

new key upwards as needed. This greatly improves the temporal locality of the algorithm when the data does not fit entirely within the cache.

To increase the utilization of cache lines (and hence require fewer cache lines to be fetched) we can use a k -ary heap instead of the binary heap that has been assumed until now. When using a k -ary heap, every node having children (except perhaps one) has k children. When sifting up or down a k -ary heap, the maximum (or minimum, depending on the heap property being used) child must be found, which requires $k - 1$ comparisons. Of course, as k grows the sort degenerates to a selection sort. Note also that for a k -ary heap, the optimization described above, which avoids bounds checking while sifting down cannot be applied. However, it is still more efficient to add a sentinel on every second iteration of the heap destruction (since this introduces $O(n)$ operations whereas the bounds check in the inner-loop introduces $O(n \log n)$ operations). For modest values of k a performance improvement can be expected over a simple binary heap [LaMarca and Ladner 1996]. We experimented with 4-heaps and 8-heaps, combined with the aforementioned cache optimizations. We did not use heaps with a fan-out greater than 8 because we aim to fit the heap nodes within a 32-byte cache-line, and therefore increasing the fan-out beyond 8 is likely to worsen cache performance. As a simple example, consider a heap of fan-out 16, where only 8 keys fit in a cache-line. A traversal of the heap from root to leaf causes approximately $2 \log_{16} n$ cache misses, which is clearly inferior to a fan-out of 8, where we have only approximately $C = \log_8 n$ misses in the traversal, since $2 \log_{16} n = \frac{3}{2}C$. A more detailed discussion of cache-aware heaps can be found in the work of LaMarca and Ladner [1996].

6.2 Branch Prediction Results

Figure 6(a) shows the average number of branches per key for our heapsort implementations. We note that the cache-optimized heapsort implementations execute fewer branches per key than our base heapsort implementation. Although these algorithms execute fewer branches per key on average, they cause a larger total number of branch mispredictions than our base heapsort implementation. Indeed, Figure 6(b) shows that the base heapsort implementation incurs the fewest branch mispredictions of any of the algorithms. This is despite the fact that it is executing more branches on average. Figure 6(b) also shows that the simpler bimodal predictors are generally no worse at predicting branches than the two-level adaptive predictors. With the exception of shellsort, this is in keeping with the trend we observed for the other sorting algorithms examined so far.

Notwithstanding the base heapsort implementation having the lowest misprediction rate, Figure 6(c) shows that the most efficient implementation on our hardware configuration to be the 8-heap variation of heapsort, followed by the 4-heap variation. This suggests the improved cache performance of these algorithms outweighs their increased level of branch mispredictions shown in Figure 6(b).

When using a k -ary heap, finding the maximum (or minimum) child at a particular node causes the i^{th} comparison branch to be taken with probability $1/(1+i)$. The comparisons behave just like those in selection sort (see Section 4.1), and are therefore quite predictable. For our 8-heap we unrolled the loop to find the minimum child into 7 separate branches. Figure 6(d) shows the behaviour of these 7 branches, for an unrolled loop. As expected, the later branches are gradually

taken less often on average, with a corresponding increase in their predictability. Unrolling this loop not only removes the loop over-head, but is also experimentally observed to improve the average prediction accuracy by about 2% [Biggar and Gregg 2005]. The average steady state predictability in the unrolled case, is given by

$$\frac{1}{7} \sum_{i=1}^7 CP \left(\frac{1}{1+i} \right) \approx 0.724$$

Where CP is the steady state predictability function introduced in Section 4.3. Without unrolling, the comparison branch is taken on average with probability $\frac{1}{7} \sum_{i=1}^7 \frac{1}{i+1}$, and its predictability can be estimated as

$$CP \left(\frac{1}{7} \sum_{i=1}^7 \frac{1}{1+i} \right) \approx 0.706$$

As is observed experimentally, the unrolled loop results in about a 2% greater prediction accuracy.

7. MERGESORT

Mergesort [Knuth 1998] is a another $O(n \log n)$ sorting algorithm, which works by repeatedly merging pairs of sorted lists into a single sorted list of twice the length. These merges can be done very simply in linear time and space. Mergesort's input can be thought of as n sorted lists of length one. To simplify the description, we assume the number of keys, n , is a power of two. Mergesort makes $\lg n$ passes over the input where the i^{th} pass merges sorted lists of length 2^{i-1} into sorted lists of length 2^i .

7.1 Mergesort Variations

For our base mergesort implementation we apply several optimizations. Firstly, our mergesort alternates the merge operation between two arrays to avoid unnecessary copying. Secondly, a bitonic sort as described by Sedgewick [2002], removes instructions from the inner-loop. A bitonic sort sorts every second list in reverse, merging then works from the two ends of the array containing the elements to be merged, terminating when the indices cross. Since the indices cannot go outside the bounds of the array unless they cross first, this eliminates the need for a bounds check in the inner-loop. An insertion sort is also initially used to create sorted lists of length four, rather than allowing mergesort to sort these lists.

We also analysed the branch prediction behaviour in several cache conscious mergesort variations using the work of LaMarca and Ladner [1997] as a reference, the details of our implementations differ slightly however [Biggar and Gregg 2005]. Mergesort is not in-place³ and uses an auxiliary array for its merge operation. With a cache of size C , the largest input we can sort in the cache therefore has size $C/2$. To avoid conflict misses, the arrays should be allocated so that the cache line index

³Worst case linear time in-place merging is possible but is generally inefficient in practice, see for example Katajainen *et al* [1996].

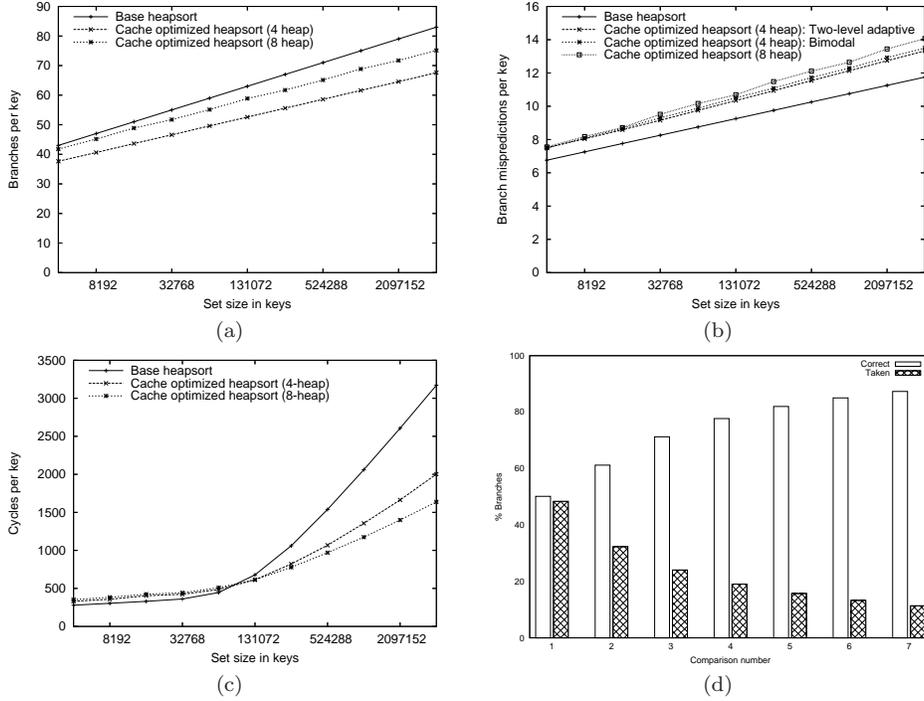


Fig. 6. Branch prediction and performance characteristics of our heap sort implementations. (a) Shows the average number of branches per key measured using `sim-bpred`. (b) Shows the average branch mispredictions per key for tables of 4096 predictors, again measured using `sim-bpred`. The two-level adaptive (which used a 10-bit history register) and bimodal predictors have close to identical performance, and to avoid cluttering the diagram only one instance of a two-level adaptive predictor is shown, the remainder of the predictors are bimodal. (c) Shows the cycle counts of the algorithms measured using Pentium 4 hardware performance counters. Note that the Pentium 4 has 32 byte cache lines in its first level cache, and 64 byte cache lines in its second level cache. Finally (d) shows the predictability of the unrolled branches associated with an 8-heap, measured with our own software simulated bimodal predictor.

of the start of the source array is $C/2$ from the cache line index of the start of the destination array. This can be done for both the level 1 and level 2 cache.

Since each key is only used once per pass in mergesort, if the input is larger than the cache then we will have no temporal re-use of data at all. To address this we can *tile* mergesort: cache sized blocks of the input are mergesorted, and then merged as usual. Combined with alignment this variation is referred to as tiled mergesort. Tiling mergesort in this way does not fully address the problem of its poor temporal re-use, since the merging of the cache sized blocks in tiled mergesort will exhibit no temporal re-use and again have very poor cache performance. To address this, a multi-way merge of the cache sized blocks can be used. When sorting n keys, a multi-way merge combines the $\lceil \frac{n}{C} \rceil$ sorted buffers into a single sorted buffer while reading through each of them only once. A heap can be used to help implement the multi-way merge efficiently. We used a cache friendly heap with a fan-out of 8, because this provided the best performance in our heap sort

implementation, as described in Section 6.1. This multi-mergesort variation has better cache performance. We refer to this combination of alignment and tiling with multi-way merging simply as multi-mergesort.

In addition to these mergesort variations, more elaborate mergesort based algorithms exist which are optimal with respect to cache usage. For example, the multi-mergesort described by Aggarwal and Vitter [1988] is optimal in the external-memory model, while several mergesort variations have been described which are optimal in the cache-oblivious model [Frigo et al. 1999; Brodal et al. 2007]. We have chosen the mergesort algorithms above because they are relatively simple and include multi-way merging, which is generally used by the more elaborate mergesort variations.

7.2 Branch Prediction Results

Figure 7(a) shows the average number of branches per key executed by our mergesort implementations. Up to 2^{18} keys can be sorted in the simulated 2 MB level 2 cache (each key is 4 bytes, and mergesort is not in-place). As a result, when the number of keys being sorted reaches 2^{19} there is a dramatic increase in the number of branches executed by multi-mergesort. This is because a multi-way merge begins to be used. Figure 7(b) shows that there is a corresponding increase in the number of branch mispredictions caused by multi-mergesort as a result of the multi-way merge.

Many of the additional comparison branches introduced by the multi-way merge are associated with an 8-heap, and are therefore quite predictable, as we saw in Section 6.2. In addition, two-level adaptive predictors outperform bimodal predictors on multi-mergesort’s branches. Figure 7(b) shows this. Both predictors have 4096 entry tables. Varying the table size of the bimodal predictor between 2048 and 8192 entries did not result in any reduction in the number of branch mispredictions. On the other hand, increasing the table size of the two-level adaptive predictor gave incremental improvements in branch predictability. This suggests that the two-level adaptive predictor is able to exploit correlations between branch outcomes which occur during the multi-way merge.

Figure 7(c) shows the simulated level 2 cache misses per key for our mergesort implementations. It is noteworthy that although multi-mergesort has the best cache performance, its cache performance is only slightly better than tiled mergesort. However, as the set size increases the difference in cache performance will slowly become more pronounced, as the trend in Figure 7(c) suggests. Figure 7(d) shows the cycles per key measured for our mergesort implementations running on actual hardware. These results show that our less cache conscious tiled mergesort implementation out-performs multi-mergesort. This is likely because the set sizes are not large enough to make the superior cache performance of multi-mergesort the dominating factor in the comparative performances. Instead, for the data sizes we used of 2^{12} to 2^{22} keys, multi-mergesort’s increased instruction count and branch mispredictions cause its performance to be inferior to tiled mergesort.

7.3 Branch Mispredictions in Multi-way Merges

Brodal and Moruz [2005] have shown that any deterministic comparison based sorting algorithm performing $O(dn \log n)$ comparisons must incur $\Omega(n \log_d n)$ branch

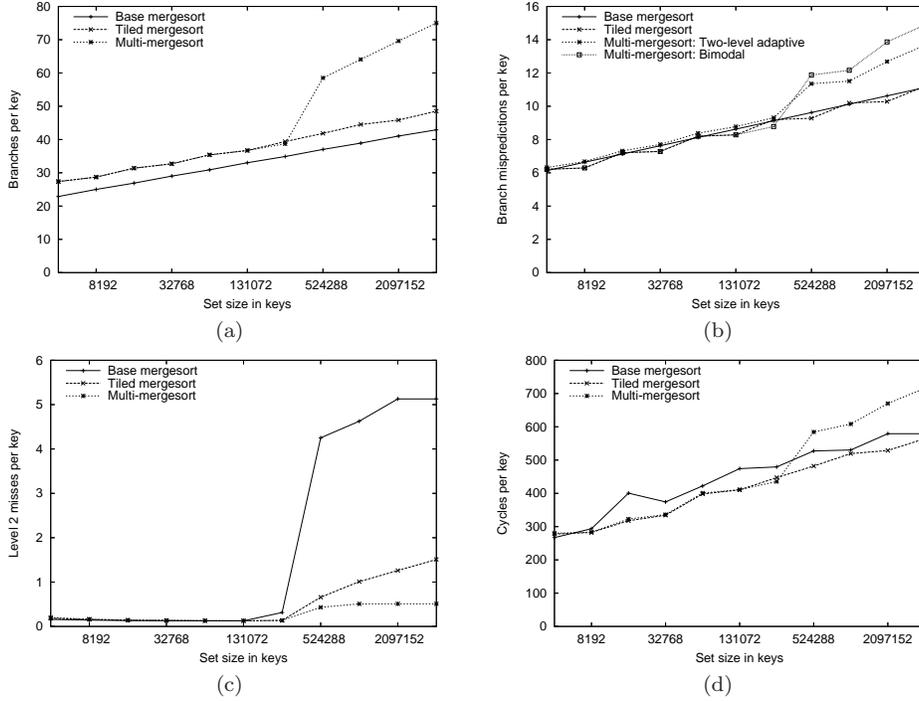


Fig. 7. (a) Shows the branches per key executed by our mergesort variations. The sudden increase in the number of executed branches for multi-mergesort results from the use of the multi-way merge when the data no longer fits within the cache. (b) Shows the number of branch mispredictions per key, all predictors had 4096 table entries. Bimodal predictors were used for base and tiled mergesort. These results were generated using `sim-bpred`, the two-level adaptive predictor had a 10-bit history register. (c) Shows the level 2 cache misses per key, base mergesort has by far the worst cache behaviour, while multi-mergesort has slightly better cache performance than tiled mergesort. These results were generated by using `sim-cache` to simulate a direct mapped 2MB cache with 32-byte cache lines. Finally (d) shows the cycles per key of our algorithms measured using Pentium 4 hardware performance counters. Despite multi-mergesort’s superior cache performance, its heightened instruction count and branch mispredictions mean that the less cache conscious tiled mergesort performs the best out of the algorithms.

mispredictions⁴. They have also described *insertion d-way mergesort*, a multi-way mergesort algorithm which is optimal in the sense that it incurs $O(n \log_d n)$ branch mispredictions when performing $O(dn \log n)$ comparisons. Brodal and Moruz do not provide performance results for their algorithm. We now describe their algorithm and will subsequently present performance results for cache-friendly implementations of their algorithm.

Insertion *d*-way mergesort is an out-of-place multi-mergesort algorithm operating in $O(dn \log n)$ time on an input of n keys. For the simplicity of the description assume n is a power of d . The algorithm performs $\log_d n$ passes over its input, with the i^{th} pass performing *d*-way merges of sorted lists of length d^{i-1} into sorted

⁴This is under the mild assumption that each key comparison determines the direction of an immediately following branch instruction.

lists of length d^i . These d -way merges operate as follows: d sorted subarrays of an array a of m keys are merged in $O(dm)$ time. The algorithm maintains a vector $i = (i_1, \dots, i_d)$ of indices into the sorted subarrays, together with a permutation $\pi = (\pi_1, \dots, \pi_d)$ of $\{1, \dots, d\}$ such that $(a[i_{\pi_1}], \dots, a[i_{\pi_d}])$ is sorted. To perform a step of the merge, $a[i_{\pi_1}]$ is appended to the output array and i_{π_1} is incremented. The inner-loop of insertion sort (see Section 4.2) is then used to update π to ensure in $O(d)$ time that $(a[i_{\pi_1}], \dots, a[i_{\pi_d}])$ is again in sorted order. This process is repeated m times, when all the keys in the subarrays will have been exhausted.

Our implementation of this algorithm is similar to our mergesort implementations described in the previous section. An insertion sort is used to initially create sorted subarrays of length 4, rather than allowing mergesort to sort these lists. In addition, we double-align the source and destination arrays to avoid conflict misses. We also implemented a variation of the double-aligned, tiled multi-mergesort described in the previous section. Instead of using an 8-heap to perform the multi-way merge we use the insertion merge just described. We refer to this latter algorithm as cache-optimized insertion multi-mergesort.

Figure 8(a) shows the number of instructions per key for our insertion mergesort variations. Cache-optimized Insertion multi-mergesort has a substantially lower instruction count than the insertion d -way mergesort algorithms. Increasing d from 3 to 6 causes a significant reduction in instruction count, however increasing d beyond 6 does not give further reductions in instruction count, as can be seen in Figure 8(a). Note that the value of d in cache-optimized insertion multi-mergesort is determined by the number of cache sized tiles that need to be merged.

Figure 8(b) shows that as d is increased in the insertion d -way mergesort there is a corresponding reduction in the number of branch mispredictions. For cache-optimized insertion multi-mergesort, the number of branch mispredictions differs noticeably depending on whether a bimodal or two-level adaptive predictor is used. The difference in the number of mispredictions is likely a result of the use of bitonic merging described in the previous section. When performing the insertion multi-merge, a branch is used to detect whether every second subarray is reversed. The two-level adaptive predictor is able to exploit the branch's history and correctly predict its outcome, giving a reduction in the number of branch mispredictions. On the other hand, the bimodal predictor is likely to oscillate between predicting taken and not-taken and mispredict the branch on nearly every iteration of the insertion multi-merge.

Despite the superior cache performance of the insertion d -way mergesort algorithms as shown in Figure 8(c), the best performing algorithm in practise is cache-optimized insertion multi-mergesort as can be seen in Figure 8(d). Cache-optimized insertion multi-mergesort also incurs more branch mispredictions than these algorithms, as Figure 8(b) shows. However its much lower instruction count appears to compensate for its inferior performance in these areas. In fact, cache-optimized insertion multi-mergesort is the best performing of all our mergesort implementations, out-performing our tiled mergesort implementation (see Figure 7(d)).

Our results indicate that Brodal and Moruz's [2005] insertion merge is a practical technique. One desirable property of the technique is that increasing d improves locality while also reducing branch mispredictions. Moreover, it may be possible to

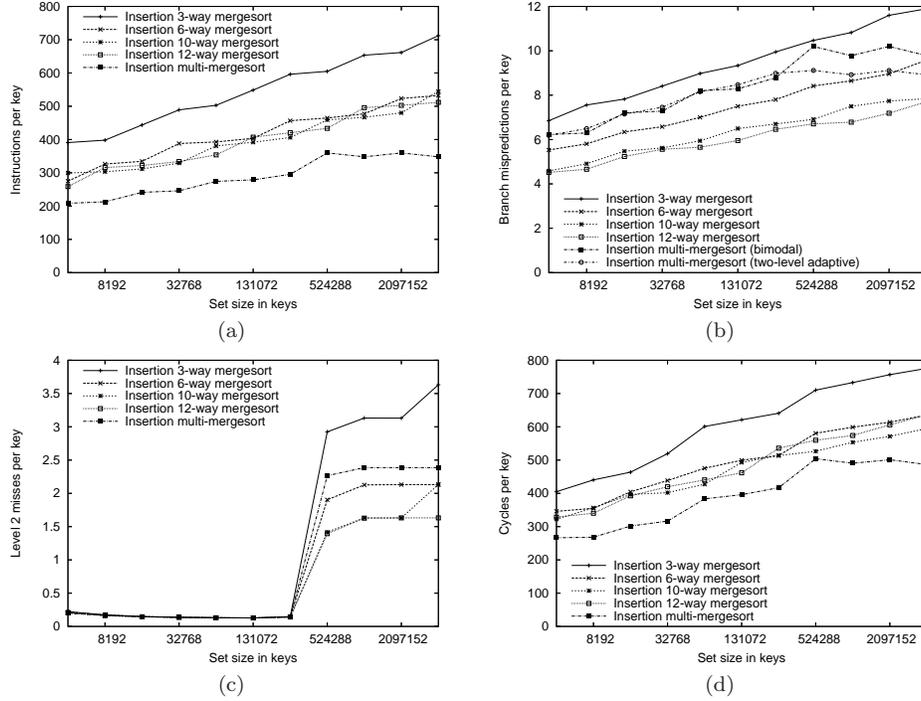


Fig. 8. (a) Shows the instruction counts for the insertion d -way mergesort algorithms, for a variety of values of d . It also shows the much lower instruction count of our cache-optimized insertion multi-mergesort variation compared to these algorithms. (b) Shows the branch mispredictions per key for the algorithms, all results show bimodal predictor results, except for cache-optimized insertion multi-mergesort, for which we also show results when using a two-level adaptive predictor with a 10-bit history register and 4096 table entries, since for this algorithm the two-level adaptive predictor is significantly better than the bimodal predictor. (c) Shows the level 2 cache misses of the algorithms when operating on a 2 MB direct mapped cache with 32-byte cache lines. These results were gathered using `sim-cache` and `sim-bpred`. Finally (d) shows the cycles per key of the algorithms, measured using Pentium 4 hardware performance counters. Despite cache-optimized insertion multi-mergesort's heightened cache misses and branch mispredictions, its low instruction count enables it to out-perform the insertion d -way mergesort algorithms.

substantially mitigate the high instruction count of the technique by varying the value of d depending on the number of keys which remain to be sorted. In addition, for small values of d the insertion merge should be special-cased. It is also likely that the cache performance of the algorithm could be substantially improved by copying blocks of keys (for example, as many keys as fit in a cache-line) to small buffers when appending keys from subarrays to the destination buffer. We leave a fuller investigation into determining the best trade-offs between reducing the instruction count of the algorithm, improving its locality and maintaining a modest number of branch mispredictions to future work.

8. QUICKSORT

Quicksort [Hoare 1962] is an algorithm which selects a key from its input called the pivot and then partitions the other keys into two sets, one set containing keys at most equal to the pivot and another containing keys at least equal to the pivot. The keys of these sets should appear respectively to the left and right of pivot in sorted order. Therefore they can be sorted independently, each using quicksort.

8.1 Quicksort Variations

For our basic quicksort implementation, we make use of a number of improvements suggested by Sedgewick [1978]. Firstly, we use an explicit stack instead of recursing, to save space. Secondly, after partitioning, we always quicksort the set with the smaller number of elements first. This reduces the worst case stack space to $O(\log n)$ from $O(n)$. Thirdly, inputs of less than a certain `THRESHOLD` (typically `THRESHOLD` is 10) in size are not quicksorted at all. Instead, insertion sort is run as a post-pass over the output of quicksort. The smallest of the first `THRESHOLD` keys is used as a sentinel for insertion sort. Finally, we use median-of-3 pivoting, greatly reducing the chances of very unbalanced partitioning and also removing the need for bounds checks in the inner-loop, thus greatly reducing the instruction count.

Improvements such as those just described are found in most efficient quicksort implementations. We also investigated the branch prediction behaviour of cache conscious quicksort implementations. LaMarca and Ladner [1997] suggest that instead of running insertion sort as a post-pass over the quicksorted input, each time quicksort is presented with an input of size less than `THRESHOLD` it immediately insertion sorts it. Although LaMarca and Ladner’s experiments show that this slightly increases the instruction count due to the extra instructions associated with setting up the many insertion sort passes, it is more than compensated for by the resultant drop in cache misses. We refer to this as memory-tuned quicksort.

When quicksort’s input fits within a particular level of the cache quicksort has good cache properties at that level. To combat a dramatic increase in cache misses when the input does not fit in the cache, LaMarca and Ladner also suggest multi-quicksort. Multi-quicksort chooses a set of distinct random pivots and performs a multi-way partition which, on average, divides the input into cache sized containers. Each container is then quicksorted. We leave the precise details of performing a multi-way partition to elsewhere [LaMarca 1996; Biggar and Gregg 2005], it is sufficient to note that we can decide which container a key should be placed in by using either a sequential or binary search.

8.2 Branch Prediction Results

Figure 9 shows the branch prediction results of our quicksort implementations. In all figures, the most important branches are the *i* and *j* branches (see Figure 10), since these are in quicksort’s inner-loop, and are executed more often than any of the other branches (with the exception of the `sequential` branch of Figure 9(d)).

The branch prediction behaviour of basic quicksort and memory-tuned quicksort are similar. As can be seen in Figure 9(b) quicksort has a slightly lower branch misprediction rate on the `insertion` branch. This is because a basic quicksort implementation, which performs insertion sort as a post-pass, will generate branch

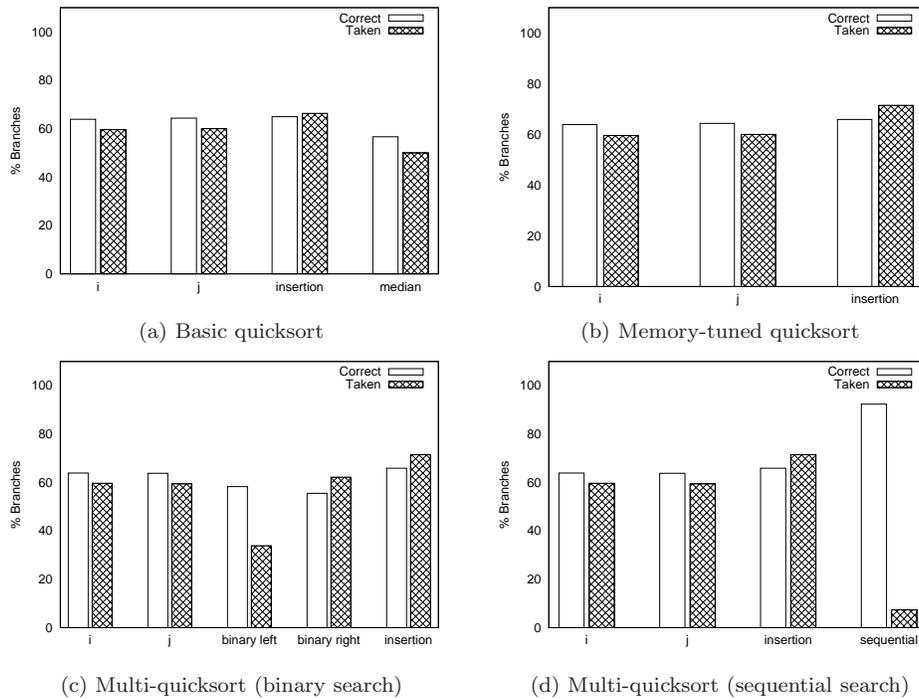


Fig. 9. Overview of branch prediction behaviour in our quicksort implementations. Every figure shows the behaviour of the i and j branches when using a median-of-3 pivot. As described in Section 8.2, these branches are about 60% biased and 64% predictable when using the median-of-3. In (a) the **median** branch is the combined results of the branches which compute the median-of-3 (these branches are also executed for (b), (c) and (d)). Comparing (a) with (b), (c) and (d), we see that the **insertion** branch associated with its insertion sort is slightly less predictable than in the other variations. This is due to it running as a post-pass. Finally, comparing (c) with (d) we see that the binary search branches of (c), **binary left** and **binary right**, are very unpredictable compared to the **sequential** branch of (d).

```

pv = a[l];
i = l, j = r + 1;
while(true)
{
    while(a[++i] < pv) ; // i-loop
    while(a[--j] > pv) ; // j-loop
    if(i >= j) break;
    swap(a[i], a[j]);
}
swap(a[l], a[j]);

```

Fig. 10. Quicksort's partition inner-loop. We refer to the inner while loops as the i and j loops. We refer to their associated branches as the i and j branches respectively.

mispredictions on keys chosen as pivots since they never require movement. In practice, the over-all reduction in branch misprediction rate from 35% in basic quicksort to 34% in memory-tuned quicksort is probably offset by the branch misprediction rate of the outer-loop exit in the many insertion sorts executed by memory-tuned quicksort. However, memory-tuned quicksort has better caching properties, and so from the point of view of both branch prediction and caching behaviour this optimization is to be recommended. Note also that unlike LaMarca and Ladner [1997] we do not observe an increase in instruction count from base quicksort to memory-tuned quicksort, as Figure 11(b) shows.

Multi-quicksort results in a 19% reduction in the number of iterations of the i and j loops of quicksort's partition inner-loop, though their misprediction rate remains the same at 37%. Figure 9(c) shows branch behaviour for multi-quicksort when using binary search. The binary search branches are highly unpredictable, being incorrectly predicted 43% of the time. If multi-quicksort uses sequential search, the number of executed branches dramatically increases, but these branches are correctly predicted 92% of the time, as shown in Figure 9(d).

The implementations of multi-quicksort have the best cache performance of the quicksort variations, as Figure 11(c) shows. However, the multi-quicksort implementations also introduce many extra instructions per key as is seen in Figure 11(b). Naturally, the sequentially searched version introduces more extra instructions than the binary searched version. However, as described in the previous paragraph, the branches introduced by binary search are highly unpredictable. As a result, multi-quicksort using binary search has more branch mispredictions than the sequentially searched version (see Figure 11(a)), consequently, despite having identical cache performance, binary searched multi-quicksort performs worse than the sequentially searched multi-quicksort, as shown in Figure 11(d). Of course, on very large data sets (larger than the 2^{22} keys we experimented with), the reduced instruction count of binary searched multi-quicksort would eventually allow it to out-perform sequentially searched multi-quicksort.

As Figure 11(d) shows, our base quicksort implementation performs the best of all. It seems the improved cache performance of multi-quicksort is not enough for it to out-perform base quicksort on the data set sizes we experimented with. Examining the trends of Figure 11(c), it seems likely that multi-quicksort would eventually out-perform base quicksort due to its improved cache performance. The fact that base quicksort out-performs memory-tuned quicksort is very difficult to explain. In the simulations memory-tuned quicksort has the same cache performance and instruction count, as shown in Figures 11(c) and 11(b). Also in the simulations, memory-tuned quicksort has slightly fewer branch mispredictions than base quicksort, as Figure 11(a) shows. Despite this, base quicksort slightly out-performs memory-tuned quicksort on our non-simulated hardware test as Figure 11(d) shows.

8.3 Pivot Choice

The choice of pivot element has a strong influence on the behaviour of the i and j branches (see Figure 10). Clearly, if the pivot is chosen as the median of the input then either branch is equally likely to be taken (i.e. the loop executes) as not taken (i.e. the loop exits). In this case, the branches are also on average approximately

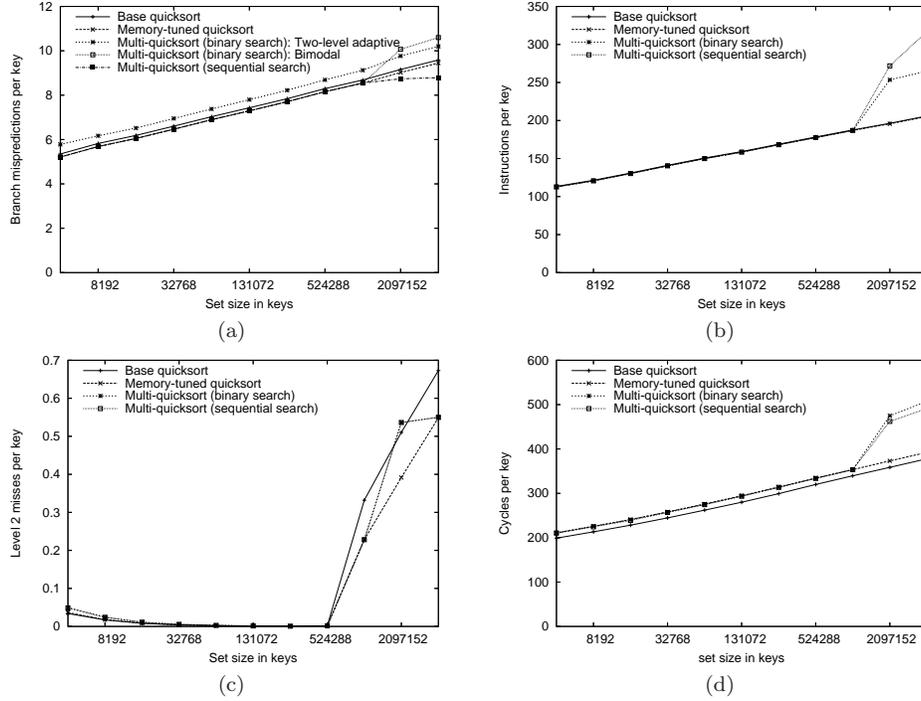


Fig. 11. (a) Shows the simulated branch mispredictions per key for our quicksort implementations. All predictors use 4096 table entries and are bimodal predictors except where indicated. Multi-quick sort using binary search is the only implementation which gives varying results from the use of a two-level adaptive predictor. The multi-quick sort implementations show an increase in branch mispredictions when the data no longer fits within the cache due to the invocation of a multi-way partition. (b) Shows the instruction counts of our quicksort implementations. When the data sets no longer fit within the cache the multi-quick sort implementations show a large increase in instruction count due to the invocation of a multi-way partition, these results were also generated using SimpleScalar. (c) Shows the 2nd level cache misses for our quicksort implementations. As is expected, the multi-quick sort implementations have the best cache performance. These results were generated using `sim-cache` to simulate a 2MB direct-mapped cache with 32-byte cache lines. These results were generated using `sim-bpred`. (d) Shows the cycles per key for our quicksort implementations measured using Pentium 4 hardware performance counters. Again, the invocation of a multi-way partition when the data no longer fits within the cache causes a noticeable increase in the cycles per key of the multi-quick sort implementations.

50% predictable. However, on average over random input, with a fixed choice of pivot the i and j branches are 66% biased (that is, they are taken 66% of the time). It is straightforward to show why this is so. Assume that the data to be partitioned is a random permutation of $\{1, \dots, n\}$. If the chosen pivot has rank q then the i branch will be taken with probability $(q-1)/(n-1)$. Moreover, the branch will be executed (but not necessarily taken) a total of q times, since there are exactly $q-1$ elements to the left of the pivot. The average bias is given by averaging over all possible choices of pivot q , thus

Median-of	Reduction In #Branches	Bias	Predictability	Misprediction Increase
1	0%	66.1%	70.6%	0%
3	14%	60.0%	64.3%	6.1%
5	16.5%	57.8%	61.6%	9.2%
7	17.8%	56.3%	59.9%	11.8%
9	19.6%	55.1%	58.3%	14.3%

Table I. The effect of choosing a pivot closer to the median on the prediction of the i and j branches (the results are the same for both). As a higher order median is used the bias of the i and j branches is reduced, with a corresponding reduction in their predictability. The reductions in the number of executed branches are relative to a fixed choice of pivot (i.e. median-of-1). The misprediction increases correspond to the total number of branches mispredicted with a higher order median compared to a fixed choice of pivot. These measurements were obtained from our own software simulations of a bimodal branch predictor. Note that the small number of branches required to compute the median approximations are included in these results.

$$B_{avg}^n = \frac{2}{n(n+1)} \sum_{q=1}^n \left(\frac{q-1}{n-1} \right) q = \frac{2}{3} \quad (3)$$

So the i -loop is on average 66% biased. With this bias, the i and j branches are about 71% predictable. The fact that predictability of these branches exceeds their biases may at first seem paradoxical. The reason the branches are more predictable than biased is because the bias is an average. The average predictability of the i and j branches can be obtained very simply from Equation 3 using the steady state predictability function, CP , introduced in Section 4.3, thus

$$P_{avg}^n = \frac{2}{n(n+1)} \sum_{q=1}^n CP \left(\frac{q-1}{n-1} \right) q$$

As $n \rightarrow \infty$ we have $P_{avg} = \int_0^1 CP(q) dq = 3/2 - \pi/4$. Thus we see P_{avg} is about 0.71. This estimated average predictability is close to what is observed experimentally when using a fixed pivot over random data, as the predictability entry of the first row of Table I shows.

If the pivot is chosen as the median-of-3, then the i and j branches are about 60% biased in the taken direction, and about 64% predictable. It is especially noteworthy that although median-of-3 gives around a 14% reduction in the total number of executed comparison branches, there is actually a 6% *increase* in the total number of branch mispredictions. As the pivot more closely approximates the median of the input, there is a continuing trend of a reduction in the total number of executed branches, as well as reducing bias and predictability with an increasing total number of mispredictions, as Table I shows.

The use of the median-of-3 or higher order median approximation greatly reduces the chances of the worst case behaviour of quicksort, and on average therefore results in fewer comparison branches being executed. From an information theoretic perspective, each comparison branch resolves more uncertainty, or yields more information, and is hence harder to predict. This observation applies also to the predictability of the branches used in the binary searched version of multi-quicksort versus the sequentially searched version. There is a trade-off to be had between

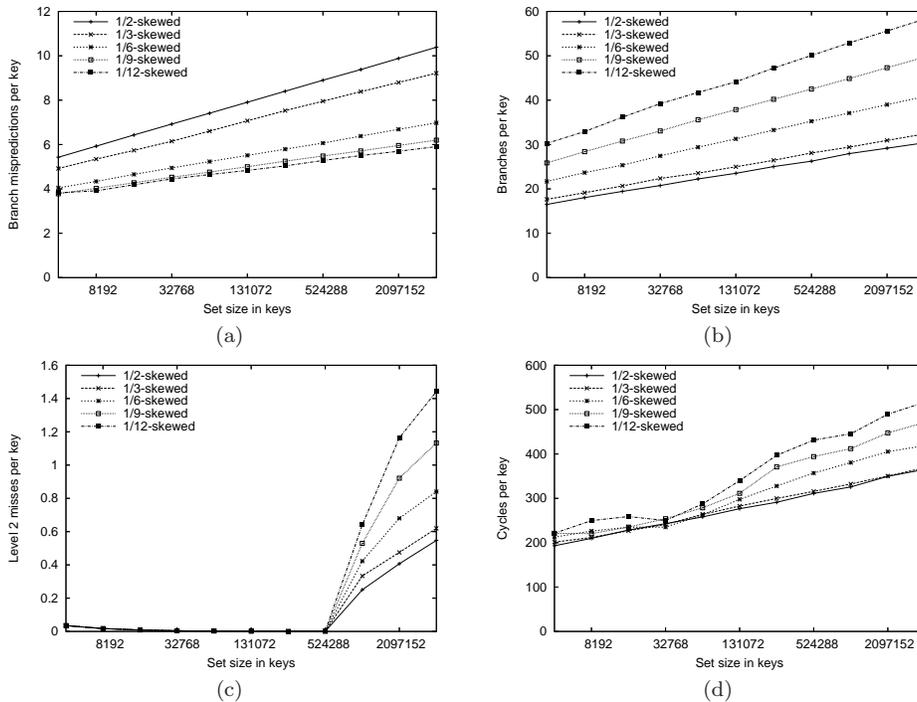


Fig. 12. This figure shows performance results for idealized skewed quicksort. Note that idealized skewed quicksort operates on random permutations of $\{1, \dots, n\}$ in order that it can directly compute a skewed pivot (see Section 8.4). (a) Shows that selecting a skewed pivot causes a reduction in branch mispredictions, and (b) shows the resultant increase in the number of branches executed as a result of skewing. These results were gathered using `sim-bpred` to simulate a bimodal predictor with 4096 table entries. (c) Shows how the level 2 cache misses are increased as the skew of the pivot is increased. These results were gathered using `sim-cache` to simulate a 2 MB direct-mapped cache with 32-byte cache lines. Finally (d) shows that skewing the pivot results in *worse* performance over-all than using an idealized (1/2)-skewed pivot. The results of (d) were gathered using Pentium 4 hardware performance counters.

the penalty induced by unpredictable comparisons, versus the increased instruction count caused by making comparisons more predictable.

8.4 Skewed Pivots

In work concurrent to and independent of our own, Kaligosi and Sanders [2006] have investigated the effect of the choice of pivot on the performance of quicksort. They found that an artificially skewed pivot gives better performance than using a random pivot or the median-of-3, due to the reduction in branch mispredictions. This performance increase is despite the increased instruction count such an artificially skewed pivot causes. Kaligosi and Sanders examine α -skewed pivoting, that is, where the pivot has rank $\lfloor \alpha n \rfloor$ when sorting n keys. They examine the performance of quicksort when the pivot is determined randomly, from the median-of-3, or is the true median (i.e. a (1/2)-skewed pivot). They show that using a (1/10)-skewed pivot gives a performance increase over all the aforementioned pivot choices.

In their experiments, Kaligosi and Sanders sort random permutations of $\{1, \dots, n\}$, and they take advantage of this to compute an α -skewed pivot very simply (e.g. as $l + \alpha(r - l)$, where the end-point indices are l and r). We refer to our implementations of quicksort which compute a skewed pivot in this way as idealized skewed quicksort. Figure 12(a) shows the reduction in branch mispredictions in quicksort when skewed pivots are used. Moreover, Figure 12(b) shows that as the pivot is skewed, more comparison branches are executed.

Choosing a skewed pivot adversely affects the spatial locality of quicksort, and Figure 12(c) shows the increase in cache misses as the skew of the pivot is increased. Finally, we note that using a skewed pivot does not result in an improvement in performance on our Pentium 4 architecture. As Figure 12(d) shows, an idealized $(1/2)$ -skewed pivot performs better than all other pivot choices. The disparity between the results we observe and those reported by Kaligosi and Sanders [2006], is likely due to difference in pipeline lengths and cache sizes.⁵

A reduction in the number of cache misses caused by skewing the pivot can be achieved by only skewing the pivot when the number of keys remaining to be sorted falls below some threshold size (for example, the size of the level 2 cache). We refer to our variations of quicksort which switch to skewing the pivot at some threshold size as switch-skewed quicksort.

Kaligosi and Sanders note that in a practical setting (i.e. when the input is not a random permutation of $\{1, \dots, n\}$) a skewed pivot must be chosen with random sampling. We now briefly describe how this may be accomplished, in order that we may compare our skewed quicksort implementations fairly with our base quicksort implementation. Let $R(n, s)$ be the average rank of a pivot selected as the 2nd smallest element of a sample of s keys taken from $\{1, \dots, n\}$. It is straightforward to show that

$$\begin{aligned} R(n, s) &= \sum_{k=2}^{n-s+2} \frac{\binom{n-k}{s-2}}{\binom{n}{s}} (k-1)k \\ &= 2 \sum_{k=2}^{n-s+2} \frac{\binom{n-k}{s-2} \binom{k}{2}}{\binom{n}{s}} \\ &= \frac{2(n+1)}{s+1} \end{aligned}$$

The performance of the partition step of quicksort (see Figure 10) is greatly improved by the use of sentinel elements, therefore it is necessary to choose the 2nd smallest element in order that a sentinel element is guaranteed to be available when

⁵Kaligosi and Sanders [2006] report a speed-up on a Pentium 4 Northwood (but do not observe speed-ups on other machines, like an AMD Opteron) with a 31-stage pipeline, while we performed our experiments on a Pentium 4 Willamette, which has only a 20-stage pipeline and smaller caches than the Northwood.

the partition is performed. Thus, for any reasonably large n in order to choose an on average $(1/q)$ -skewed pivot we must select the 2nd smallest element of a sample of $s = 2q - 1$ keys as the pivot. Note that $q = 2$ corresponds to ordinary median-of-3 pivoting. We refer to the variations of quicksort which use random sampling to select a skewed pivot as sample-skewed quicksort.

We experimented with two further variations of quicksort with skewed pivots: Sample-skewed quicksort, which uses random sampling to select a skewed pivot as well as sample-switch-skewed quicksort. In order to improve cache performance, sample-switch-skewed quicksort uses a median-of-3 pivot until the number of keys remaining falls below a threshold, at which point a $(1/q)$ -skewed pivot is chosen via random sampling.

In Figure 13 we present results for the skewed quicksort variations with their parameters chosen as the ones which gave best performance. We tested $(1/q)$ -skewed pivots for $q \in \{2, \dots, 12\}$. For sample-switch-skewed quicksort, we tested thresholds of $2^p C$ keys, for $p \in \{-4, \dots, 1\}$, for each of the skews. Here C is the level 2 cache size. We found a threshold of 131,072 keys and a skew of $1/3$ gave the best performance.

Figure 13(a) shows that the sample-switch-skewed quicksort has slightly fewer level 2 cache misses than sample-skewed quicksort, due to only using a skewed pivot when the number of keys remaining to be sorted is 131,072 or less. Both algorithms have slightly worse cache performance than base quicksort. The overhead of using random sampling to select a skewed pivot, combined with the extra comparisons resulting from the use of a skewed pivot cause the instruction count of the sampling quicksorts to be higher than the instruction count of base quicksort, as Figure 13(b) shows.

Figure 13(c) shows the branch mispredictions caused by each algorithm. The $(1/3)$ -sample skewed algorithms have very slightly fewer branch mispredictions than base quicksort. Note that base quicksort is in fact just a $(1/2)$ -sample skewed quicksort. We also note that the sample-skewed quicksort has slightly fewer branch mispredictions than sample-switch-skewed quicksort. This is a result of the fact that the sample-switch-skewed quicksort only switches to using a skewed pivot when the number of keys remaining to be sorted is at most 131,072.

Finally, Figure 13(d) shows that, as with idealized skewing discussed above, sample based skewing does not result in a performance improvement. The sample-skewed quicksorts are slightly slower than a simple base quicksort implementation.

Our results show that deliberately skewing the pivot in quicksort is a somewhat fragile technique for achieving a speed-up. As noted above, while Kaligosi and Sanders [2006] reported an idealized $(1/10)$ -skewed quicksort to give better performance than an idealized $(1/2)$ -skewed quicksort, our performance results are different due to architectural variations. As a result of this, where possible it is desirable to avoid comparisons altogether in order to remove the penalty associated with branch mispredictions. When sorting a lexicographically ordered set (like integers or strings), radix sorting is therefore an attractive technique since it avoids comparisons between keys altogether. We examine a simple least significant digit radix sort in the following section.

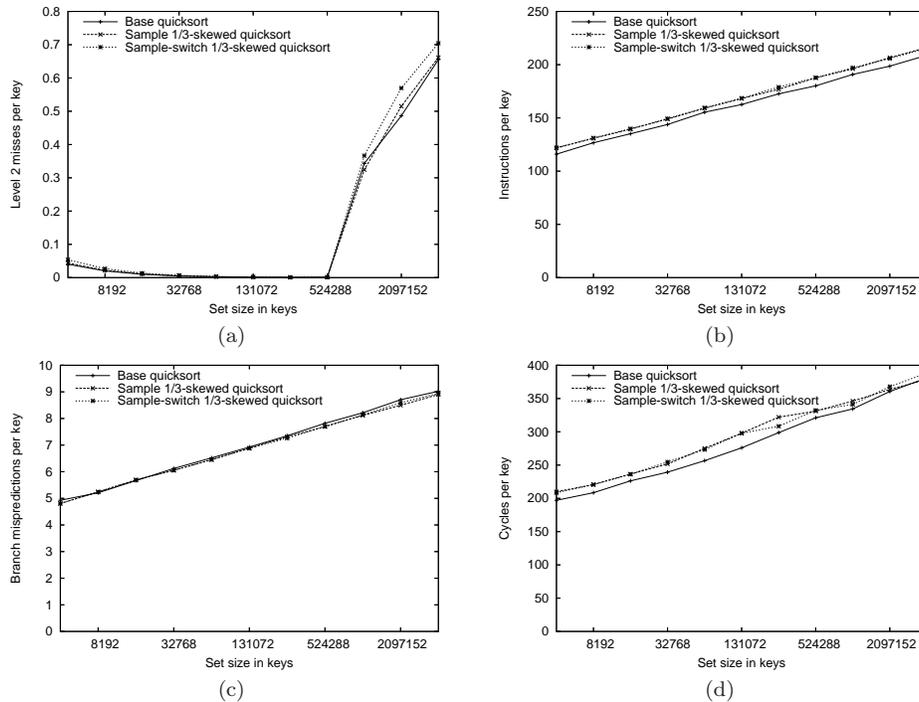


Fig. 13. This figure shows performance results for (1/3)-skewed quicksorts using random sampling to select their pivots, compared to base quicksort. (a) Shows the skewed quicksorts have worse level 2 cache performance than base quicksort, with sample-switch-skewed quicksort having slightly better cache performance than sample-skewed quicksort. These results were gathered using `sim-cache` to simulate a 2MB direct mapped cache with 32 byte cache lines. (b) Shows that the skewed pivots and overhead of random sampling increases the instruction count of the skewed quicksort algorithms. In (c) we see that the skewing does slightly reduce the number of branch mispredictions incurred by the skewed quicksorts, compared to base quicksort. These results were gathered with `sim-bpred`. Finally (d) shows that we do not observe an improvement in performance when using a skewed pivot, compared to a simple median-of-3 quicksort. These results were gathered using Pentium 4 hardware performance counters.

9. RADIX SORT

The purpose of this paper is to show the relevance of branch mispredictions to sorting. We now show how to develop a simple least significant digit radix sort [Friend 1956] implementation for 32-bit integers that, in all our tests, is more efficient than all other sorting algorithms presented in this paper. This is in part due to the tiny number of branch mispredictions incurred by radix sorting. It is also in part because of the fact that radix sort operates in $O(\lceil w/r \rceil (n + 2^r))$ time when sorting n w -bit keys with a radix of r . When sorting integers w and r are comparable in magnitude and relatively small (e.g. on a 32-bit machine we have $w = 32$ and we might choose $r = 8$), and as a result radix sort implementations usually have lower instruction counts than $O(n \log n)$ time comparison based sorting algorithms.

The cache performance of radix sort is generally poor compared to algorithms like mergesort or quicksort. In radix sort, each key is used just once each time

it is in the cache, when the data to be sorted is larger than the cache every key access will result in a cache miss, due to the random accesses to the destination array in a simple least significant digit radix sort. On the other hand, mergesort and quicksort have good temporal locality and use keys repeatedly before they are ejected from the cache.

9.1 Memory-tuned Radix Sort

Our basic radix sort implementation is a typical least significant digit radix sort. We operate in radix 256, and begin by counting the number of keys with a particular least significant byte by indexing a table of 256 entries with the least significant bytes of the keys. The entries of this table are then summed to store the cumulative count of keys with a smaller least significant byte. Now each key is placed in an auxiliary array according to the value retrieved by using its least significant byte to index into the table of cumulative counts. Before the next iteration, the auxiliary array is written back to the original array and the sort can proceed on the next most significant byte. We use this implementation to compare against the simple caching improvements we now suggest.

We can firstly remove the copying back and forth between the original and auxiliary array, by just alternating their roles on each step. Since we are sorting 32 bit integers, the number of iterations is even and the data will end up sorted in its original location. If the number of steps was odd, an additional copy would be required at the end. In addition, we use an improvement suggested by LaMarca and Ladner [1997], where the counting of the next most significant bytes is begun during their preceding iteration. We refer to this optimized version of radix sort as memory-tuned radix sort. More elaborate cache optimizations are possible for least significant digit radix sorts, for example those described by Rahman and Raman [2001]. Indeed, even choosing an 8-bit radix is a somewhat ad-hoc decision, Rahman and Raman have suggested using an 11-bit radix when sorting 32-bit keys. However, these simple cache optimizations were the only ones we applied because they were enough that our radix sort implementation was faster than any of the comparison based algorithms.

In our basic radix sort implementation, the first array is accessed 3 times and the second array twice, in each of the four iterations over the four bytes in the 32-bit integers. This gives 20 accesses in total. In memory-tuned radix sort each key of the two arrays is accessed only nine times in total. At the start, it traverses the original array once to count the least significant bytes. It then iterates four times, positioning the keys based on the previously established counts. These four iterations cause a total of eight accesses, because each array is accessed once per iteration.

9.2 Results

Our memory-tuned radix sort implementation requires fewer processor cycles per key than even our base quicksort implementation as Figure 14(a) shows. Since radix sort is out-of-place, we might consider aligning its two arrays as we did with mergesort (see Section 7.1). However, radix sort does not access its arrays in the same regular fashion that mergesort does and so is unlikely to benefit from such a change. Indeed, as Figure 14(a) shows, the double-alignment does not result in any

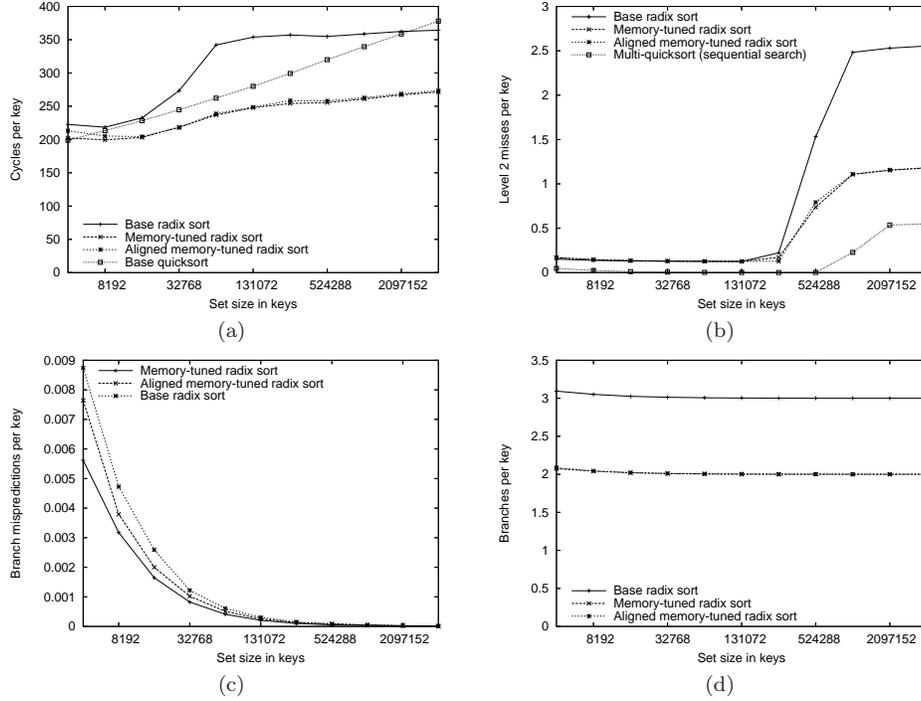


Fig. 14. (a) Shows the cycles per key of our radix sort implementations alongside our base quicksort implementation, measured using Pentium 4 hardware performance counters. (b) Shows the level 2 cache misses incurred by our radix sort implementations and a multi-quicksort implementation for a 2MB direct mapped cache with 32-byte cache lines. These results were generated using `sim-cache`. (c) Shows the average branch mispredictions per key incurred by our radix sort implementations while using bimodal predictors with 4096 table entries, the results for two-level adaptive predictors are extremely similar. Finally, (d) shows the average number of branches per key executed by our implementations. The results of (c) and (d) were generated using `sim-bpred`.

performance improvement.

Figure 14(b) shows the level 2 cache misses for our basic, memory-tuned, and double-aligned implementations. The number of level 2 cache misses is only slightly higher than we measured for quicksort while the array fits inside the cache, presumably because radix sort is out-of-place. We note that when the array no longer fits inside the cache the memory-tuned implementation has approximately half the number of misses per key (1.25) than the basic version (2.5). When the data no longer fits inside the cache there should be about one miss for each key per pass per cache line. Each cache line can fit 8 keys, and thus we expect the observed 2.5 misses per key since there are 20 passes in total. Similarly, in memory-tuned radix sort we expect slightly below 1.25 misses per key since there are 9 passes in total. Finally, we note that the double-aligned radix sort has very slightly worse cache performance than the simpler memory-tuned radix sort.

A large advantage of radix sort is its almost complete lack of branch mispredictions however. Figure 14(c) shows the tiny number of branch mispredictions incurred by radix sort. The tiny increase in the number of branch mispredictions

per key of the double-aligned radix sort compared to the memory-tuned implementation is a result of an extra comparison branch used in the alignment process. The radix sort implementations also execute a similarly tiny number of branch instructions per key as Figure 14(d) shows.

10. COMPARISON

We now provide a brief comparison of the best variants of each of the algorithms discussed in the preceding sections. Figure 15 shows the cycles per key of shellsort (see Section 5), cache optimized heapsort (see Section 6), insertion multi-mergesort (see Section 7.3), base quicksort (see Section 8) and memory-tuned radix sort (see Section 9). We exclude the $O(n^2)$ algorithms from the comparison because they are enormously slower than these algorithms except for tiny input sizes.

We note that our memory-tuned least significant digit (LSD) radix sort algorithm is the best performing of all the algorithms. This result highlights the fact that although LSD radix sort algorithms generally have worse cache performance than efficient divide and conquer algorithms like quicksort, their low instruction counts coupled with their almost complete lack of branch mispredictions makes them more efficient over-all. As a result, when sorting data chosen from a lexicographically ordered set, it is wise to choose a radix sorting algorithm.

Our base quicksort algorithm using a median-of-3 pivot is the next best performing algorithm. As described in Section 8.4, we observed that on our Pentium 4 hardware quicksort’s performance is adversely affected by a skewed pivot, due to increased cache misses and a higher instruction count. This is in contrast to the speed-up achieved by Kaligosi and Sanders [2006] when using a skewed pivot for quicksort, although they do note that they did not observe a speed-up on all the architectures they experimented with.

Our insertion multi-mergesort algorithm using the insertion merge of Brodal and Moruz [2005] is less efficient than both base quicksort and memory-tuned radix sort. However, as we noted at the end of Section 7.3, it seems plausible the cache performance and instruction count of insertion multi-mergesort algorithms could be improved while retaining the algorithms’ attractive ability to control the number of branch mispredictions they incur.

The remaining algorithms, heapsort and shellsort, are substantially less efficient than the algorithms just described. However, we note once more that shellsort comfortably out-performs our best heapsort implementation.

11. RELATED WORK

We present the first large-scale systematic study of branch prediction and sorting. However, researchers have been aware for many years that the branches in comparison-based sorts cause problems on pipelined architectures. As early as 1972 Knuth commented on the “ever-increasing number of ‘pipeline’ or ‘number crunching’ computers that have appeared in recent years” whose “efficiency deteriorates noticeably in the presence of conditional branch instructions unless the branch almost always goes the same way”. He also notes that “radix sorting is usually more efficient than any other known method for internal sorting on such machines” (the same comment appears in [Knuth 1998]). Most likely, Knuth was

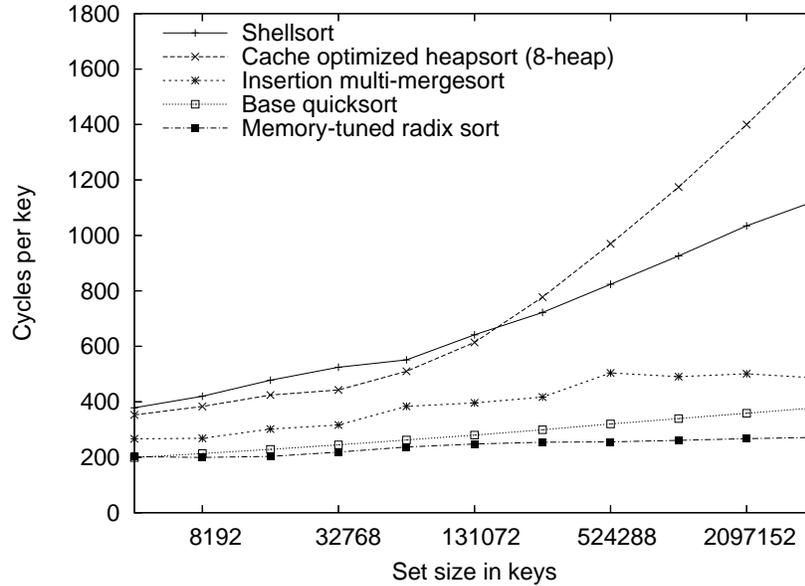


Fig. 15. Shows the cycles per key of the best performing variants of the all the algorithm described in the previous sections (excluding the $O(n^2)$ algorithms). These results were measured using Pentium 4 hardware performance counters. Section 10 provides a fuller discussion of these results.

referring to the IBM 7030 “Stretch” computer (released in 1961 with a four-stage pipeline), Seymour Cray’s Freon-cooled CDC 6600 supercomputer (released 1964), or other early supercomputers which used pipelining in the execution of general-purpose instructions.

A number of other authors have considered optimized implementations of sorting algorithms for more recent single-processor machines. Nyberg *et al* [1994] present a quicksort-based algorithm for RISC architectures. They found that the main limit on performance was cache behaviour, and did not mention branch prediction even once. Bentley and McIlroy [1993] implemented a highly-tuned quicksort, to be used in the `qsort` function of the standard C library. Again, they did not consider branch prediction, perhaps because the first desktop processors with dynamic branch predictors (such as the DEC Alpha 21064, MIPS R8000 and Intel Pentium) were only just appearing around that time.

Agarwal [1996] developed an optimized algorithm for sorting random records on an IBM RS/6000 RISC machine. His is the first published algorithm that we are aware of that deliberately attempts to improve performance by eliminating branch mispredictions. First, it uses the seven higher order bits of the key to divide the data into 128 buckets. Each of these buckets is radix-sorted. However, before the radix sort is reaches the lowest order bits, the probability of any adjacent pair of keys being out of order is extremely low, because the problem states that the data is guaranteed to be random. In the final stage, the algorithm checks that large runs of keys are correctly ordered using subtraction and bitwise operations, and only sorts a section if some key is found to be out of order. Although Agarwal’s techniques

are interesting, they are very dependent on the randomness of the data; it is easy to construct common cases that would result in very poor performance. Nonetheless, the work is interesting because it shows the effectiveness of radix sorting techniques on pipelined architectures.

A recent trend in optimizing algorithms for FFT, linear algebra, and digital signal processing has been to automatically generate and evaluate very large numbers of variants of the algorithm. A similar approach is used by Li *et al* [2005] to automatically generate sorting algorithms that are tuned to the architecture of the target machine and the characteristics of the input data. They identify six sorting primitives, which can be combined to construct a sorting algorithm. A large number of combinations are tested, using a genetic algorithms search to find the most efficient. The resulting implementations are, in many cases, significantly faster than commercial sorting libraries. Although they do not consider branch prediction explicitly, they find that the implementations that perform best usually use radix sort for a large part of the sorting process.

Sanders and Winkel [2004] investigate the use of *predicated* instructions on Intel Itanium 2 processors. In addition to its normal inputs, a predicated instruction takes a predicate register input. If the value in the predicate register is true, the instruction takes effect; otherwise it does not. Predication is normally implemented by the instruction being executed regardless of whether the predicate is true. However, its result is only *written back* to a register or memory if the predicate is true. Although predicated instructions use execution resources regardless of whether they are allowed to write back, they allow conditional branches to be eliminated. Sanders and Winkel show how the partition step in quicksort can be rewritten to use predicated instructions on an Itanium 2, a highly instruction-level parallel machine. Itanium 2 provides large numbers of parallel functional units and registers which make such trade-offs worthwhile.

Mudge *et al* [1996] examine the behaviour of the i and j comparison branches in randomized quicksort, as an example to demonstrate a general method for estimating the limit on the predictability of branches in a program. They argue that the optimal predictor for these branches would keep a running count of the total number of array elements examined so far that are greater than the pivot. If the majority of the elements examined so far are greater than the pivot, the next element is predicted as greater than the pivot, and vice versa. They estimate that this approach will give a prediction accuracy of 75%, which is close to our result for these branches, using a bimodal predictor. This shows that a bimodal predictor is close to optimal for randomized quicksort's branches, since P_{avg} of Section 8.2 is about 0.71.

The simple technique described by Mudge *et al* could be made use of on architectures featuring semi-static branch predictors. In such a situation the hint-bit of the appropriate i and j branches in quicksort could be dynamically adjusted according to their simple scheme just described. A modern architecture where this may be practical is IBM's CELL architecture. On this architecture there is an 18-cycle branch misprediction penalty, and the branch predictor is semi-static [Eichenberger *et al.* 2006]. In fact, in a situation such as this, where it is possible to hint branches optimally, a semi-static predictor should out-perform dynamic predictors.

Brodal *et al* [2005] examine the adaptiveness of quicksort. They find that although randomized quicksort is not adaptive (i.e. its asymptotic analysis does not improve with respect to some measure of presortedness), its performance is better on data with a low number of inversions. They justify this by showing that the expected number of element swaps performed by randomized quicksort falls as the number of inversions does. The number of branch mispredictions is roughly twice the number of element swaps, because two branch mispredictions tend to occur when the two while loops of the partition code exit (see Figure 10). Furthermore they note that the branch mispredictions are the dominant part of the running time of randomized quicksort. This provides an empirical relation between the presortedness of the input, randomized quicksort’s performance, and the number of branch mispredictions.

In closely related work Brodal and Moruz [2005] provide a lower bound on the number of branch mispredictions a comparison-based sorting algorithm must incur based on the number of comparisons it performs. They show that any deterministic sorting algorithm performing $O(dn \log n)$ comparisons must incur $\Omega(n \log_d n)$ branch mispredictions. This result demonstrates that minimizing the number of comparisons (i.e. $d = 2$) maximizes the number of branch mispredictions. Moreover, an algorithm with $d = n/\log n$, performing $O(n^2)$ comparisons must incur $\Omega(n)$ branch mispredictions. Indeed, insertion sort is an example of an algorithm achieving the optimal trade-off in this case, performing $O(n^2)$ comparisons and incurring $O(n)$ branch mispredictions.

Brodal and Moruz also show that a mergesort variation, insertion d -way mergesort, can achieve this lower bound based on the number of comparisons it performs, incurring $O(n \log_d n)$ branch mispredictions and performing $O(dn \log n)$ comparisons. We have provided experimental results for cache efficient implementations of their algorithm in Section 7.3.

12. CONCLUSION

This paper has presented the first large scale study on the characteristics of branch instructions in all of the most common sorting algorithms. We noted that except where the sorting algorithms involve reasonably complicated flow control (i.e. shell-sort, multi-mergesort, multi-quicksort), bimodal predictors perform just as well as two-level adaptive predictors. Indeed we noted that for shaker sort a bimodal predictor out-performs a two-level adaptive predictor. Since both types of predictor generally result in similar numbers of branch mispredictions, the performance of most sorting algorithms when using a bimodal branch predictor is not likely to differ substantially from their performance when using a two-level adaptive predictor.

Of the elementary sorting algorithms we noted that insertion sort, causing just a single misprediction per key, has the best branch prediction characteristics. On the other hand, bubble and shaker sort operate in a manner which causes surprisingly unpredictable branches.

Sorting (and indeed, searching) algorithms have been designed to minimize the number of comparisons necessary in the worst case. However, minimizing the number of comparisons makes each comparison less predictable, as we noted at the end of the previous section. In addition, it is important to note that a sorting algorithm

which performs more comparisons than another does not necessarily have more predictable branches with respect to a given predictor. Indeed, bubble and shaker sort are examples algorithms which perform $O(n^2)$ comparisons but nevertheless have very unpredictable branches when using bimodal or two-level adaptive predictors.

When a branch is mispredicted, there is often a large penalty on modern processors. There is therefore a tension between reducing the number of executed comparisons and the penalty associated with increasing the chance of branch mispredictions. This tension is especially evident in the choice of pivot for quicksort. We have also noted that the speed-ups resulting from the use of a skewed pivot, as described by Kaligosi and Sanders [2006], can also slow down quicksort depending on the pipeline length and cache sizes of the underlying hardware. Indeed, on our hardware a skewed pivot gave worse performance than a simple median-of-3 approach, due to increased cache misses and a higher instruction count.

Since cache optimizations are now recognized as very important, we have also noted the affect that these cache optimizations have on the branch prediction behaviour of sorting algorithms. In particular we provided performance results for cache optimized versions of quicksort, mergesort, heapsort and radix sort. We have also shown that the insertion d -way merge of Brodal and Moruz [2005] is a practical technique for performing multi-way merges, out-performing implementations using a heap. An interesting piece of future work could examine the branch prediction characteristics of practical cache oblivious sorting algorithms, such as the algorithm described by Brodal *et al* [2007].

Our study provides concrete results of the branch prediction characteristics for a large number of sorting algorithms, accompanied with the relevant analysis. These branch prediction results have been obtained for realistic dynamic predictors, using a variety of configurations of bimodal and two-level adaptive predictors.

A. APPENDIX: STEADY STATE PREDICTABILITY

In Sections 4.3, 6.2 and 8.2 we made use of the steady state predictability function CP . When a branch instruction is executed repeatedly with a known probability p of being taken, $CP(p)$ gives a good estimate of the probability that it will be correctly predicted.

The bimodal predictor associated with a branch instruction can be thought of as a Markov chain. Note that we are thus assuming that successive branches are independent. If the branch has probability p of being taken then the transition matrix of the associated Markov chain is

$$M = \begin{bmatrix} p & p & 0 & 0 \\ 1-p & 0 & p & 0 \\ 0 & 1-p & 0 & p \\ 0 & 0 & 1-p & 1-p \end{bmatrix}$$

The steady state probabilities are given by s such that $Ms = s$. Since the steady state probabilities sum to one this can be reduced to solving a 3×3 linear system. The resultant probabilities are given by

$$\begin{bmatrix} P(ST) \\ P(T) \\ P(NT) \\ P(SNT) \end{bmatrix} = \frac{1}{2p^2 - 2p + 1} \begin{bmatrix} p^3 \\ p^2(1-p) \\ p(1-p)^2 \\ (1-p)^3 \end{bmatrix}$$

Here $P(ST)$, $P(T)$, $P(NT)$ and $P(SNT)$ denote the respective probabilities of the predictor states strongly taken, taken, not-taken and strongly not-taken described in Section 2. A branch is correctly predicted with probability p if we are in the strongly taken or taken state. Similarly, a branch is correctly predicted with probability $1 - p$ if we are in the strongly not-taken or not-taken state. Thus,

$$CP(p) = p[P(ST) + P(T)] + (1 - p)[P(SNT) + P(NT)]$$

This equation simplifies to Equation 2 of Section 4.3.

REFERENCES

- AGARWAL, R. C. 1996. A super scalar sort algorithm for RISC processors. 240–246.
- AGGARWAL, A. AND VITTER, J. S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9, 1116–1127.
- AUSTIN, T., ERNST, D., LARSON, E., WEAVER, C., RAJ DESIKAN, R. N., HUH, J., YODER, B., BURGER, D., AND KECKLER, S. 2001. *SimpleScalar Tutorial (for release 4.0)*.
- BENTLEY, J. L. AND MCILROY, M. D. 1993. Engineering a sort function. *Softw. Pract. Exper.* 23, 11, 1249–1265.
- BIGGAR, P. AND GREGG, D. 2005. Sorting in the presence of branch prediction and caches. Tech. Rep. TCD-CS-05-57, University of Dublin, Trinity College. August.
- BRODAL, G. S., FAGERBERG, R., AND MORUZ, G. 2005. On the adaptiveness of quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*. 130–140.
- BRODAL, G. S., FAGERBERG, R., AND VINTHER, K. 2007. Engineering a cache-oblivious sorting algorithm. *J. Exp. Algorithmics* 12, 2.2.
- BRODAL, G. S. AND MORUZ, G. 2005. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *WADS*. 385–395.
- EICHENBERGER, A. E., O'BRIEN, J. K., O'BRIEN, K. M., WU, P., CHEN, T., ODEN, P. H., PRENER, D. A., SHEPHERD, J. C., SO, B., SUR, Z., WANG, A., ZHANG, T., ZHAO, P., GSCHWIND, M. K., ARCHAMBAULT, R., GAO, Y., AND KOO, R. 2006. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal* 45, 1, 59–84.
- FLOYD, R. W. 1964. Treesort 3: Algorithm 245. *Communications of the ACM* 7, 12, 701.
- FRIEND, E. H. 1956. Sorting on electronic computer systems. *J. ACM* 3, 3, 152.
- FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 285.
- GONNET, G. H. AND BAEZA-YATES, R. 1991. *Handbook of algorithms and data structures: in Pascal and C (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- HAHR, M. 2006. Random.org: True random number service. Web resource, available at <http://www.random.org>.
- HINTON, G., SAGER, D., UPTON, M., CARMEAN, D., KYKER, A., , AND ROUSSEL, P. 2001. The microarchitecture of the pentium 4 processor. *Intel Technology Journal* Q1.
- HOARE, C. A. R. 1962. Quicksort. *Computer Journal* 5, 1, 10–15.
- INTEL. 2001. Desktop performance and optimization for intel pentium 4 processor. Tech. rep.
- INTEL. 2004. Ia-32 intel architecture optimization - reference manual. Tech. rep.
- KALIGOSI, K. AND SANDERS, P. 2006. How branch mispredictions affect quicksort. In *Proceedings of The 14th Annual European Symposium on Algorithms (ESA 2006)*. (to appear).

- KATAJAINEN, J., PASANEN, T., AND TEUHOLA, J. 1996. Practical in-place mergesort. *Nordic J. of Computing* 3, 1, 27–40.
- KNUTH, D. E. 1997. *The Art Of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, pp120–160, pp73–78.
- KNUTH, D. E. 1998. *The Art Of Computer Programming, Volume 3 (2nd ed.): Sorting And Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, p82, p110, p175, pp73–180, pp153–155, pp158–168.
- LAMARCA, A. AND LADNER, R. 1996. The influence of caches on the performance of heaps. *J. Exp. Algorithmics* 1, 4.
- LAMARCA, A. AND LADNER, R. E. 1997. The influence of caches on the performance of sorting. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 370–379.
- LAMARCA, A. G. 1996. Caches and algorithms. Ph.D. thesis, University of Washington.
- LI, X., GARZARAN, M. J., AND PADUA, D. 2005. Optimizing sorting with genetic algorithms. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, Washington, DC, USA, 99–110.
- MUCCI, P. J. 2004. *PapiEx Man Page*.
- MUDGE, T., CHEN, I.-C., AND COFFEY, J. 1996. Limits to branch prediction. Tech. Rep. CSE-TR-282-96. 2.
- NYBERG, C., BARCLAY, T., CVETANOVIC, Z., GRAY, J., AND LOMET, D. 1994. Alphasort: a risc machine sort. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. ACM Press, New York, NY, USA, 233–242.
- PATTERSON, D. A. AND HENNESSY, J. L. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc.
- RAHMAN, N. AND RAMAN, R. 2001. Adapting radix sort to the memory hierarchy. *J. Exp. Algorithmics* 6, 7.
- SANDERS, P. AND WINKEL, S. 2004. Super scalar sample sort. In *Algorithms . ESA 2004: 12th Annual European Symposium*, S. Albers and T. Radzik, Eds. Lecture Notes in Computer Science, vol. 3221. Springer, Bergen, Norway, 784–796.
- SEGEWICK, R. 1978. Implementing quicksort programs. *Commun. ACM* 21, 10, 847–857.
- SEGEWICK, R. 2002. *Algorithms in C*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc.
- SHELL, D. L. 1959. A high-speed sorting procedure. *Commun. ACM* 2, 7, 30–32.
- UHT, A. K., SINDAGI, V., AND SOMANATHAN, S. 1997. Branch effect reduction techniques. *Computer* 30, 5, 71–81.
- WICKREMESINGHE, R., ARGE, L., CHASE, J. S., AND VITTER, J. S. 2002. Efficient sorting using registers and caches. *J. Exp. Algorithmics* 7, 9.
- WILLIAMS, J. W. J. 1964. Heapsort: Algorithm 232. *Communications of the ACM* 7, 6, 347–348.
- XIAO, L., ZHANG, X., AND KUBRICHT, S. A. 2000. Improving memory performance of sorting algorithms. *J. Exp. Algorithmics* 5, 3.