Analysing cache effects in distribution sorting

Naila Rahman
and
Rajeev Raman
King's College London

We study cache effects in *distribution* sorting algorithms for sorting keys drawn independently at random from a uniform distribution ('uniform keys'). We note that the performance of a recently-published distribution sorting algorithm, Flashsort1, which sorts n uniform floating-point keys in $O(n)$ expected time, does not scale well with the input size due to poor cache utilisation. We present an approximate analysis for distribution sorting uniform keys which, as validated by simulation results, predicts the expected cache misses of Flashsort1 quite well. Using this analysis, we design a multiple-pass variant of Flashsort1 which outperforms Flashsort1 and comparison-based algorithms on uniform floating-point keys for moderate to large values of n. Using experimental results we also show that the integer distribution sorting algorithm MSB radix sort performs well on both uniform integer and uniform floating-point keys.

Categories and Subject Descriptors: F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*; E.5 [**Data**]: Files—*Sorting/searching*

General Terms: Efficient sorting algorithms, Memory hierarchy, External-memory algorithms

Additional Key Words and Phrases: Cache

1. INTRODUCTION

Most algorithms are analysed on the random-access machine (RAM) model of computation [Aho et al. 1974], using some variety of unit-cost criterion. In particular, the RAM model assumes that accessing a location in memory costs the same as a built-in arithmetic operation, such as adding two word-sized operands. However, over the last 20 years or so CPU clock rates have grown explosively, with an average annual rate of increase of $35 - 55\%$ [Hennessy and Patterson 1996]. As a result, nowadays even entry-level machines come with CPUs with clock frequencies of 400 Mhz or above. Unfortunately, the speeds of main memory have not increased as

Address: Department of Computer Science, King's College London, Strand, London WC2R 2LS, U. K. e-mail: {naila, raman}@dcs.kcl.ac.uk
This work was supported in part by grant GR/L92150 from the UK Engineering and Physical Sciences Research Council (EPSRC).
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.

rapidly, and today's main memory typically has a latency of about 70ns. Hence, a conservative estimate is that a main memory access can take 30+ CPU clock cycles.

In order to overcome this difference in speeds, modern computers have a *memory hierarchy* which inserts multiple levels of *cache* between CPU and main memory. A cache is a fast associative memory which holds the values of some main memory locations. If the CPU requests the contents of a memory location, and the value of that location is held in some level of cache, the CPU's request is answered by the cache itself (a cache *hit*); otherwise it is answered by consulting the main memory (a cache *miss*). A cache hit has small or no penalty (1-3 cycles is fairly typical) but a cache miss is very expensive.

Having a hierarchy of memories is not new: when dealing with very large input sizes, which are considerably larger than main memory, it is well known that algorithms which explicitly take into account the difference in speeds between main memory and external (disk) memory can considerably outperform algorithms which ignore this difference. A large number of external-memory algorithms have been developed, mostly using refinements of a model introduced in [Aggarwal and Vitter 1988]. However, there are significant differences between the two frameworks, a major one being the cost of a disk access (which could be tens or hundreds of thousands of CPU cycles) versus the cost of a cache miss. If one wishes to obtain the best practical performance on problem instances which fit into main memory, it can be better to tune the cache performance of proven internal-memory algorithms as was shown by [LaMarca and Ladner 1999] and as we demonstrate here in the context of *distribution* sorting.

Distribution sorting is a popular technique for sorting data which is assumed to be randomly distributed, and involves distributing the n input keys into m *classes* based on their value. The classes are chosen so that all the keys in the ith class are smaller than all the keys in the $(i+1)$st class, for $i = 0, \dots, m-2$, and furthermore, the class to which a key belongs can be computed in $O(1)$ time. In $O(n)$ time, the problem is reduced to sorting the $m \leq n$ classes. Distribution sorting can be used to sort independent randomly distributed keys in $O(n)$ time on average [Knuth 1997, Ch 5.2, 5.2.1], which is asymptotically faster than the $\Theta(n \log n)$ average running time of comparison-based approaches. An important special case is when the keys are drawn independently at random from a uniform distribution ('uniform keys').

Neubert [Neubert 1998] presented an implementation of a distribution sorting algorithm *Flashsort1*, which used a combination of a well-known counting method and an 'in-place' permutation method similar to one described in [Knuth 1997, Soln 5.2-13]. With Neubert's choice of parameters Flashsort1 uses only $n/10$ words of memory in addition to the memory required by the data, and sorts n uniform floating-point keys in $O(n)$ time. Neubert's experiments showed that his implementation of Flashsort1 was twice as fast as Quicksort when sorting about 10,000 keys. In the RAM model, the lower asymptotic growth rate of Flashsort1 and the fact that Flashsort1 outperforms Quicksort for $n = 10,000$ would indicate that Flashsort1 would continue to outperform Quicksort for larger values of n. Unfortunately, this is not the case. We translated Neubert's FORTRAN code for Flashsort1 into C++ and performed extensive experiments, see Figure 3, which clearly indicated that although Flashsort1 was significantly faster than Quicksort for n in the range 4K to 128K, Quicksort caught up with and surpassed Flashsort1 at 1M keys. In

fact, the ratio of the running times of Flashsort1 to Quicksort continued to grow with n, up to $n = 64$M.

Our simulations verify that this is due to the poor cache performance of Flashsort1. We present an analysis of the cache behaviour of 'in-place' distribution sorting algorithms for uniform keys. The analysis holds for a wide range of parameter choices, including those of Neubert, and assumes a direct-mapped single-level cache. The analysis is approximate in that it simplifies the underlying probability distributions, but our simulations show that the formulae that we obtain accurately predict cache misses to within a few percent in most cases. In some cases, however, the number of misses is much higher than is predicted by our formulae. We analyse the reasons why our predictions are inaccurate in these cases, and we suggest heuristics (Sections 4.4.2 and 4.4.3) for avoiding these 'bad' cases. One of these heuristics appears to work well. Using the analyses, we determine that when sorting n uniform single-precision floating-point keys, for roughly $600000 \leq n \leq 5.3 \times 10^9$ one should do the distribution in two passes on our test platform (more passes are necessary for larger values of n). By incorporating these heuristics we obtain a distribution sorting algorithm multi-pass Flashsort (*MPFlashsort*) which is over twice as fast as Flashsort1 for medium to large n, even though it can perform many more operations than Flashsort1.

We compare these distribution sorting algorithms against a multi-pass implementation of most-significant-bit first radix sort (MSB radix sort) [Knuth 1997]. This exploits the well-known fact that in the IEEE 754 standard, the relative order of two non-negative floating-point numbers is the same as the order of the bit-strings which represent them [Hennessy and Patterson 1996]. As a uniform distribution on the floats does not induce a uniform distribution on the underlying integers, it is not clear *a priori* that MSB radix sort is a good algorithm in this context. However, we observe that this implementation performs well, as it uses fast integer operations such as shifts and masks and also appears to have good cache utilisation.

We also compare these algorithms against cache-tuned implementations of comparison-based sorting algorithms. We considered all implementations of Quicksort and Mergesort by [LaMarca and Ladner 1999] and Heapsort by [Sanders 1999]. On our machine Memory-Tuned Quicksort (MTQuicksort) was the fastest in-place algorithm, marginally faster than Sedgewick's Quicksort and at least 8% faster than all other Quicksort or Mergesort implementations. Heapsort using Sanders' cache-tuned heaps [Sanders 1999] seemed to be the fastest out-of-place algorithm, and its performance seemed roughly comparable to MTQuicksort.

All our distribution sorting algorithms here are 'in-place' in that they use very little additional space. We note that both MPFlashsort and MSB radix sort clearly outperform MTQuicksort even when the data to be sorted is 'small', i.e. single-precision (4-byte) floats. For double precision floats (8-byte data) the gap between MPFlashsort and MTQuicksort is quite large[1].

[1] As our machine does not have native support for 64-bit integers we did not test MSB radix sort with double precision floats

2. PRELIMINARIES

This section introduces some terminology and notation regarding caches. The size of the cache is normally expressed in terms of two parameters, the *block size* (B) and the number of *cache blocks* (C). We consider main memory as being divided into equal-sized *blocks* consisting of B consecutively-numbered memory locations, with blocks starting at locations which are multiples of B. The cache is also divided into blocks of size B; one cache block can hold the value of exactly one memory block. Data is moved to and from main memory only as blocks.

In a *direct-mapped* cache, the value of memory location x can only be stored in cache block $c = (x \text{ div } B) \bmod C$. If the CPU accesses location x and cache block c holds the values from x's block the access is a cache hit; otherwise it is a cache miss and the contents of the block containing x are copied into cache block c, *evicting* the current contents of cache block c. For our purposes, cache misses can be classified into *compulsory misses*, which occur when a memory block is accessed for the first time, and *conflict misses*, which happen when a block is evicted from cache because another memory block that mapped to the same cache block was accessed.

An important consideration is what happens when a value is written to a location stored in cache. If the cache is *write-through* the value is simultaneously updated in the cache and in the next lower level of the memory hierarchy; if the cache is *write-back*, the change is recorded only in cache. Of course, when a block is evicted from a write-back cache it must be copied to the next lower level of the hierarchy.

We performed our experiments on a Sun UltraSparc-II, which has a blocksize of 64 bytes, a L1 cache size of 16KB and a L2 cache size of 512KB [2]. Both L1 and L2 caches are direct-mapped, the L1 cache is write-through and the L2 cache is write-back. As our programs hardly ever read a memory location without immediately modifying it, the L1 cache is ineffective for our programs and we focus on the L2 cache. Hence, for our machine's L2 cache we have $C = 8192$. This paper deals mainly with single-precision floating-point numbers and integers, both of which are 4 bytes long on this system. It is useful to express B in terms of the number of 'items' (integers or floats) which fit in a block; hence we use $B = 16$ in what follows.

3. OVERVIEW OF ALGORITHMS

We now describe the steps in one pass of a generic 'in-place' distribution sorting algorithm. This description embraces Flashsort1, MPFlashsort and MSB radix sort. After the description we explain how to obtain the algorithms Flashsort1 and MSB radix sort, but the description of MPFlashsort is given only after the cache analysis. While describing these algorithms, the term *data* array refers to the array holding the input keys, and the terms *count* and *start* refer to auxiliary arrays used by these algorithms. After one pass of an 'in-place' distribution sorting algorithm, the data array should have been permuted so that all elements of class k lie consecutively before all elements of class $k+1$, for $k = 0, \ldots, m-2$. The pseudocode that accomplishes this is given in Fig 1. In the pseudocode it is assumed that m has been appropriately initialised, and that the function `classify` maps a key to a

[2] K = 1024 and M = 1024K in this paper.

(a) A count phase
```
1 for i := 0 to m − 1 do
    COUNT[i] := 0;
2 for i := 0 to n − 1 do
    COUNT[classify(DATA[i])]++;
3 COUNT[m − 1] := n
         - COUNT[m − 1];
4 for i := m − 2 downto 0 do
    COUNT[i] := COUNT[i + 1]
         - COUNT[i];
```

(b) A permute phase
```
1 leader := n − 1;
2 idx := leader; x := DATA[idx];
3 c := classify(x); idx := COUNT[c];
  COUNT[c]++; swap x and DATA[idx];
  if idx ≠ leader repeat 3;
4 while (c > 0 ∧ COUNT[c − 1] ≥ START[c])
    c--;
5 if (c > 0) leader := START[c]−1;
    go to 2;
```

Fig. 1. Count and permute phases for an 'in-place' distribution sorting algorithm. DATA holds the input keys. COUNT and START are auxiliary arrays, and COUNT is copied into START at the end of the count phase.

class numbered $\{0, \ldots, m-1\}$ in $O(1)$ time.

It is convenient to divide a pass into a *count* phase and a *permute* phase. The count phase consists of four steps. Step 1 initialises each element of COUNT to zero. Step 2 counts the frequency of keys in each class by applying the classify function. Steps 3 and 4 perform a prefix-summation of the count array, after which $\texttt{COUNT[0]} = 0$ and for $i = 1, \ldots, m-1$, $\texttt{COUNT}[i]$ holds the number of keys in classes $0, \ldots, i-1$.

Before the permute phase begins, a copy of the count array is made in the start array. During the permute phase, for any class k, an invariant is that locations $\texttt{START}[k], \texttt{START}[k]+1, \ldots, \texttt{COUNT}[k]-1$ contain elements of class k, i.e. $\texttt{COUNT}[k]$ points to the leftmost (lowest-numbered) available location for an element of class k. Thus, for $k = 0, \ldots, m-2$, all elements of class k have been permuted if $\texttt{COUNT}[k] \geq \texttt{START}[k+1]$, and such a class will be called *complete* in what follows. Class $m-1$ is complete when $\texttt{COUNT}[m-1] \geq n$.

We will now describe the permute phase, which consists of two main activities: *cycle following* and *cycle leader finding*. In cycle following, keys are moved to their final destinations in the data array along a cycle in the permutation (Steps 2 and 3). Once a cycle is completed, we move to cycle leader finding, where we find the 'leader' (index of the rightmost element) of the next cycle (Steps 1, 4 and 5). A cycle leader is simply the rightmost location of the highest-numbered incomplete class. By the definition of a complete class, initially the leader must be position $n-1$. In more detail, the steps are as follows:

—In Step 1 $n-1$ is selected as the the first cycle leader.

—In Step 2 the key at the leader's position is copied into the variable x, thus leaving a 'hole' in the leader's position.

—In Step 3 the key x is swapped with the key at x's final position. If x 'fills the hole', the cycle is complete, otherwise we repeat this step.

—In Step 4 the algorithm searches for a new cycle leader. Suppose the leader of the cycle which just completed was the last location of class k. When this cycle ends, class k must also be complete, as a key of class k has been moved into the last location of class k. Note that classes $k+1, k+2, \ldots$ must already have been complete when the leader of this cycle was found. Note that the program variable c has value k at the end of this cycle, so the search for the next leader

begins with class $k-1$, counting down (Step 4).

—In Step 5 we check to see if all classes have completed and terminate if this is the case.

Clearly the count and permute phases take $O(n)$ time whenever $m \leq n$. We now describe the algorithms Flashsort1 and MSB radix sort.

3.1 Flashsort1

In Flashsort1, it is assumed that the keys are real numbers drawn from a uniform distribution. Flashsort1 first scans the input numbers to determine the largest (max) and smallest (min) values in the input. After this, we perform one distribution pass into $m \geq 2$ classes, using the function

$$\texttt{classify}(x) = \left\lfloor m \cdot \frac{x - min}{max - min} \right\rfloor .$$

Finally, the data array is sorted using insertion sort. If $m \leq n$ the total expected cost of the insertion sort is easily seen to be $O(n^2/m)$, while the rest of the algorithm evidently runs in $O(n)$ time. The algorithm uses $2m$ extra memory locations[3]. After experimentation Neubert chose $m = n/10$ to minimise extra memory while maintaining a near-minimum expected running time [Neubert 1998]. However, for large values of n, it is intuitively apparent that the cache performance of Flashsort1 will be poor: e.g., for $n = 64$M the count array will be approximately 50 times the size of cache, which suggests that almost all accesses to the count array will be cache misses.

3.2 MSB radix sort

Most significant bit (MSB) radix sort is a distribution sorting algorithm for integers. In one pass of MSB radix sort with radix r for some integer $r \geq 1$, we distribute the keys into $m = 2^r$ classes by letting $\texttt{classify}(x)$ return the value of the r most significant bits of x. Each class is then either sorted recursively or by insertion sort. In a recursive sub-problem, the radix r' may be chosen afresh, and also it will be the case that all keys in this recursive sub-problem will have a common value in some of their most significant bits: $\texttt{classify}(x)$ should then skip over the common bits and return the next most significant r' bits.

Although it is an integer sorting algorithm, MSB radix sort can sort floating point numbers. It is well-known that if the floating point numbers are represented according to the IEEE 754 standard, then the [bit-strings viewed as] integers that represent two floating point numbers have the same ordering as the floats themselves, at least if both the numbers are non-negative [Hennessy and Patterson 1996]. Hence, one can sort floats by sorting the representing integers using MSB radix sort. It is important to note that a uniform distribution on floating point numbers does *not* induce a uniform distribution on the representing integers. For example, if the keys are uniform numbers in the range $[0, 1)$, half the numbers will have value in $[0.5, 1)$; after normalisation their exponents will all be 0. This means that in the IEEE 754 standard representation of single-precision floating-point numbers, half

[3]In fact, Neubert's code had a more complex procedure for cycle leader finding which used only m extra memory locations. As this was significantly slower we did not implement it.

the keys will have the pattern `001111110` in their most significant 9 bits. Due to this non-uniform distribution, the approach used in this paper to approximately analyse the cache misses of Flashsort1 and MPFlashsort does not extend to sorting uniform floats using MSB radix sort.[4]

Here we present a somewhat *ad hoc* implementation of MSB radix sort. We first perform a distribution according to the most significant r bits, where $r = \min\{\lceil \log n \rceil - 2, 16\}$ and n is the number of keys to be sorted. Each class consisting of $n' > 40$ keys is then sorted recursively using MSB radix sort with radix $r' = \min\{\lceil \log n' \rceil - 2, 16\}$. Classes consisting of 40 or fewer keys are solved with insertion sort.

4. CACHE ANALYSIS

In this section we analyse the cache behaviour of Flashsort1 and some variants. Our analysis is actually more general in that it applies to one pass of the generic 'in-place' distribution sorting algorithm described in Section 3 for uniform keys. More precisely, we require that `classify`(x) returns each value in $\{0, \ldots, m-1\}$ with equal probability for a randomly generated key x [5]. For example, it applies also to MSB radix sort applied to uniform integers. However, the parameters for which the analysis holds are guided by Flashsort1 and its variants. Also, a complete cache analysis also must include algorithm-specific details such as the minimum and maximum finding phase for Flashsort1. Hence, the following description is tailored specifically to Flashsort1 and its variants.

We will assume that the input size is an integral multiple of CB if $n \geq CB$ and that important arrays begin at locations which are multiples of B. A memory block in the data array will be called a *data block* and one in the count array will be called a *count block*. From the above, the data array consists of a number of complete data blocks.

The analysis assumes the cache is single-level and direct-mapped. For the sake of simplicity we will assume that the various phases are independent, i.e. the cache is emptied after each phase. This assumption causes inaccuracies for small input sizes, where a significant part of the input may stay in cache between say the count and permute phases, so we report only the predicted values for $n \geq 2CB = 256$K. Throughout we assume that $m \leq n$.

In each case we calculate the expected number of cache misses per key for a given phase of the algorithm, which we will denoted by μ with a subscript to indicate the phase of the algorithm. In some cases the analyses are approximate, in that a simplified probabilistic process is analysed. Hence, the formulae that we obtain may be higher or lower than the actual expected number of misses, and we do not attempt to mathematically quantify the extent of any errors. However, we do validate the formulae by simulations. We use the notation $\mu :=$ to highlight cases where the analysis is approximate in the above sense.

The analysis of the count phase uses the results of [Ladner et al. 1999] to a large

[4]Recently the authors have analysed distribution sorting under non-uniform key distributions [Rahman and Raman 2000].

[5]The `classify` function for Flashsort1 does not generate a perfectly uniform distribution—e.g. only the maximum-valued key is in class $m-1$—but this effect is ignored in the analysis.

extent.

4.1 Determining the range of the keys

Before the count phase Flashsort1 sequentially accesses n elements of the data array to determine the minimum and maximum valued keys. Since n/B data blocks must be loaded into cache, this step requires $1/B$ cache misses per key. Hence:

$$\mu_{range} = 1/B. \tag{1}$$

4.2 The count phase

We now give three cases which describe the cache misses during the count phase. The results were obtained by slight modifications to the results in [Ladner et al. 1999] and details can be found in Appendix A.

Case 1: $n > CB$ and $m > CB$. In this case the data and count arrays are larger than the cache and:

$$\begin{aligned}\mu_{count} :=& \frac{1}{B} + \frac{m - CB}{m} + \\ & \frac{C}{m}\left[1 - \left(1 - \frac{1}{C}\right)^{CB-B+1}\right] + \frac{(B-1)(m+CB)}{CBm} + \frac{2m/B}{n}.\end{aligned} \tag{2}$$

Case 2: $n > CB$ and $m \leq CB$. In this case the data array is larger than the cache but the count array fits in cache and:

$$\mu_{count} := \frac{1}{B} + \frac{m/B}{CB}\left[1 - \left(1 - \frac{B}{m}\right)^{CB-B+1}\right] + \frac{2(B-1)}{CB} + \frac{2m/B}{n}. \tag{3}$$

Case 3: $n \leq CB$ and $m \leq CB$. In this case the data and count arrays fit in cache and:

$$\mu_{count} := \frac{1}{B} + \frac{m/B}{n} + \frac{2(B-1)}{CB} + \frac{m/B}{CB}. \tag{4}$$

4.3 The insertion sort phase

Given that the problems which need to be solved are of expected size ≤ 10 in Flashsort1 it is extremely unlikely in practice that any problem will exceed the size of the cache. As we will see, the same holds for MPFlashsort. Hence we allocate $1/B$ misses per key for this phase:

$$\mu_{ins_sort} := 1/B. \tag{5}$$

(MPFlashsort may incur no additional misses for the insertion sort phase for some values of n.)

4.4 The permute phase

In this section we present an analysis of the permute phase of Flashsort1 and its variants. The permute phase can be viewed as alternating cycle-following with finding cycle leaders. During the cycle following phase we make $2n$ memory accesses which may lead to cache misses. These comprise alternating accesses to the count

array and to the data array. Any access to the data array must be to one of the m *active locations* pointed to by the *data pointers* for the m classes (the data pointer for class j points to the next available location for a key of class j). It is not hard to see that for any fixed i, the ith access to the count array is equally likely to be to any of the m count array locations, independently of the previous accesses to the count array; similarly, the ith access to the data array is equally likely to access any of the m active locations, independently of the previous accesses to the data array.

However, this is not enough to analyse the permute phase exactly. For example, it appears difficult even to get a good understanding of the distribution of the active locations at the start of the permute phase. Hence, we use an approximate analysis. This analysis uses the idea that although the active locations change as keys are moved to their destinations within classes, the movement is at roughly the same rate and thus the active locations should maintain roughly the same relative position. Therefore, we fix some 'typical' positions of active locations, giving a 'snapshot' of the algorithm, and analyse a simple process which involves accesses to the count array and to the fixed active locations. Of course, the accuracy of the formulae we get will depend heavily on how well we fix the active locations, and we also ignore other accesses, such as those for the cycle leader finding phase.

In what follows, an *active block* is either a count block or a data block that has an active location in it. Having fixed the active locations, we consider a sequence of memory accesses, where each access reads or writes one of the m count array locations or one of the m active locations with equal probability $1/m$, independently of the previous accesses[6]. We calculate the probability p_d of a miss if an active location is accessed, and the probability p_c of a miss if a count array location is accessed; we then estimate:

$$\mu_{perm} := 1/B + (1 - 1/B)p_d + p_c. \tag{6}$$

The reason for using this formula is that p_d only estimates the probability of misses on data array accesses due to conflicts between active blocks. However, the n accesses to the data array cover n distinct locations in all, resulting in at least n/B compulsory misses. Thus, p_d should be applied only to the approximately $n(1 - 1/B)$ accesses that remain.

Using the probabilities specified in the above process, and having fixed the active locations, we can easily calculate the probability that the process accesses a given active block. For example, we can infer that a count block has an access probability of B/m. After this, the analyses make extensive use of the following elementary idea [LaMarca and Ladner 1999]. Suppose we consider a process which accesses k blocks of memory $b_1, \ldots, b_k$, with each access being made independently to block b_i with probability p_i, for $i = 1, \ldots, k$. Let $S \subseteq \{1, \ldots, k\}$ and suppose that exactly the blocks $\{b_i | i \in S\}$ are mapped to some cache block c. Let c be randomly initialised to b_i, $i \in S$ with probability p_i. Then, at any stage in the process, for any $i \in S$:

$$\Pr[b_i \text{ is in cache block } c] = \frac{p_i}{\sum_{j \in S} p_j}. \tag{7}$$

[6]These probabilities add up to 2, and should be normalised to $1/(2m)$ each. Since we are only interested in the ratios of probabilities, we omit the normalisation to improve readability.

An arbitrary initialisation of the cache may result in at most one more miss per cache block than given by the above equation, which is negligible. Equation 7 implies the following as well. If there are τ data blocks mapped to a cache block and the number of active locations in data block $1, \ldots, \tau$ is $n_1, n_2, \ldots, n_\tau$ and $N = \sum_{i=1}^{\tau} n_i$ then the probability of a hit in accesses to that cache block is:

$$\frac{1}{N^2} \sum_{i=1}^{\tau} {n_i}^2. \tag{8}$$

For a given value of N, by Jensen's inequality (see e.g. [Rudin 1974]), the probability of a hit is minimised when $n_i = N/\tau$ for $i = 1, \ldots, \tau$. A similar calculation can be made when τ data blocks, with N active locations in all, conflict with one or more count blocks.

In order to fix the active locations, we consider three cases; these cases cover the possible parameter choices in our two Flashsort variants. The first pair of cases deal with the situation that the expected class sizes are smaller than the block size B. If so, it will often be the case that there are multiple active locations in a single data block. The third case deals with the situation where the expected class sizes are much larger than B. In this case it is (intuitively) very unlikely that two active locations are present in a single data block.

Case 1: $m \leq CB$ and $n/m < B$. In this case the count array fits in cache and the expected class size is less than a cache block. We assume that there are $p = m/(n/B)$ data pointers in each data block, allowing p to be non-integral. By the reasoning associated with Equation 8, this should overestimate the number of misses. There are two regions in the cache: region R_1 which has no count blocks mapped to it, and region R_2 which has count blocks mapped to it.

We first consider the case $n \geq CB$. In this case each cache block in R_1 has $\tau = n/(CB)$ data array blocks mapped to it, whereas each block in R_2 has one count block and τ data blocks mapped to it (recall that τ is assumed to be integral). The probability of accessing a count array block is B/m and of accessing any data block is $p \cdot (1/m) = B/n$. For any cache block c in R_2, an access to a data block which is mapped to c is a hit only if that data block is currently stored in c. As the sum of the probabilities of the blocks mapped to c is $B/m + \tau \cdot B/n$, by Equation 7 we have:

$$\Pr[\text{Data access in } R_2 \text{ is a hit}] := \frac{B/n}{B/m + \tau \cdot B/n} = \frac{1}{n/m + \tau}. \tag{9}$$

Similarly, we get:

$$\Pr[\text{Count access in } R_2 \text{ is a hit}] := \frac{B/m}{B/m + \tau \cdot B/n} = \frac{B}{B + m/C}. \tag{10}$$

In region R_1, there are τ active blocks (from the data array) mapped to each cache block. As each active block has the same probability of being accessed, we get that $\Pr[\text{Data access in } R_1 \text{ is a hit}] = \tau^{-1}$. Since a data access goes to R_1 with probability $1 - m/(CB)$, and R_2 with probability m/CB, we have:

$$p_d := \left(1 - \frac{m}{CB}\right)\left(1 - \frac{1}{\tau}\right) + \frac{m}{CB}\left(1 - \frac{1}{n/m + \tau}\right). \tag{11}$$

Input size	256K	512K	1M	2M	4M	8M	16M	32M	64M
Predicted	0.7760	1.1183	1.3793	1.6582	1.8291	1.9146	1.9573	1.9786	1.9893
Simulated	0.7919	1.1446	1.4079	1.6950	1.8627	1.9482	1.9813	2.0178	2.0187

Fig. 2. Predicted and simulated miss rates for permute phase of Flashsort1. For details of the simulation framework, see Section 6

As there are no count array accesses in R_1, p_c is given by Equation 10. By Equation 6,:

$$\mu_{perm} := \frac{1}{B} + \left(1 - \frac{B}{B + m/C}\right) + \left(\frac{B-1}{B}\right)\left[\left(1 - \frac{m}{CB}\right)\left(1 - \frac{1}{\tau}\right) + \frac{m}{CB}\left(1 - \frac{1}{n/m + \tau}\right)\right], \quad (12)$$

if $n \geq CB$, and where $\tau = n/(CB)$. For Flashsort1, this formula explains data points for 256K to 1M as shown in Fig 2.

We now consider the case $n < CB$. In this case data blocks cannot conflict with each other so there are no cache misses in R_1. To determine the cache misses in R_2 we assume the start of the data array is uniformly and randomly distributed at cache block boundaries so the probability of a data block being mapped to a cache block in region R_2 is $\frac{m/B}{C}$. Using the above reasoning, if a data block is mapped to R_2 the probability that an access to it results in a hit is $\frac{B/n}{B/m+B/n} = \frac{1}{n/m+1}$. This gives $p_d = \frac{m}{CB}\left(1 - \frac{1}{n/m+1}\right)$. A given count block has probability $\frac{n/B}{C}$ of being in conflict with a data block. In case there is a conflict, the probability of a hit on a count array access is $B/(B + m/C)$. This gives $p_c = \frac{n}{CB}\left(1 - \frac{B}{B+m/C}\right)$. Using Equation 6 we get:

$$\mu_{perm} := \frac{1}{B} + \frac{n}{CB}\left(1 - \frac{B}{B + m/C}\right) + \left(\frac{B-1}{B}\right)\left[\frac{m}{CB}\left(1 - \frac{1}{n/m + 1}\right)\right], \quad (13)$$

if $n < CB$. Equation 13 is applicable to the last pass of MPFlashsort. The misses for a given sorting problem may be more or less than the expected value, depending on the exact number of data blocks in R_2. However, averaged over all problems the result should be accurate since they are in consecutive segments of the data array so an underestimate for some problems will lead to an overestimate for other problems.

Case 2: $m > CB$ and $n/m < B$. In this case the size of the count array exceeds the cache size and the expected class size is less than a cache block. Equation 14 is derived in a manner similar to Case 1. First, we assume that m is a multiple of CB, and apply the above reasoning, taking region R_1 to be empty and R_2 as the whole cache. Then we use the same formula when m is not a multiple of CB. The final formula is:

$$\mu_{perm} := 1/B + (1 - CB/(2m)) + ((B-1)/B) \cdot (1 - CB/(2n)). \quad (14)$$

For Flashsort1, this formula explains data points for 2M to 64M as shown in Fig 2.

Case 3: $m \leq CB$ and $n/m \gg B$. In this case the count array fits in cache and the expected class size is much larger than a cache block. Since $n/m \gg B$ we assume that there is at most one data pointer per data block. Again, we divide the cache into two regions: region R_1 which has no count blocks mapped to it, and region R_2 which has count blocks mapped to it. Note that R_2 occupies a $m/(CB)$ fraction of the cache. We assume that $N = m \cdot (m/(CB))$ data pointers are mapped to R_2. Again, by Jensen's inequality, the overall hit rate is minimised when each of the m/B blocks comprising R_2 has $N/(m/B) = m/C = \rho$ data pointers mapped to it. We proceed with the calculation based upon ρ data pointers mapped to each cache block in R_2, each with an access probability of $1/m$; note, however, that ρ may be non-integral, and possibly even much smaller than 1. Consider a cache block i in R_2. The count block mapped to i is accessed with probability B/m, and each of the ρ data blocks mapped to i are accessed with probability $1/m$. The sum of the access probabilities of the blocks mapped to i is $B/m + \rho \cdot (1/m) = B/m + 1/C$. Thus:

$$\Pr[\text{Count access in } R_2 \text{ is a hit}] := \frac{B/m}{B/m + 1/C} = \frac{B}{B+\rho}, \tag{15}$$

$$\Pr[\text{Data access in } R_2 \text{ is a hit}] := \frac{1/m}{B/m + 1/C} = \frac{1}{B+\rho}. \tag{16}$$

It is more interesting to study the hit rate in region R_1, which only has data pointers mapped to it. It will be convenient to assume that R_1 covers the entire cache, and has m data pointers mapped to it; as we will see, we can scale down the values without changing the hit rate. Let the number of data pointers mapped to cache block i be m_i. If cache block i has $m_i \neq 0$, then the probability that a data array access reads cache block i is m_i/m, but the probability of this access being a hit is $1/m_i$. Hence, the probability of a hit given that cache block i was accessed is $1/m$. Summing over all i such that $m_i \neq 0$ gives the overall hit rate as simply ν/m, where ν is the number of cache blocks such that $m_i \neq 0$. Assuming that the data pointers are independently and uniformly located in cache blocks, we calculate the expected value of ν as:

$$C \cdot (1 - (1 - 1/C)^m) \approx C \cdot (1 - e^{-\rho}). \tag{17}$$

Hence we have:

$$\Pr[\text{Data access in } R_1 \text{ a hit}] := \nu/m = \rho^{-1}(1 - e^{-\rho}), \tag{18}$$

and note that this is invariant to scaling C and m by the same amount. We now derive the misses per key for this case. First, the probability that a data access is to R_1 is $m/(CB)$; using Equations 18 and 16, we get that $p_d = 1 - (m/(CB)) \cdot 1/(B+\rho) - (1 - m/(CB)) \cdot \rho^{-1}(1 - e^{-\rho})$, and hence using Equations 6 and 15 we get that:

$$\begin{aligned} \mu_{perm} := {} & \frac{1}{B} + \left(1 - \frac{B}{B+\rho}\right) + \\ & \frac{B-1}{B}\left[1 - \frac{m}{CB}\left(\frac{1}{B+\rho}\right) - \left(1 - \frac{m}{CB}\right)\left(\rho^{-1}(1 - e^{-\rho})\right)\right], \end{aligned} \tag{19}$$

where $\rho = m/C$.

Case 4: $m > CB$ and $n/m \gg B$. Finally, for the sake of completeness we study the case where $m > CB$ and $n/m \gg B$. The approach is similar to that of Case 2: we assume that m is a multiple of CB, and use the formula even in the case that it is not. If m is a multiple of CB, then we have $m/(CB)$ count blocks mapped to each cache block, along with $\rho = m/C$ active locations; this gives the probability of a count hit as $\frac{B}{\rho + B \cdot m/(CB)} = CB/(2m)$. The probability of a data hit is $\frac{1}{\rho + B \cdot m/(CB)} = C/(2m)$. Thus:

$$\mu_{perm} := 1/B + (1 - CB/(2m)) + ((B-1)/B)(1 - C/(2m)), \tag{20}$$

where $\rho = m/C$.

4.4.1 *Accuracy of predictions.* In Appendix B we summarise extensive experiments which study the cache misses during the permute phase in Flashsort1 variants where $m \leq CB$ and $n/m \gg B$. Figure 11 in Appendix B shows cache misses per key during the permute phase for $n = 2^{24}$ and 100 random values of m between 25 and 8192 and for some selected values of m. Excluding the selected values of m we see that the values predicted using Equation 19 are quite close to the simulated values, indicating that the equation seems to be accurate for most values of m. However, the equations are very inaccurate for the selected values of m, which are: 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2047, 2048, 2049, 4095, 4096 and 4097.

This suggests that the assumption of random data pointer placement is not always a good one. We now analyse this further. To simplify the analysis, we make the assumption that a 'typical' value for the number of non-empty blocks, ν, is simply the value of ν at the start of the permute phase. As argued before, one would expect all data pointers to move at roughly the same rate, hence maintaining the original pattern of data pointers. Figure 11 can be used to validate this. We see that, for most m, the misses per key predicted using actual ν at the start of the permute phase are quite close to the simulated values. From now onwards, we will focus the discussion on the initial value of ν.

We first note that it is surprising that predictions made by assuming a uniform and independent mapping of data blocks are at all accurate, since these assumptions are manifestly untrue. The expected starting location of the k-th data pointer in the data array is precisely the expected number of elements in classes $0, \ldots, k-1$, and so it very much depends on the locations of the previous pointers. Furthermore, the number of elements in classes $0, \ldots, k-1$ is a binomial random variable with expected value $k \cdot n/m$, and is not uniformly distributed over cache blocks.

We now make some observations regarding the expected starting locations of the data pointers. If n is a multiple of CB then one can reason about this particularly simply. We view the cache as being 'continuous' and place m *marks* on the cache numbered $0, 1, \ldots, m-1$, with the i-th mark being $i \cdot (CB)/m$ words from the beginning of the cache. Letting $\tau = n/(CB)$, we get that the expected number of elements in classes $0, \ldots, k-1$ is $k \cdot n/m = (k\tau) \cdot (CB)/m$. Hence the expected starting location in cache of the k-th data pointer is seen to be at the mark numbered $(k\tau) \bmod m$.

The number of distinct marks among $0, \tau \bmod m, 2\tau \bmod m, \ldots, (m-1)\tau \bmod m$ depends upon the gcd of τ and m; more precisely, if $\gcd(\tau, m) = g$ then there will be m/g marks, each of which is the expected starting location of g data pointers. If g is large, one can expect the number of non-empty blocks to be quite small at the start of the permute phase. For example, in Figure 11, the prediction using Equation 19 is very inaccurate when $m = 512$. Since $n = 16\text{M} = 2^{24}$, giving $\tau = 128 = \gcd(\tau, m)$, and there are only $512/128 = 4$ different marks where the data pointers' expected locations lie, and one would expect ν to be very small at the outset.

However, it appears that it is difficult to obtain a simple closed-form expression for the initial value of ν for arbitrary m. One way out might be to choose m so that the analysis is made convenient. We now present two heuristics for selecting m and show that the first heuristic though plausible is not always effective, whereas the second appears to be effective.

4.4.2 *Heuristic 1 for selection of m.* If $\tau = n/CB$ is an integer, and we choose m to be relatively prime to τ, then we know that all the data pointers are expected to start at different marks. As the marks are precisely evenly spaced in the cache, it would appear that this heuristic is optimal. Unfortunately, the examples of $m = 2047$ or $m = 2049$ below, taken from Figure 11, show that this is not so:

m	Simulate	Predict	m	Simulate	Predict	m	Simulate	Predict
2047	0.245	0.198	2048	0.320	0.198	2049	0.241	0.198

In our experiment, $\tau = 128$ and $\gcd(2047, 128) = \gcd(2049, 128) = 1$. Although all the data pointers are expected to start at different marks, the miss rates for $m =$ 2047 and 2049 are still quite high. We note that the expected data pointer starting locations for $m = 2047$ are as follows:

mark	0	1	2	3	...	126	127	128	129	130	...
pointer	0	16	32	48	...	2016	2032	1	17	33	...

The standard deviation of the kth data pointer location is $\sqrt{n \cdot (k/m) \cdot (1 - k/m)}$. Thus, data pointers with low or high indices have low standard deviation, so the data pointers which will stay close to their expected positions are all clustered around marks $0, 128, 256 \ldots$, while the data pointers which are expected to start around mark $64, 192, 320, \ldots$ all have high standard deviation and can vary considerably from their expected starting position. Hence there is a concentration of pointers again around marks $0, 128, \ldots$. We see other occurrences of this at $m =$ 2239, 2945, 3841, 4095, 4096 and 5634, where $\gcd(\tau, m-1)$ or $\gcd(\tau, m+1)$ is large.

We do not know how to compensate for this additional requirement in this heuristic, but there are other serious disadvantages: one is that the heuristic only works when n is a multiple of CB, and another, more theoretical, objection is that for sufficiently large n, it may not be possible to pick m within a certain range of values which is relatively prime to τ. For instance, if we wish to pick a good m from the range $[m_1, \ldots, m_2]$ for integers m_1 and m_2, this is clearly impossible if τ is $m_1 \cdot (m_1 + 1) \cdot \ldots \cdot m_2$.

4.4.3 *Heuristic 2 for selection of m.* Another heuristic for selecting m comes from the following fact [Knuth 1997, pp. 550, Q8,9]. Let θ be an irrational number

which has the form $\frac{1}{k+\phi^{-1}}$ where $k \geq 1$ is an integer and $\phi = (1+\sqrt{5})/2$. If $\langle x \rangle$ denotes $x - \lfloor x \rfloor$, then for all $l \geq k$ the smallest segment of $[0,1]$ that is induced by the points $\langle \theta \rangle, \langle 2\theta \rangle, \ldots, \langle l\theta \rangle$ is at most $1+\phi$ times smaller than the largest segment. By symmetry, if θ is of the above form, then the intervals formed by $\langle -\theta \rangle, \langle -2\theta \rangle, \ldots, \langle -l\theta \rangle$ are also roughly evenly sized. We can use the above to get a 'good' value m^* in the rough vicinity of a given value m. The value m^* is chosen such that $n/m^* = \gamma CB$, where γ is chosen from:

$$\{1/(k+\phi^{-1}) \mid k = 1,2,3,\ldots\} \cup \{j \pm 1/(k+\phi^{-1}) \mid j = 1,2,\ldots \text{and } k = 1,\ldots,10\} \tag{21}$$

(There may be some duplication among these values: for instance $1/(1+\phi^{-1}) = 1-1/(2+\phi^{-1})$.) Suppose now that we choose $m^* = n/(\gamma CB)$, where $\langle \gamma \rangle = \pm 1/(r+\phi^{-1})$ for some integer $r \geq 1$. Again, if we consider the cache to be continuous, the ith class has an expected starting location which is offset $\langle i\gamma \rangle CB$ locations from the start of the cache. By the above fact, we may conclude that the expected starting locations of the classes are roughly evenly spaced in the cache, provided that $m^* \geq r$. Furthermore, all data pointers for classes $\geq r$ are evenly spaced with respect to the data pointers for lower-numbered classes.

The limit $k \leq 10$ in Equation 21 when $j \geq 1$ is somewhat arbitrary and is chosen so that m^* can always be chosen to within about 10% of m, provided that $n/m \geq 0.5CB$, as can easily be verified. As an example the 'good' values of n/m in the range $[0.5CB, 1.5CB]$ are (roughly) $0.62CB$, $0.72CB$, $0.79CB$, $0.82CB$, $1.18CB$, $1.21CB$, $1.28CB$ and $1.38CB$, so the nearest larger approximations to $n/m = CB$ and $n/m = 0.5CB$ are respectively $0.62CB$ and $1.18CB$. (When n/m is small we can remove the restriction on k, as the data pointers $0,\ldots,k-1$ anyway do not collide with each other.)

It should be noted that this gives a non-integral value of m, but the algorithm remains essentially unchanged. In the analysis, the last pointer now may have a lower access rate (as the final class may be small), but the effect appears to be negligible. The main advantages of this approach are: (i) it works even when n is not a multiple of CB and (ii) the low-numbered and high-numbered data pointers are also equally spaced. Figure 15 in Appendix B shows that this heuristic appears to place pointers more or less at random.

5. DESIGN OF MPFLASHSORT

In the previous section we saw that cache misses during both the count and permute phase are greatly influenced by the number of classes and it is clear that an algorithm which, when appropriate, uses fewer classes in a pass and applies distribution sorting recursively to keys within each class would out-perform Flashsort1. The algorithm should switch to a higher number of passes only when the reduction in cache misses can compensate for the increase in the number of operations. We now make this calculation. Firstly, we begin by estimating the relative cost δ of the "pure computation" cost of one distribution pass versus a cache miss. Inspecting the optimised assembly code produced by our compiler, `g++ 2.8.1` using optimisation level 6, there are approximately 30 operations per key in one distribution pass, and the cost of a L2 cache miss on our machine is approximately 30 cycles, so $\delta \approx 1$. (It should be noted that this calculation can only yield a rough guide, since

the underlying model is very crude.)

MPFlashsort uses one pass until the number of keys is larger than a threshold value T_1. The value of T_1 is selected to be minimum value of n such that:

$$M_1(z) > M_1(y) + M_2(x) + \delta. \tag{22}$$

Here $M_1(m)$ is the number of misses per item for sorting $10m$ keys in one pass by classifying into m classes, as described in cases 1 and 2. Similarly, $M_2(m)$ is the number of cache misses incurred in a distribution sorting pass using m classes as described in case 3. Recall that the analysis of case 3 is applicable only when $n/m \gg B$ and the value of m is not "bad". Hence we fix $x = n/(\theta CB)$ where $\theta = \frac{1}{1+1/\phi}$; since we want $xy = z = n/10$, this fixes $y = (\theta CB)/10$. Performing the calculation gives $z = 62153$, giving $n = 621530$, which is the value coded into MPFlashsort. For reference, in an earlier paper [Rahman and Raman 1999] we noted that a single pass with $m = n/10$ was slower than two passes with m approximately $n^{1/2}$ in each pass at $n \geq 2^{20} \approx 1,040,000$ but faster at $n \leq 2^{19} \approx 520,000$, so this rough calculation appears not too inaccurate. Another demonstration of the accuracy of our calculation may be found in the next section.

We now explain how to generalise this to multiple passes. We first set the expected problem size in the final distribution pass to θCB. Again letting $x = n/(\theta CB)$, we have to perform distribution passes until the expected class size is reduced by a factor of x. In the k-pass algorithm for $k \geq 2$, this is done in $k-1$ distribution passes with $x^{1/(k-1)}$ classes in each pass. The algorithm switches to $k+1$ passes when it is more efficient to reduce the expected problem size by a factor of x in k passes with $x^{1/k}$ classes in each pass. Thus, we switch from $k \geq 2$ to $k+1$ passes when:

$$(k-1)M_2(x^{1/(k-1)}) > k \cdot M_2(x^{1/k}) + \delta. \tag{23}$$

Doing the calculations for $k = 2$, we get that we should switch from two to three passes when $x > 65587$, corresponding to $n > x \cdot \theta CB \approx 5,313,000,000$.

It should be borne in mind that $M_2(m)$ does not predict cache misses well if m is 'bad', and selecting m according to Heuristic 2 is our only suggestion for avoiding 'bad' m. Hence, we modify the operation of the k-pass algorithm as follows. Instead of performing k passes with $x^{1/(k-1)}$ classes in each pass, the number of classes used in the passes are $m_1, m_2, \ldots, m_{k-1}$, which satisfy: (i) the expected problem size for the final pass is θCB as required; (ii) m_i is of the required form for all i, and (iii) m_i varies by no more than about 10% from $x^{1/(k-1)}$ for all i. With such a small variation in the class sizes, the calculations will continue to be broadly correct.

We illustrate how this may be done for $k = 3$. First, we should choose m_1 classes for the first pass, such that $0.9x^{1/2} \leq m_1 \leq 1.1x^{1/2}$, and m_1 is of the appropriate form. As noted in the text after Equation 21, this can always be done. The expected class sizes after the first pass are $n' = n/m_1$. In the second pass, we simply choose $m_2 = n'/(\theta CB)$ and note that $0.9m_2 \leq x^{1/2} \leq 1.1m_2$. The following element of approximation should be noted: the expected class sizes after the first pass are $n' = n/m_1$ for all classes but one (as m_1 is not an integer). The misses incurred by the 'small' class are ignored in the calculation.

Also, these threshold values of 62153 and 65587, which are approximately $8C$, should be contrasted with the fact that in the I/O model, the number of classes

in a pass should be no more than C [Aggarwal and Vitter 1988]. Furthermore, in order to get the optimal $\Theta\left(\frac{N}{B}\frac{\log(N/B)}{\log C}\right)$ number of cache misses [Aggarwal and Vitter 1988] in the presence of conflicts using a direct-mapped cache, we need to have no more than $O(C/B)$ classes in all passes but the last one (this is arrived at by trying to find ρ such that the bound in case 3 equals $O(1/B)$ misses per key). However, the above accounting shows that these choices would result in excessive operations, which would not be paid for by the reduced number of cache misses.

6. EXPERIMENTAL RESULTS

We have implemented Flashsort1, MPFlashsort and MSB radix sort, and have used them to sort n uniform floating-point numbers, for $n = 2^i$, $i = 10, 11, \ldots, 26$. Memory-tuned Quicksort (MTQuicksort) was found to be the fastest algorithm from [LaMarca and Ladner 1999] on our machine and we also tested this. We have also included some run-times from Sanders' heapsort (an out-of-place algorithm), but as we note below, it is not easy to compare the other algorithms with Sanders' heapsort.

Our algorithms and Sanders' heapsort were coded in C++ and MTQuicksort was coded in C. All algorithms were compiled using `g++ 2.8.1` with optimisation level 6. For each algorithm, we have measured actual running times as well as simulated numbers of cache misses. The running times were measured on a Sun UltraSparc-II with 2×300 Mhz processors and 512MB main memory. As mentioned in Section 2, this machine has a 16KB L1 data cache and a 512KB L2 cache, both of which are direct-mapped. Our simulator simulates only the L2 cache on this machine, and only reports cache hit/miss statistics. Each run time and simulation value reported in this section is the average of 50 runs.

Figure 3 summarises the running times for Flashsort1, MPFlashsort, MTQuicksort and MSB radix sort on single precision keys. We observe that:

—For small values of n Flashsort1 gets steadily faster than MTQuicksort until it uses about 75% of the time of MTQuicksort for $n = 64$K. After that the performance advantage narrows until at $n = 512$K MTQuicksort overtakes Flashsort1. From 1M onwards the relative gap between MTQuicksort and Flashsort1 (the ratio of the running times) generally increases steadily until the largest input value we considered. This is interesting given that MTQuicksort has a higher asymptotic running time than Flashsort1.

—The performance of MPFlashsort matches that of Flashsort1 upto $n = 512$K, after which point the algorithm performs two passes on the data and out-performs both Flashsort1 and MTQuicksort. At the largest value of n considered MPFlashsort is over twice as fast as Flashsort1. Note that the running time of MPFlashsort at n=1M is almost exactly twice the running time of MPFlashsort at n=512K, even though the algorithm switches from one to two passes. This suggests that the threshold value for switching from one to two passes has been calculated accurately.

—MSB radix sort out-performs the other algorithms for all values of n >2K shown.

Figure 5 summarises the running times for Flashsort1, MPFlashsort and MTQuicksort on 8-byte data, using double precison keys. This is a limited test

Timings (s) on UltraSparc-II, single precision keys				
n	Flash1	MPFlash	MTQuick	MSB radix
1K	0.0004	0.0004	0.0004	0.0004
2K	0.0007	0.0007	0.0008	0.0007
4K	0.0015	0.0015	0.0017	0.0013
8K	0.0032	0.0031	0.0037	0.0027
16K	0.0065	0.0064	0.0081	0.0055
32K	0.0134	0.0132	0.0175	0.0110
64K	0.0282	0.0280	0.0376	0.0225
128K	0.0634	0.0630	0.0800	0.0481
256K	0.1555	0.1547	0.1700	0.1209
512K	0.3623	0.3645	0.3591	0.2572
1M	0.9420	0.7102	0.7760	0.5440
2M	2.3250	1.4302	1.6630	1.1360
4M	5.2690	2.8622	3.5260	2.3348
8M	12.047	6.4995	7.4709	4.8096
16M	26.088	14.045	15.863	9.8264
32M	54.613	29.276	33.372	21.394
64M	124.80	59.852	69.969	47.041

Fig. 3. Running times of Flashsort1, MPFlashsort, MTQuicksort and MSB radix sort on a Sun UltraSparc-II using single-precision floating-point keys.

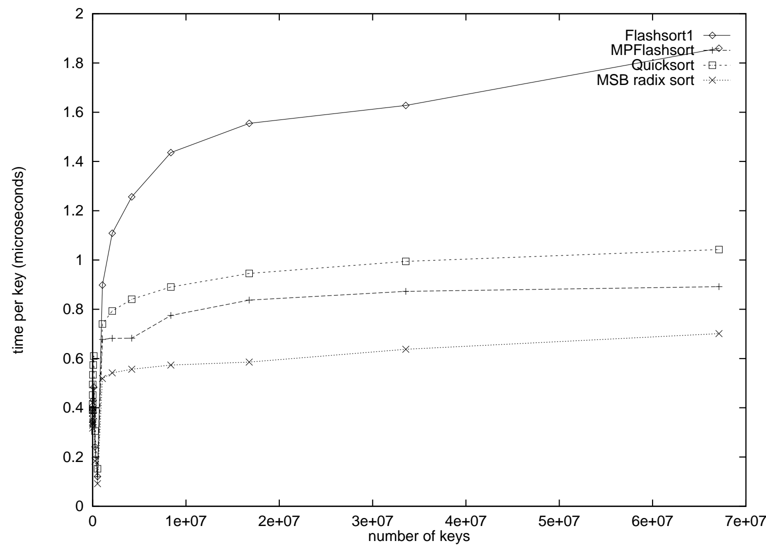

Fig. 4. Time per key to sort single-precision floating-point keys using Flashsort1, MPFlashsort, MTQuicksort and MSB radix sort on a Sun UltraSparc-II

in some ways. First of all, we could not include MSB radix sort as our platform does not offer native support for 64-bit integers. To apply MSB radix sort to 64-bit floating-point keys would require us to use emulated 64-bit integers which would slow down MSB radix sort. A second problem is that our analyses assume $B = 16$ throughout, but when using the Flashsort variants, we have $B = 16$ for the count array and $B = 8$ for the data array. Hence the one to two pass threshold for MPFlashsort is not as accurate as for the single-precision case. We note that:

—MTQuicksort starts to out-perform Flashsort1 later than for single-precision data, namely only for $n \geq 2$M.

—MPFlashsort is 35% faster than MTQuicksort for the largest values of n, a bigger performance gap than for the single-precision case.

—MPFlashsort and Flashsort1 are only slightly slowed down relative to the single-precision case, but MTQuicksort is signficantly slower. Although the distribution sorting algorithms are slowed down about 10-15% relative to the single-precision case, with no clear trend, MTQuicksort for doubles is about 35% slower than for floats at 64K, with the slowdown gradually increasing to about 50% at 32M. Possibly this is due to the fact that Quicksort makes $\Theta(n \log n)$ data moves but the distribution sorts make only $O(n)$ data moves.

Finally, we attempted comparisons with Sanders' heapsort, but it should be noted that Sanders' algorithm sorts 8-byte records with a key and a pointer, whereas our implementations are optimised for simply sorting keys. Indeed, the only commonality with Sanders' code is that 8-byte data are being sorted. Nevertheless, we have included these running times as a very rough guide.

6.1 Performance of MSB radix sort

We now try to understand the reasons behind the better performance of MSB radix sort. As discussed in Section 3.2 applying MSB radix sort to uniform floating-point numbers in the range $[0.0, 1.0)$ does not induce a uniform distribution of keys to classes, so a direct comparison to the Flashsort algorithms is difficult. However, if MSB radix sort is applied to uniform unsigned integers then its memory access patterns will be similar to the Flashsort variants. The other differences between the implementations of these algorithms are:

—the number of classes in the first pass is different;

—MPFlashsort has to determine the range of keys, but MSB radix sort does not,

—the classify functions are different.

We will now show that the performance is attributable to faster integer operations used in the classify function of MSB radix sort, by removing the other differences. For this, we modify MSB radix sort in the following manner:

—we determine the range of keys before the count and permute phases,

—we select the radix such that for a given value of n the number of classes is as close as possible to that selected by MPFlashsort.

At $n = 1.3 \times 10^6, 2.6 \times 10^6, 5.2 \times 10^6$, $10.4 \times 10^6, 20.8 \times 10^6, 41.5 \times 10^6$ and 83×10^6 MPFlashsort chooses radix values which are close to powers of two. For instance,

Timings (s) on UltraSparc-II				
8-byte keys				8-byte rec's
n	Flash1	MPFlash	MTQuick	Sanders
1K	0.0004	0.0004	0.0005	0.0007
2K	0.0008	0.0009	0.0010	0.0016
4K	0.0017	0.0017	0.0023	0.0036
8K	0.0034	0.0035	0.0051	0.0079
16K	0.0069	0.0070	0.0109	0.0164
32K	0.0144	0.0145	0.0238	0.0341
64K	0.0317	0.0320	0.0510	0.0693
128K	0.0765	0.0774	0.1093	0.1413
256K	0.1723	0.1733	0.2330	0.3016
512K	0.4463	0.4502	0.5099	0.6321
1M	1.0362	0.7922	1.1082	1.3088
2M	2.4700	1.5906	2.4014	2.7248
4M	5.5752	3.5402	5.1698	5.6428
8M	12.706	7.6008	11.050	11.704
16M	27.487	15.722	23.533	24.697
32M	60.917	32.227	50.111	50.472

Fig. 5. Running times of Flashsort1, MPFlashsort, MTQuicksort using 8-byte (double-precision floating-point) keys and Sanders' heapsort using 8-byte records on a Sun UltraSparc-II

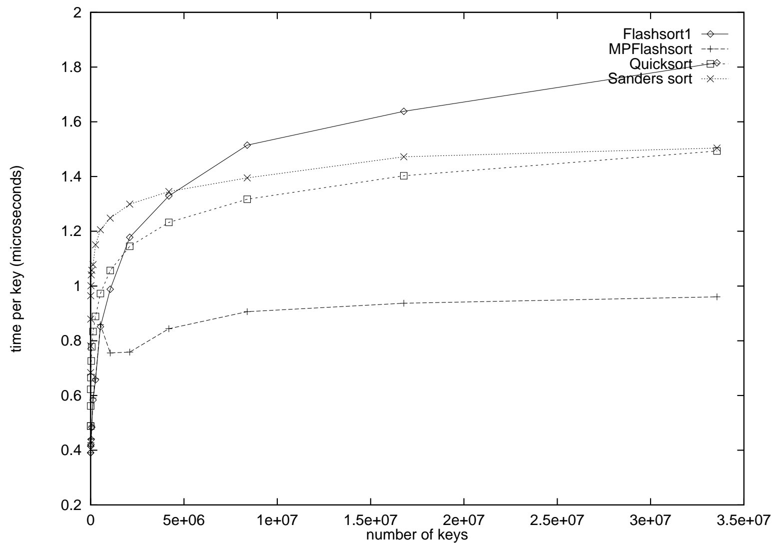

Fig. 6. Time per key to sort 8-byte (double-precision floating-point) keys using Flashsort1, MPFlashsort, MTQuicksort and 8-byte records using Sanders' heapsort on a Sun UltraSparc-II

Timings (s) on UltraSparc-II		
n (millions)	MPFlashsort	MSB radix
1.3	0.8885	0.5045
2.6	1.7855	1.0050
5.2	3.6550	2.0740
10.4	8.2570	5.1360
20.8	17.399	11.225
41.5	35.884	23.608
83.0	75.327	51.228

Fig. 7. Running times of MPFlashsort using single-precision floating-point keys and modified MSB radix sort using unsigned 32-bit integers on a Sun UltraSparc-II.

with $n = 83 \times 10^6$, MPFlashsort chooses about 1024.6 classes in the first pass. At this value of n, the running time of MPFlashsort is compared with MSB radix sort with $r = 10 = \log_2 1024$ in the first pass and with the subsequent passes left the same. By doing so, we ensure that for the selected values of n, the cost of memory accesses in both algorithms should be almost exactly the same. We can then conclude that any remaining difference in running time must be due to the different classify functions.

Figure 7 shows the running times for the modified MSB radix sort on uniform 32-bit unsigned integers and MPFlashsort on uniform single-precision floating-point numbers in the range $[0.0, 1.0)$ at the above values of n. We see that MSB radix sort outperforms MPFlashsort by about the same margin in this case, suggesting that its better performance is due to faster integer operations in the classify function.

6.2 Cache simulations

We also ran these algorithms on an L2 cache simulator for the Sun UltraSparc-II. Figure 8 compares MTQuicksort and Flashsort1, while Fig 9 compares the three faster algorithms. These figures show the number of L2 cache misses per key for the four algorithms on single-precision floating-point keys. We observe that:

—when the problem is small and fits in L2 cache, for $n \leq 64$K, the number of misses per key are almost constant for each algorithm, these are the compulsory misses.

—in Flashsort1 for $n \geq 128$K we see a rapid increase in the number of misses per key as n grows, appearing to level off at over 3 misses per key (virtually every access that could be a miss is a miss).

—in MTQuicksort the number of cache misses per key increases linearly with $\log n$ for $n \geq 128$K, reaching about 0.82 misses per key at 64M. Note that MTQuicksort makes $O((\log n)/B)$ misses per key on average, so this is not unexpected.

—in MPFlashsort we first see a rapid increase in misses per key going from 0.331 at $n = 128$K to 1.361 at $n = 512$K. At $n = 1$M the misses per key drops to 0.446, as the algorithm switches to two passes, and we see a very gradual increase reaching 0.504 at $n = 64$M.

—in MSB radix sort the number of cache misses per key increases gradually reaching a maximum of 1.1. Other than for small n, the cache utilisation for MSB radix

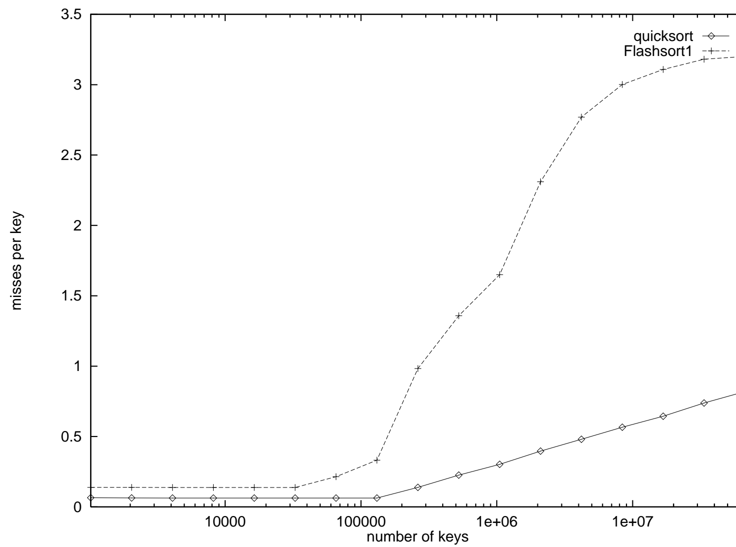

Fig. 8. L2 cache misses per key on single-precision floating-point keys: Flashsort1 vs MTQuicksort. Note that the x-axis is on a log scale.

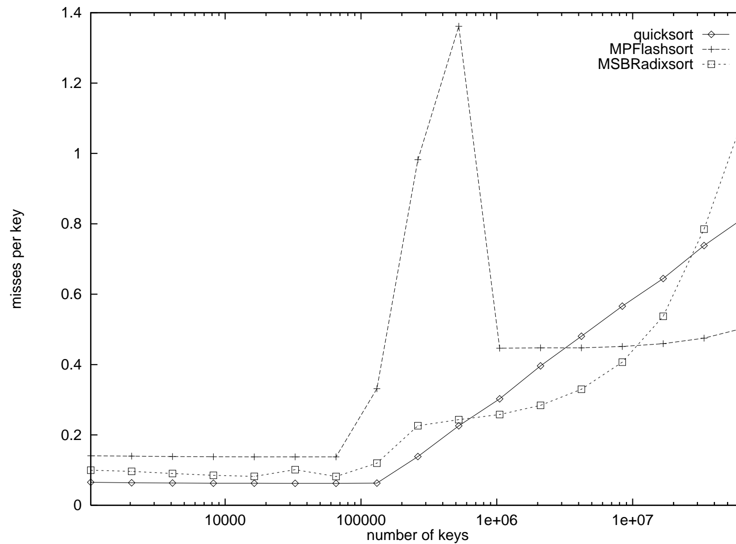

Fig. 9. L2 cache misses per key on single-precision floating-point keys: MPFlashsort vs MTQuicksort vs MSB radix sort. Note that the x-axis is on a log scale.

sort is much better than Flashsort1. The number of misses per key for MSB radix sort increases relatively rapidly for the larger values of n considered. One possible explanation for this is that the non-uniform distribution in the first pass of MSB radix sort results in sub-problems which may be larger than the size of the cache.

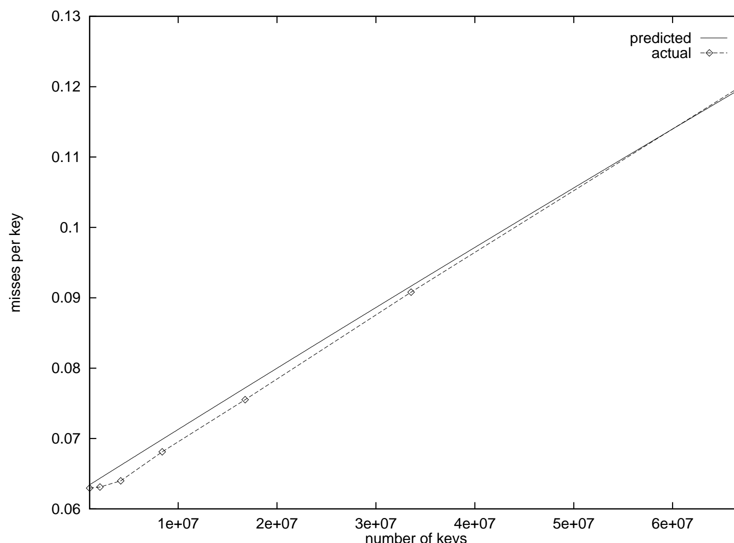

Fig. 10. Predicted L2 cache misses per key for values of $n = 1\text{M}, \ldots, 64\text{M}$ in increments of 128K using MPFlashsort and simulated L2 cache misses per key for $n = 1\text{M}$, 2M, 4M, 8M, 16M, 32M, 64M during the first 'in-place' permutation.

Fig 10 shows for the first permutation phase of MPFlashsort the cache misses per key predicted using Equation 19 for values of $n = 1\text{M}, \ldots, 64\text{M}$ in increments of 128K and the simulated cache misses per key for $n = 1\text{M}$, 2M, 4M, 8M, 16M, 32M, 64M and we see that the two lines follow each other very closely.

7. CONCLUSIONS

We have shown that Flashsort1 is fast when sorting a small number of random keys, but due to poor cache utilisation it starts to perform poorly when the data is larger than the cache size. We have shown that a variant of Flashsort1 which increases the number of passes when appropriate, called MPFlashsort, matches or outperforms Flashsort1 and MTQuicksort for all values of n, as it makes many fewer cache misses. We have analysed the cache miss rates of the Flashsort variants and can accurately predict the miss rates in the permute phases of these algorithms. We have also shown that the integer sorting algorithm MSB radix sort can be used very effectively on floating-point data. The algorithm is very fast due to fast integer operations and relatively good cache utilisation.

REFERENCES

AGGARWAL, A. AND VITTER, J. S. 1988. The I/O complexity of sorting and related problems. *Communications of the ACM 31*, 1116–1127.

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

HANDY, J. 1998. *The Cache Memory Handbook*. Academic Press.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach* (Second ed.). Morgan Kaufmann.

RUDIN, W. 1974. *Real & Complex Analysis*. McGraw-Hill, New York.

KNUTH, D. E. 1997. *The Art of Computer Programming. Volume 3: Sorting and Searching, 3rd ed.*. Addison-Wesley.

LADNER, R. E., FIX, J. D., AND LAMARCA, A. 1999. Cache performance analysis of traversals and random accesses. In *Proc. 10th ACM-SIAM SODA* (1999), pp. 613–622.

LAMARCA, A. AND LADNER, R. E. 1999. The influence of caches on the performance of sorting. *Journal of Algorithms 31*, 66–104. Preliminary version in *Proc. 8th ACM-SIAM SODA* (1997), pp. 370–379.

NEUBERT, K. D. 1998. The Flashsort1 algorithm. In *Dr Dobb's Journal* (February 1998), pp. 123–125. FORTRAN code listing, p. 131, *ibid.*

RAHMAN, N. AND RAMAN, R. 1999. Analysing cache effects in distribution sorting. Preliminary version in *Algorithm Engineering: Proc. 3rd Workshop on Algorithm Engineering* (1999), Number 1668 in LNCS, pp. 184–198.

RAHMAN, N. AND RAMAN, R. 2000. Analysing the Cache Behaviour of Non-uniform Distribution Sorting Algorithms. In *Proc. 8th Annual European Symposium on Algorithms* (2000), Number 1879 in LNCS, pp. 380–391.

SANDERS, P. 1999. Accessing multiple sequences through set-associative cache. In *Proc. 26th ICALP* (1999), Number 1644 in LNCS, pp. 655–664.

Appendix A

Using results in [Ladner et al. 1999] we now give results for the number of cache misses during the count phase of Flashsort1 and its variants. Our approach is to briefly describe the results in [Ladner et al. 1999] and then discuss how the results have been extended to deal with the count phase in Flashsort1 and its variants.

The analysis in [Ladner et al. 1999] assumes that the cache is single level and direct mapped, that count array locations are accessed uniformly and randomly, that $n = xCB$ for integer $x \geq 2$ and that $m = yB$ for integer $y \geq 1$. The expected number of cache misses per memory access predicted in that analysis is as follows:

$$T + R + I + I'.$$

Where T is the cache misses per memory access due to the traversal of the data array and is given by:

$$T = \frac{1}{2B}.$$

R is the expected number of cache misses per memory access due to memory blocks from the count array conflicting with each other and is given by:

$$R = \begin{cases} 0 & \text{if } m \leq CB \\ \frac{1}{2}(1 - \frac{C}{m/B}) & \text{if } m > CB. \end{cases}$$

I is the expected number of cache misses per memory access caused by the traversal periodically capturing each block in the cache and is given by:

$$I = \frac{m/B}{2CB}\left[1 - \left(1 - \frac{1}{m/B}\right)^{CB-(B-1)}\right]$$

when $m \leq CB$, otherwise it is approximately:

$$I \approx \frac{C}{2m}\left[1 - \left(1 - \frac{1}{C}\right)^{CB-(B-1)}\right].$$

I' is the expected number of cache misses per memory access caused by random count array accesses to a block that is mapped to the same one as being traversed and is given by:

$$I' = \begin{cases} \frac{B-1}{CB} & \text{if } m \leq CB \\ \frac{B-1}{2CB}\left(1 + \frac{C}{m/B}\right) & \text{if } m > CB. \end{cases}$$

There are n memory accesses to the data array and n memory accesses to the count array, so to express these results in terms of cache misses per key we have to multiply each of the terms above by a factor of 2.

Since the keys are uniform, the accesses to count blocks are uniform and random, so these results are applicable to Flashsort1 and its variants. However these results are only for the cache misses during the frequency counting step of the count phase (Step 2 of the count phase pseudo-code shown in Figure 1) and they do not take into account cache misses to initialise and prefix-sum the count array (Steps 1 and 3).

Assuming the cache is empty at the start of the count phase there will be m/B cache misses to initialise the count array. If the cache does not contain any count blocks before the prefix-sum phase, because the first C blocks were removed due to the traversal of data, then there will be an additional m/B cache misses. So there will be at most $(2m/B)/n$ cache misses per key to initialise and prefix-sum the count array.

We now give three cases which describe the cache misses per key during the count phase when distribution sorting uniform keys. In the first two cases we ignore the fact that n may not be a multiple of CB as the effect of this to the cache miss rate is not significant - only the values of I and I' would be inaccurate and their contribution is small. In all three cases we ignore the fact that m may not be a multiple of B.

Case 1: $n > CB$ and $m > CB$. In this case the data and count arrays are larger than the cache. The misses per key are as described by [Ladner et al. 1999] plus at most $2(m/B)/n$ misses per key for initialisation and prefix-summation of the count array. Thus re-writing $2(T + R + I + I') + (2m/B)/n$ we obtain:

$$\begin{aligned}\mu_{count} := {} & \frac{1}{B} + \frac{m - CB}{m} + \\ & \frac{C}{m}\left[1 - \left(1 - \frac{1}{C}\right)^{CB-B+1}\right] + \frac{(B-1)(m+CB)}{CBm} + \frac{2m/B}{n}. \end{aligned} \quad (24)$$

Case 2: $n > CB$ and $m \leq CB$. In this case the data array is larger than the cache but the count array fits in cache. The misses per key are as described by [Ladner et al. 1999] plus at most $2(m/B)/n$ misses per key for initialisation of the count array. Thus re-writing $2(T + R + I + I') + (2m/B)/n$ we obtain:

$$\mu_{count} := \frac{1}{B} + \frac{m/B}{CB}\left[1 - \left(1 - \frac{B}{m}\right)^{CB-B+1}\right] + \frac{2(B-1)}{CB} + \frac{2m/B}{n}. \quad (25)$$

Case 3: $n \leq CB$ *and* $m \leq CB$. In this case the data and count arrays fit in cache. Most of the misses here are as in Cases 1 and 2 above. There are $(m/B)/n$ misses per key for initialisation of the count array. The misses per key for the traversal acting alone, from T, is $1/B$. There are no conflict misses between count blocks. The misses per key due to random access to a count block mapped to the cache block being traversed by the data array, from I', is $\frac{2(B-1)}{CB}$.

For a cache block i which has a count block mapped to it, after the last data traversal access in i the count block is no longer in i so the count block must be reloaded at most once, either during the remaining data traversal accesses or during prefix-summation. The probability that cache block i has a data block mapped to it is $(n/B)/C$, so the combined expected number of cache misses per key is at most $\frac{n/B}{C}\frac{(m/B)}{n}$. So the overall number of cache misses per key is:

$$\mu_{count} := \frac{1}{B} + \frac{m/B}{n} + \frac{2(B-1)}{CB} + \frac{m/B}{CB}. \tag{26}$$

This case is applicable to the count phase in the final pass of MPFlashsort.

Appendix B

Figure 11 shows cache misses per key during the 'in-place' permutation phase for $n = 2^{24}$ and 100 random values of m between 25 and 8192 and for selected values of $m = 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2047, 2048, 2049, 4095, 4096$ and 4097. The misses per key were obtained:

—from 20 cache simulations, labelled *Sim*,

—predicted from actual values of ν taken at the start of permutation in the full simulations above, labelled *Act*,

—using Equation 19, labelled *Pred*.

For a value of m where there is greater than 5% variation between these values m is prefixed with a superscript of $a, \ldots, c$, the superscripts denote variations:

a) between simulated and predicted alone,

b) between simulated and predicted and between actual and predicted,

c) between all three values.

Some observations on the results are:

—For most random values of m the variation between the simulated and predicted misses per key is at-most 5%.

—For most values of m where there is greater than 5% variation the superscript is b, indicating that simulated misses per key and misses per key predicted from actual ν vary together.

—For most random values of m where there is greater than 5% variation we note that $\gcd(\tau, m)$ is small but $\gcd(\tau, m-1)$ or $\gcd(\tau, m+1)$ is large. These are examples of heuristic 1 not being effective, as discussed in Section 4.4.2.

Note that greater than 5% variation between other combinations are not observed, for instance there was never a more than 5% variation between the actual and predicted values alone.

Figures 12, 13 show for $n = 1$M, 4M and 16M all values of m between 32 and 8192 where there was greater than 5% variation between the actual value of ν at the start of permutation, averaged over 20 experiments, and ν predicted using Equation 17. Figure 14 shows these results at $n = 16$M where the actual value of ν was averaged over 195 experiments. Values of m where $\gcd(\tau, m)$ is not equal to τ, $\tau/2$, $\tau/4$, or m are prefixed with a *. Some observations on the results are:

—We note that values which are prefixed with * have $\gcd(\tau, m-1) = \tau$ or $\gcd(\tau, m+1)$. These are further examples of heuristic 1 not being effective.

—As m increases fewer and fewer values are listed in the table, this shows that our equations will be accurate for most values of $m > 32$ while $n/m \gg B$.

We can report that at $n = 16$M for 6676 values of m between 32 and 8192 the actual value of ν averaged over 195 experiments was higher than the value predicted using Equation 17.

Figure 15 shows for $n = 16$M at values of m selected using heuristic 2, actual ν at the start of permutation, averaged over 20 experiments, and ν predicted using Equation 17. We note that at all these values of m the actual value of ν was higher than the predicted value.

m	Sim	Act	Pred	m	Sim	Act	Pred	m	Sim	Act	Pred
49	0.065	0.065	0.066	[b]128	0.301	0.290	0.072	[b]129	0.065	0.065	0.072
210	0.076	0.077	0.077	[b]240	0.094	0.093	0.080	[b]253	0.076	0.076	0.080
[b]256	0.297	0.291	0.081	[b]384	0.287	0.293	0.090	[2]472	0.094	0.090	0.096
[b]512	0.300	0.292	0.098	554	0.103	0.102	0.101	615	0.106	0.104	0.105
[b]640	0.290	0.294	0.107	662	0.109	0.109	0.109	[b]768	0.292	0.297	0.116
781	0.116	0.113	0.117	789	0.121	0.118	0.117	[b]896	0.293	0.298	0.124
930	0.125	0.124	0.127	[b]1024	0.295	0.298	0.133	1038	0.133	0.130	0.134
1105	0.137	0.136	0.138	[a]1120	0.147	0.145	0.139	1123	0.140	0.135	0.139
1137	0.140	0.140	0.140	[b]1152	0.296	0.302	0.141	1178	0.147	0.142	0.143
1267	0.148	0.145	0.149	[b]1280	0.298	0.301	0.150	1318	0.151	0.150	0.152
[b]1408	0.299	0.306	0.158	[b]1536	0.301	0.306	0.166	[b]1664	0.302	0.309	0.174
[b]1792	0.304	0.311	0.182	1888	0.192	0.189	0.188	1910	0.188	0.185	0.189
[b]1920	0.306	0.312	0.190	2042	0.201	0.195	0.198	[b]2047	0.245	0.239	0.198
[b]2048	0.320	0.313	0.198	[b]2049	0.241	0.240	0.198	2060	0.198	0.192	0.199
2164	0.204	0.198	0.205	[b]2239	0.228	0.221	0.210	2293	0.212	0.206	0.213
2364	0.221	0.216	0.217	2420	0.220	0.215	0.220	2456	0.223	0.218	0.222
2500	0.230	0.222	0.225	2545	0.227	0.222	0.228	2582	0.231	0.225	0.230
2596	0.233	0.227	0.231	2618	0.234	0.227	0.232	2629	0.236	0.230	0.233
2693	0.246	0.241	0.236	2842	0.246	0.242	0.245	2885	0.251	0.245	0.247
2937	0.256	0.250	0.250	[c]2945	0.304	0.286	0.251	3266	0.280	0.269	0.269
3368	0.279	0.273	0.275	3658	0.295	0.287	0.291	3662	0.295	0.286	0.291
3742	0.298	0.289	0.295	3768	0.301	0.292	0.296	[c]3841	0.345	0.328	0.300
4040	0.315	0.307	0.311	[b]4095	0.355	0.352	0.314	[b]4096	0.354	0.355	0.314
[b]4097	0.353	0.338	0.314	4143	0.320	0.310	0.316	4148	0.321	0.312	0.317
4170	0.322	0.312	0.318	4200	0.322	0.313	0.319	4312	0.329	0.322	0.325
4387	0.333	0.324	0.329	4677	0.350	0.340	0.344	4845	0.357	0.346	0.352
4896	0.359	0.350	0.355	4977	0.364	0.358	0.359	5015	0.365	0.357	0.360
5115	0.383	0.374	0.365	5344	0.380	0.372	0.376	5447	0.387	0.379	0.381
5470	0.387	0.378	0.382	5499	0.401	0.394	0.384	5562	0.393	0.385	0.387
[a]5634	0.415	0.405	0.390	5691	0.401	0.392	0.393	5700	0.399	0.389	0.393
5778	0.403	0.394	0.397	5809	0.403	0.396	0.398	5811	0.404	0.395	0.398
5812	0.404	0.396	0.398	5929	0.409	0.398	0.404	5935	0.410	0.401	0.404
6119	0.418	0.408	0.412	6243	0.423	0.415	0.418	6249	0.424	0.414	0.418
6260	0.427	0.420	0.419	6503	0.435	0.426	0.430	6515	0.439	0.431	0.430
6855	0.451	0.441	0.445	6911	0.464	0.454	0.447	6990	0.457	0.449	0.451
7013	0.457	0.449	0.452	7039	0.469	0.458	0.453	7106	0.463	0.453	0.456
7120	0.462	0.453	0.456	7262	0.468	0.459	0.462	7273	0.470	0.461	0.463
7274	0.469	0.459	0.463	7278	0.470	0.462	0.463	7279	0.472	0.462	0.463
7324	0.471	0.461	0.465	7356	0.473	0.463	0.466	7436	0.479	0.471	0.469
7584	0.482	0.473	0.475	7593	0.482	0.472	0.476	7839	0.492	0.481	0.486
8073	0.506	0.499	0.495								

Fig. 11. $Misses/n$ during inplace-permutation for $n = 2^{24}$ using random and selected values of m, the $misses/n$ values are: simulated; predicted using actual ν at start of permutation; predicted using Equation 19.

m	Actual	Predict	m	Actual	Predict	m	Actual	Predict
40	37.900	39.903	56	53.150	55.809	96	90.850	95.440

Fig. 12. For $n = 2^{20}$ all values of m between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 17 at start of permutation.

m	Actual	Predict	m	Actual	Predict	m	Actual	Predict
32	27.600	31.938	48	44.750	47.860	64	55.650	63.751
80	74.800	79.611	96	83.300	95.440	112	104.650	111.238
128	111.550	127.005	144	134.950	142.742	160	138.650	158.448
176	164.350	174.123	192	166.700	189.767	208	194.900	205.382
224	196.250	220.965	240	225.000	236.518	256	223.300	252.041
288	251.800	282.996	320	277.950	313.831	352	307.800	344.545
384	335.100	375.139	416	363.800	405.614	448	392.750	435.970
480	417.450	466.208	512	447.850	496.328	544	476.750	526.331
576	502.200	556.216	608	532.750	585.985	640	558.200	615.639
672	587.300	645.176	704	615.850	674.598	736	644.000	703.906
768	670.650	733.099	800	700.000	762.179	832	729.100	791.145
864	756.900	819.998	896	782.900	848.739	928	809.200	877.367
960	835.650	905.884	992	863.650	934.290	1024	895.600	962.585
1056	922.550	990.770	1088	953.850	1018.845	1120	982.150	1046.811
1152	1006.550	1074.667	1184	1035.450	1102.415	1216	1061.700	1130.055
1248	1092.900	1157.587	1280	1114.600	1185.011	1312	1143.950	1212.329
1344	1175.800	1239.540	1376	1203.050	1266.645	1408	1228.550	1293.644
1440	1255.200	1320.538						

Fig. 13. For $n = 2^{22}$ all values of m between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 17 at start of permutation.

m	Actual	Predict	m	Actual	Predict	m	Actual	Predict
32	30.103	31.940	64	56.256	63.755	96	90.200	95.445
128	98.764	127.013	160	150.062	158.457	192	168.503	189.779
224	210.359	220.979	256	197.487	252.056	320	280.862	313.849
384	295.564	375.161	448	392.764	435.996	512	395.077	496.358
576	504.195	556.249	640	493.672	615.675	704	616.995	674.638
768	590.964	733.142	832	729.518	791.191	896	689.979	848.788
960	841.841	905.936	1024	789.215	962.641	1088	953.882	1018.903
1152	885.713	1074.728	1216	1066.462	1130.119	1280	986.990	1185.078
1344	1177.482	1239.610	1408	1082.513	1293.717	1536	1183.000	1400.671
1664	1281.949	1505.967	1792	1378.677	1609.631	*1793	1529.574	1610.434
1920	1477.467	1711.687	*1921	1622.308	1712.478	*2047	1724.036	1811.382
2048	1577.923	1812.161	*2049	1713.836	1812.940	*2175	1816.959	1910.311
2176	1675.385	1911.078	*2177	1805.559	1911.844	*2303	1907.713	2007.705
2304	1772.359	2008.460	*2305	1894.251	2009.215	*2431	1997.749	2103.590
2432	1872.549	2104.333	*2433	1982.041	2105.076	*2559	2088.015	2197.987
2560	1972.785	2198.719	*2561	2067.682	2199.451	*2687	2175.426	2290.922
2688	2066.862	2291.642	*2689	2157.744	2292.362	*2815	2263.800	2382.415
2816	2163.441	2383.124	*2817	2243.610	2383.833	*2943	2354.072	2472.490
2944	2259.780	2473.188	*2945	2330.426	2473.886	3072	2356.205	2561.855
*3073	2419.210	2562.542	3200	2449.185	2649.148	*3201	2503.903	2649.824
3328	2544.554	2735.087	*3329	2587.631	2735.753	3456	2633.533	2819.694
*3457	2672.313	2820.349	3584	2726.615	2902.989	*3585	2754.549	2903.634
3712	2814.467	2984.992	*3713	2838.764	2985.628	3840	2901.872	3065.724
*3841	2919.005	3066.350	3968	2988.113	3145.204			

Fig. 14. For $n = 2^{24}$ all values of m between 32 and 8192 where there was greater than 5% variation between actual ν and ν predicted using Equation 17 at start of permutation.

m	Actual	Predict	m	Actual	Predict	m	Actual	Predict
46.997	46.940	46.865	136.196	136.210	135.078	144.802	144.060	143.538
163.378	162.760	161.769	176.892	175.490	175.006	207.108	205.860	204.524
335.108	330.130	328.366	591.108	573.350	570.319	1103.108	1037.320	1032.121
2127.108	1882.650	1873.477	4175.108	3290.240	3271.204	8271.108	5232.550	5207.478

Fig. 15. For $n = 2^{24}$ at values of m selected using the ϕ heuristic, actual ν and ν predicted using Equation 17 at start of permutation.