Adapting radix sort to the memory hierarchy

Naila Rahman
and
Rajeev Raman
King's College London

We demonstrate the importance of reducing misses in the *translation-lookaside buffer (TLB)* for obtaining good performance on modern computer architectures. We focus on least-significant-bit first (LSB) radix sort, standard implementations of which make many TLB misses. We give three techniques which simultaneously reduce cache and TLB misses for LSB radix sort: *reducing working set size*, *explicit block transfer* and *pre-sorting*. We note that:

- All the techniques above yield algorithms whose implementations outperform optimised cache-tuned implementations of LSB radix sort and comparison-based sorting algorithms. The fastest running times are obtained by the pre-sorting approach and these are over twice as fast as optimised cache-tuned implementations of LSB radix sort and quicksort. Even the simplest optimisation, using the TLB size to guide the choice of radix in standard implementations of LSB radix sort, gives good improvements over cache-tuned algorithms.
- One of the pre-sorting algorithms and explicit block transfer make few cache and TLB misses in the worst case. This is not true of standard implementations of LSB radix sort.

We also apply these techniques to the problem of permuting an array of integers, and obtain gains of over 30% relative to the naive algorithm by using explicit block transfer.

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*; D.1.0 [**Software**]: Programming Techniques—*General*; B.3.2 [**Hardware**]: Memory Structures—*Design Styles*

General Terms: Efficient sorting algorithms, Locality of reference, Radix sort,Memory hierarchy, External-memory algorithms

Additional Key Words and Phrases: Cache, Translation-lookaside buffer (TLB)

1. INTRODUCTION

Most algorithms are analysed on the random-access machine (RAM) model of computation [Aho et al. 1974], using some variety of the unit-cost criterion. This postulates 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, and CPUs with clock rates exceeding 1GHz are available in the mass market. Unfortunately, the speeds of main memory have not increased as rapidly: today's main memory typically has a latency of

This work was supported in part by grant GR/L92150 from the UK Engineering and Physical Sciences Research Council (EPSRC).
Address: Department of Computer Science, King's College London, Strand, London WC2R 2LS, U. K. e-mail: {naila, raman}@dcs.kcl.ac.uk

about 60ns. This implies that the cost of accessing main memory can be 60 times greater than the cost of performing an operation on two operands which are in the CPU's registers.

In order to overcome this difference in speeds, modern computers have multiple levels of *cache* between CPU and 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 main 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 requires a main memory access, and is therefore very expensive. To amortise the cost of a main memory access in case of a cache miss, an entire *block* of consecutive main memory locations which contains the location accessed is brought into cache on a miss. Thus, a program that exhibits good *locality*, i.e. one that accesses memory locations near those which it accessed previously, will incur fewer cache misses and will consequently run faster. Much recent work has focussed on minimising cache misses in various algorithms [LaMarca and Ladner 1999; Rahman and Raman 1999; Sen and Chatterjee 2000; Mehlhorn and Sanders 2000; Frigo et al. 1999]. Asymptotically, one can minimise cache misses by simulating external-memory algorithms [Sen and Chatterjee 2000], for which there is a large literature [Vitter 2000].

There is an important related optimisation which can contribute as much or more to performance, namely minimising misses in the *translation-lookaside buffer (TLB)*. Some papers from the early 90's (see e.g. [Moret and Shapiro 1994]) have noted the importance of minimising TLB misses when implementing algorithms, but there has been no systematic study of this optimisation, even though TLB misses are often at least as expensive as cache misses.

The TLB is used to support *virtual memory* in multi-processing operating systems [Hennessy and Patterson 1996]. Virtual memory means that the memory addresses accessed by a process refer to its own unique logical address space. This logical address space contains as many locations as can be addressed on the underlying architecture, which far exceeds the number of physical main memory locations in a typical system. Furthermore, there may be several active processes in a system, each with its own logical address space. To allow this, most operating systems partition main memory and the logical address space of each process into contiguous fixed-size *pages*, and stores only some pages from the logical address space of each active process in main memory at a time. Owing to its myriad benefits, virtual memory is considered to be "essential to current computer systems" [Hennessy and Patterson 1996]. The disadvantage of virtual memory is that every time a process accesses a memory location, the reference to the corresponding logical page must be *translated* to a physical page reference. This is done by looking up the *page table*, a data structure in main memory and would lead to unacceptable slowdown[1].

The TLB is used to speed up address translation. It is a fast associative memory which holds the translations of recently-accessed logical pages. If a memory access results in a TLB hit, there is no delay, but a TLB miss can be significantly more

[1] In case the logical page is not present in main memory at all, it is brought in from disk. The time for this is not counted in the CPU times reported.

expensive than a cache miss; hence locality at the page level is also very desirable. In most computer systems, a memory access can result in a TLB miss alone, a cache miss alone, neither, or both: algorithms which make few cache misses can nevertheless have poor performance if they make many TLB misses. We argue later that due to the different characteristics of a cache miss and a TLB miss, these should be counted separately.

The sizes of cache and TLB are limited by several factors including cost and speed [Handy 1998]. Cache capacities are typically 512KB to 2MB, which is considerably smaller than the size of main memory. The size of TLBs are also very limited, with 64 to 128 entries being typical. This means that many algorithms which have good performance in the RAM model perform poorly in practice. In this paper, we focus on one such: sorting records with integer keys using *least significant bit first (LSB) radix sort*.

Radix sorting, applied to integers, consists in viewing w-bit integer keys as $\lceil w/r \rceil$ consecutive r-bit *digits*. The records are sorted in $\lceil w/r \rceil$ passes: in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$ we sort the records according to the i-th least significant digit. There are a number of ways of implementing a single pass in $O(n + 2^r)$ time, the best one in practice being *counting sort* [Cormen et al. 1990]. We follow current usage and refer to radix sort plus counting sort as 'LSB radix sort'.

LSB radix sort has an overall running time of $O(\lceil w/r \rceil (n+2^r))$ for w-bit keys. For medium-to-large input sizes, LSB radix sort may be considered to be a linear-time algorithm for practical purposes. However, experimental work has shown that LSB radix sort, even when tuned for cache performance, fails to outperform good implementations of $O(n \log n)$ comparison-based algorithms on modern architectures [LaMarca and Ladner 1999]. We show that cache-tuned LSB radix sort still has poor TLB performance. On the other hand, many comparison-based algorithms intrinsically have good TLB performance. Hence it is necessary to use models which incorporate the TLB if one is to get good implementations of LSB radix sort. In this paper, we define a natural model for analysing TLB performance, and give three approaches which reduce TLB misses for radix sort: *reducing working set size*, *explicit block transfer* and *pre-sorting*. We now outline these techniques.

The *working set* of a program is the the set of pages it accesses at a particular time. If the program makes random accesses to these pages, and the working set size is much larger than the size of the TLB, then the number of TLB misses will be large (in this paper we use the term *random access* to denote accesses which do not exhibit locality). Hence, reducing the working set size to about the capacity of the TLB can greatly reduce the number of TLB misses. In the case of LSB radix sort, this is accomplished by reducing the radix far below the values suggested in textbooks [Cormen et al. 1990, p. 179] or those obtained from cache analyses [LaMarca and Ladner 1999].

Our second technique, explicit block transfer, enforces locality by ensuring that all random accesses are made only for the purpose of copying blocks of memory locations. This technique uses an observation of [Sen and Chatterjee 2000] that given any algorithm A designed for the external-memory model of [Aggarwal and Vitter 1988], one can design a main-memory based emulation of A which incurs $O(1)$ cache misses for every I/O operation performed by A. We note that essentially the same emulation incurs $O(1)$ cache and $O(1)$ TLB misses for each I/O operation

performed by A. In the external-memory model, good algorithms perform as few I/O operations as possible—this translates into few cache and TLB misses for the emulation. Our new algorithm *explicit block transfer radix sort (EBT radix sort)* essentially emulates a natural external-memory analogue of counting sort in each pass. On a more general note, the emulation is a special case of the well-known trick of copying data to reducing cache conflict misses [Gannon and Jalby 1987]. Our result suggests that copying data can also help improve TLB performance.

In our final technique, pre-sorting, prior to each pass, the array containing the keys is 'conditioned' by sorting the records within relatively small segments. The pre-sorting is then used in two different ways, giving two algorithms. In the first algorithm, *pre-sorting LSB radix sort (PLSB radix sort)*, we note that the pre-sorting brings together keys with equal values which can then be moved in a group, imparting locality to the 'global' sort. In the second, *extended-radix PLSB radix sort (EPLSB radix sort)*, we use the increased locality afforded by pre-sorting to increase the radix size—hence reducing the number of passes—whilst continuing to incur an acceptable number of cache and TLB misses.

From the theoretical viewpoint, these approaches have different characteristics. As we note here, LSB radix sort has very poor cache performance on worst-case problem instances, and reducing the radix size does not alleviate this problem. However, LSB radix sort can make few TLB misses in the worst case. These characteristics are shared by EPLSB radix sort. On the other hand, both EBT radix sort and PLSB radix sort make an asymptotically optimal number of cache misses, and few TLB misses, in the worst case. None of the algorithms simultaneously achieves asymptotic optimality for cache and TLB misses in general.

We have tested implementations of these four approaches on a Sun UltraSparc-II architecture. These implementations generally vary slightly from the algorithm descriptions which means they may sacrifice worst-case performance (excepting PLSB radix sort). All implementations show good speed-ups over highly optimised implementations of cache-tuned LSB radix sort and cache-tuned comparison-based algorithms. In our experiments EPLSB radix sort performed the best, running over twice as fast as cache-tuned LSB radix sort or cache-tuned quicksort.

2. MODELS USED

The basic model we use is the random-access machine, consisting of a CPU and main memory, augmented with a cache and TLB.

2.1 Cache Model

For the cache, we view main memory as being partitioned into equal-sized *blocks* of locations, each consisting of B memory words. Blocks are *aligned*, that is, they begin at addresses which are congruent to 0 modulo B. The model assumes a single cache consisting of S *sets* each consisting of a *lines*. Each line can hold a memory block, and memory block i can be stored in any of the lines in set (i mod S). We denote by $C = aS$ the capacity of the cache, and a is called the *associativity* of the cache. When $a = 1$, the cache is said to be *direct-mapped*, and when $a > 1$, we say the cache is a-*way* associative. If the program accesses a location in block i, and block i is not in the cache, then one of the blocks in the set (i mod S) is *evicted* or copied back to main memory, and block i is copied into the set in its

place. We assume that blocks are evicted from a set on a *least recently used (LRU)* basis. Note that for a direct-mapped cache, there is only one block in each set, and the replacement policy becomes trivial.

The above model simplifies the architecture of real machines considerably. For instance, our experiments are performed on a Sun UltraSparc-II, which supports two levels of cache, as do most other current architectures. However, in the Ultra-II, the L1 (faster, smaller) cache is *write-through*, i.e., in case of a cache hit during a write, the value of the memory location is simultaneously updated in the L1 cache and in the L2 (slower, larger) cache. Since most cache misses in the algorithms we consider occur during writes to memory locations, this simplification is reasonable in our case.

2.2 TLB model

For the TLB, we consider main memory as being partitioned into equal-sized and aligned *pages* of P memory words each. A TLB holds address translation information for at most T pages. If the program accesses a memory location which belongs to (logical) page i, and the TLB holds the translation for page i, the contents of the TLB do not change. If the TLB does not hold the translation for page i, the translation for page i is brought into the TLB, and the translation for some other page is removed, again on a least-recently-used (LRU) basis.

We assume that TLB misses and cache misses happen independently, in that a memory access may result in a cache miss, a TLB miss, neither, or both. This is because caches are usually *physically tagged*, i.e., the values stored in cache are stored according their physical, and not their logical, memory addresses. Hence, when a program accesses a memory location using its logical address, the address first has to be translated to a physical address before the cache can be checked. Furthermore, a memory access which results in both a cache miss and a TLB miss pays the cache miss penalty plus the TLB miss penalty (there is no saving, or loss, if both kinds of misses occur simultaneously).

Again, this model is a simplification of real TLBs. In principle a TLB miss can be much more involved than a cache miss, requiring software intervention. Hence some architectures may provide hardware support for handling TLB misses. For example, on the Sun Ultra-II, a TLB miss can cost any of: (i) a L2 cache *hit* (2-3 CPU clock cycles) (ii) a memory access (35-40 cycles) or (iii) a trap to a software miss handler (hundreds of cycles) [Sun Microsystem 1997, Chapter 6]. Another simplification is that TLBs almost always implement an approximation to LRU replacement, rather than a true LRU policy.

2.3 Parameter values and Performance measures

We make a number of assumptions regarding the parameter values to simplify the analyses. These assumptions normally hold in practice, as explained below (see [Hennessy and Patterson 1996] for further details). The first assumption is that B, C, P and T are all powers of two. The remaining assumptions are:

(1) $B \leq P$ — Data is transferred from secondary memory as a page, and since secondary memory has much higher latency than main memory a page is much larger than a block. Indeed we should assume that $B \ll P$.

(2) $T \leq C$ — A TLB is fully-associative. This makes the hardware realisation of a TLB much more complex than a cache with limited associativity. Furthermore, a TLB hit must be serviced very quickly. These two factors imply that a TLB must be much smaller than a cache, and we should assume that $T \ll C$.

(3) $BC \leq PT$ — $PT < BC$ implies that some portion of cache will always lead to a TLB miss, thus effectively wasting part of cache. However, $PT > BC$ makes sense and does occur in practice (see below).

(4) $T \leq P$, $B \leq C$ — These are technical assumptions, which are similar to the *tall cache* assumption of [Frigo et al. 1999] and appear to hold in practice (strictly speaking, we make a 'tall cache' and 'short TLB' assumption).

Our experiments are performed on the Sun UltraSparc-II architecture, details of which can be found in [Sun Microsystem 1997]. The UltraSparc-II has a word size of 4 bytes, and a block size of 64 bytes, giving $B = 16$. Its L2 cache, which is direct-mapped, holds $C = 8192$ blocks and its TLB holds $T = 64$ entries. On the system we use for experiments, the page size is 8192 bytes, giving $P = 2048$ and also (coincidentally) giving $BC = PT$. The relationship between BC and PT can vary even across systems using the same processor. For example the UltraSparc-II processor supports different page sizes—the largest being 4MB—giving a maximum PT of 256MB, but the maximum supported cache size is 16MB.

The main performance measures are the number of instructions, the number of TLB misses and the number of cache misses. These are all counted separately. We count TLB and cache misses separately as these have different characteristics. The advantage of counting instructions and misses separately is that it allows us to use the coarse O-notation for simple operations and also to analyse the number of cache or TLB misses more carefully, if necessary. In this respect we are faithful to both the external-memory approach of [Aggarwal and Vitter 1988] and the cache-analysis approach of [LaMarca and Ladner 1999]. We make extensive use of the following, which are restatements of a result of [Sleator and Tarjan 1985]:

THEOREM 1. *(i) For any given sequence of memory accesses, a TLB which has size τ and uses LRU replacement makes at most $\tau/(\tau - \tau^* + 1)$ times as many misses as a (hypothetical) TLB of size $\tau^* \leq \tau$ which knows the memory access sequence and follows the optimal offline replacement policy for that sequence. This assumes that both the LRU and the optimal TLBs are initially empty.*
(ii) For any given sequence of memory accesses, an α-way associative cache which uses LRU replacement makes at most $\alpha/(\alpha - \alpha^* + 1)$ times as many misses as a (hypothetical) α^*-way associative cache with the same number of sets, which knows the memory access sequence and follows the optimal offline replacement policy for

that sequence. This assumes that $\alpha^* \leq \alpha$ and that both the LRU and the optimal caches are initially empty.

3. LSB RADIX SORT

In this and the following sections, we will use the following default terminology. The term *key* will refer to a single digit, and the term *record* will denote the integer which was input to the sorting algorithm plus any associated information.

In radix sorting we view w-bit integer keys as $\lceil w/r \rceil$ consecutive r-bit *digits*. The records are sorted in $\lceil w/r \rceil$ passes: in the i-th pass, for $i = 1, \ldots, \lceil w/r \rceil$ we sort the records according to the i-th least significant digit.

We now review one pass of LSB radix sort. The array containing the records at the start of the pass is called the *source* array. The records into which the keys are sorted is called the *destination* array. In addition, one or more *count* arrays are used to keep auxiliary information. Counting sort comprises of a *count* phase, a *prefix sum* phase, and a *permute phase* [Cormen et al. 1990, pp. 175–177]. During the count phase the algorithm counts the numbers of keys with the same *class*, where the class of a key is the value of the digit being sorted on in this pass. During the prefix sum phase the starting locations in the destination array are marked off for keys with the same class. During the permute phase, each record is moved to its new location, using the count array to determine the new location for the record. We will generally focus on one pass of radix sorting.

3.1 Cache misses for LSB radix sort

During the count phase, LSB radix sort makes the following memory accesses:

(1) A sequential read access to the source array, to move to the next key to count.
(2) One or two random read/write accesses to the count array to increment the count values, depending on whether or not Friend's [Friend 1956] improvement is implemented (this consists in simultaneously accumulating counts for two passes in one count phase).

During the permute phase, the algorithm makes the following accesses:

(1) A sequential read access to the source array, to find the next record to move.
(2) A random read/write access to the count array, to find where to move the next record and to increment the count array location just read.
(3) A random write to one of 2^r *active locations* in the destination array, to actually move the record.

An average-case analysis of the count phase in LSB radix sort was provided by [LaMarca and Ladner 1999; Ladner et al. 1999] for a direct-mapped cache. Although an approximate average-case analysis of the permute phase was given in [LaMarca and Ladner 1999], it ignored misses caused by conflicts between active locations, which are essential to Proposition 1 below. In [Rahman and Raman 1999; Mehlhorn and Sanders 2000] algorithms with memory access patterns which are very similar to LSB radix sort are studied 'on average' [2]. We now note that in the worst case

[2] [Rahman and Raman 1999] gives an average-case approximate analysis based on random input and [Mehlhorn and Sanders 2000] gives a expected-case analysis based on randomisation by the

Timings(sec)						
n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}
pathological (r=11)	0.44	0.94	1.92	4.37	7.70	15.40
random (r=11)	0.31	0.65	1.33	3.01	5.47	10.89

Fig. 1. Running times for one pass of LSB radix sort using 11-bit radix for random 32-bit unsigned integers and the pathological input.

LSB radix sort makes many misses.

PROPOSITION 1. *For any fixed cache associativity $a \geq 1$, one pass of LSB radix sort with radix r makes $\Omega(n)$ cache misses in the worst case whenever $2^r > a$.*

PROOF. For convenience let n be a power of two. Consider just the random write accesses (step (3) above) in the permute phase of LSB radix sort and let the input consist of the sequence $0, 1, \ldots, 2^r - 1$ repeated $n/2^r$ times. With this input, the 2^r active blocks will be mapped into a total of $\max\{1, BS/(n/2^r)\}$ sets, giving $\min\{2^r, n/(BS)\}$ active locations mapped to each set. Provided that $n > aBS = BC$, the number of active blocks mapped to each set will exceed a. Owing to the round-robin nature of accesses to the active data blocks mapped to a set, all accesses to active locations will be misses, even ignoring other conflicts. This gives a minimum of n misses for a single permute phase in the worst case.

Note that the count phase incurs $O(n/B)$ misses in the worst case whenever $a \geq 2$ and $2^r \leq BC/2$ (we require that $2^r \leq BC/4$ if Friend's improvement is used). □

The construction of Proposition 1 can easily be generalised to get inputs which have bad performance on several successive passes:

PROPOSITION 2. *For any fixed cache associativity $a \geq 1$, in the worst case, LSB radix sort with radix r makes $\Omega(n)$ cache misses in each of $\lfloor w/r \rfloor$ passes, provided $2^r > a$, when sorting w-bit keys.*

PROPOSITION 3. *For any fixed cache associativity $a \geq 1$ and n a power of 2, LSB radix sort with radix r makes $\Omega(n)$ cache misses in each of $\lfloor (\log n)/r \rfloor$ passes on the input $0, 1, \ldots, n-1$, provided $2^r > a$.*

From the asymptotic viewpoint, the results from [Rahman and Raman 1999; Mehlhorn and Sanders 2000] suggest that 'on average' if we have $2^r = O(C/B)$, then each pass of radix sort makes an optimal $O(n/B)$ expected misses even on a direct-mapped cache. For a cache with associativity a [Mehlhorn and Sanders 2000] suggests that the expected number of misses is $O(n/B)$ for $2^r = O(C/B^{1/a})$. The worst-case input increases this to $\Theta(n)$ misses per pass even when r is very small, i.e. when $2^r > a$.

This difference is evident in practice too: Fig 1 compares the running times for one pass of LSB radix sort using a 11-bit radix when the input data is the worst-case described above and random data. It shows that the algorithm is up to 45% slower on worst-case data than on random input for radix 11.

algorithm and an oblivious adversary.

3.2 TLB misses in LSB radix sort

We now give calculate the number of TLB misses in the count phase and permute phase of LSB radix sort in the worst case:

PROPOSITION 4. *The number of TLB misses in the count phase of one pass of LSB radix sort with radix r when sorting n keys is:*

$$\leq \quad (\lceil n/P \rceil + W)\left(\frac{T}{T-W}\right), \quad \text{if } W \leq T-1 \tag{1}$$

$$\geq \quad \lceil n/P \rceil + n, \qquad \text{if } W > T-1. \tag{2}$$

in the worst case, where $W = \lceil 2^r/P \rceil$.

PROOF. During the count phase, the algorithm accesses the following *working* set of pages: (i) one active page in the source array and (ii) $W = \lceil 2^r/P \rceil$ count array pages. Thus, if $W + 1 \leq T$, an optimal algorithm with a TLB of size $W + 1$ will make no more than $\lceil n/P \rceil + W$ misses (to bring in the count array pages as well as to bring in the successive active source array pages.) Applying Theorem 1 proves (1).

If $W + 1 > T$ and $T = 1$ then clearly all $2n$ memory accesses cause misses whatever replacement policy is used. If $T > 1$ then the active source array page is always in the TLB, but at least one count array page is always not in the TLB, and in the worst case, the count array page which is not in the TLB will always be accessed (a round-robin access pattern suffices for this). Thus all n accesses to count array pages are misses, and we add the compulsory misses for source array pages to get (2). □

The following proposition is proved analogously:

PROPOSITION 5. *The number of TLB misses in the permute phase of one pass of LSB radix sort with radix r when sorting n keys is:*

$$\leq \quad (\lceil n/P \rceil + W)\left(\frac{T}{T-W}\right), \quad \text{if } W \leq T-1 \tag{3}$$

$$\geq \quad \lceil n/P \rceil + n, \qquad \text{if } W > T-1. \tag{4}$$

in the worst case, where $W = \lceil 2^r/P \rceil + \min\{2^r, \lceil n/P \rceil\}$.

This shows that provided $T \ll P$ (which is usually the case) there is a sharp threshold for TLB misses, going from $O(nT/P)$ when the working set fits into the TLB to $\Omega(n)$ when it does not. Hence the radix should be chosen small enough that the former case holds.

3.3 Implementing LSB radix sort

3.3.1 *Reducing working set size.* From the above discussion, we can reduce TLB misses by reducing r to the point where the working set is sufficiently small. Clearly, the permute phase is the bottleneck in this respect, and we should choose r such that $\lceil 2^r/P \rceil + \min\{2^r, n/P\} \leq T - 1$. Plugging in the Ultra-II values of $P = 2048$ and $T = 64$, we get that for $n \geq PT \geq BC$ we should choose $r = 5$.

Of course, one may prefer that an average-case argument determines the radix size, especially since LSB radix sort anyway has poor worst-case cache behaviour.

We now heuristically analyse the permute phase, as it is the major contributor to the TLB misses, to calculate the number of TLB misses for $r = 6, \ldots, 11$. For all these values, the count array fits into one page and we may assume that the count page and the current source page, once loaded, will never be evicted. We further simplify the process of accesses to the TLB and ignore disturbances caused when the source or one of the destination pointers crosses a page boundary (as these are transient). With these simplifications, TLB misses on accesses to the destination array may be modelled as uniform random access to a set of 2^r pages, using an LRU TLB of size $T - 2$. The probability of a TLB miss is then easily calculated to be $(2^r - (T - 2))/2^r$. This suggests that choosing $r = 6$ on our Ultra-II still gives a relatively low miss rate on average (miss probability 1/32), but choosing $r = 7$ is significantly worse (miss probability 1/2).

Experiments suggest that on random data, for $r = 1, \ldots, 5$ a single permute phase with radix r takes about the same time, as expected. Also, for $r = 7$ the permute time is—as expected—considerably (about 150%) slower than $r \leq 5$. However even $r = 6$ is about 25% slower than $r \leq 5$. This is probably because in practice T is effectively 61 or 62—it seems that the operating system reserves a few TLB entries for itself and locks them to prevent them from being evicted. Even using the simplistic estimate above, we should get a miss probability in the 1/13 to 1/16 range.

The choice $r = 5$ which guarantees good TLB performance turns out not to give the best performance on random data: it requires seven passes for sorting 32-bit data, at the end of which the keys are in a temporary array and have to be copied back into the original input array. The following sequence of radices saves one pass, avoids the extra copy at the end and also has other advantages noted below: $5, 5, 6, 5, 5, 6$.

3.3.2 *Speeding up the count phase.* The very small radix sizes suggested in the previous section mean that the number of passes over the data can be very large. In this situation the count phase (which normally contributes little to cache or TLB misses) can be a significant part of the computation. We now discuss ways of speeding up the count phase.

The first is Friend's improvement [Friend 1956], which coalesces two count phases into one. More precisely, the frequencies of two successive digits (say the ith and $i+1$st digits) are counted in one pass by updating two count arrays for each record in the input. An immediate extension is to count frequencies for $k > 2$ passes in one phase, *k-tuple counting*. The main disadvantage is that accessing multiple count arrays may cause conflict misses, but for the small radix values we consider, the count arrays are miniscule and do not interfere significantly with each other. For $r = 6$ and 32-bit data, we found that counting all six digits in one pass was considerably faster than three passes each counting two digits.

We found the following method to be still faster, as it greatly reduced the instruction count. To count the i-th and $i+1$-st digit simultaneously, we concatenate the i-th and $i + 1$-st digits (giving a $2r$-bit number) and increment a single count array of size 2^{2r}. In practice of course, we would simply mask out the relevant $2r$ bits in one step. From the $2r$-bit counts, frequencies for the i-th and $i + 1$-st passes are easily extracted. This approach can extend to coalescing three or more

count phases into one. For example, for the radix sequence mentioned at the end of Section 3.3.1, we can obtain all six counts in one pass, by incrementing two 16-bit arrays for each key. The first array would implicitly contain frequencies for the three least signficant digits (totalling $6 + 5 + 5 = 16$ bits), and the second likewise for the three most significant digits. Note that this approach may increase conflict misses, and so may not work for all architectures.

4. EXPLICIT BLOCK TRANSFER

Let $\mathbf{C}(a, B, C, P, T)$ be the model described in Section 2, where a is the associativity of the cache, B is the block size, C is the capacity of the cache in blocks, P is the page size and T is the number of TLB entries. Recall that the model assumes that $BC \leq PT$.

Let $\mathbf{I}(B, M)$ denote the following model, which is the external-memory model of [Aggarwal and Vitter 1988], specialised to disallow parallel disk access. There is a fast main memory, which is organised as M/B blocks $m_1, \ldots, m_{M/B}$ of B words each and an unbounded secondary memory, which is organised as blocks $d_1, d_2, \ldots$ of B words each as well. An algorithm in this model performs computations on the data in main memory, or else it performs an *I/O step*, which copies a specified block from main memory into a specified block in secondary memory or vice versa.

As noted in [Sen and Chatterjee 2000], an algorithm A designed for the $\mathbf{I}\ (B, BC/2)$ model can be emulated by an algorithm A' in the $\mathbf{C}(a, B, C, P, T)$ model, such that A' incurs at most $O(1)$ (amortised) cache misses for every I/O performed by A. Their emulation is quite simple: an array Mem of size $C/2$, each entry of which can hold B words, emulates the main memory of A, with m_i corresponding to $Mem[i]$. The secondary memory of A is emulated by another array D, each entry of which can also hold B words, and an I/O operation is emulated by copying $Mem[i]$ to $D[j]$ or vice versa. In order to avoid conflict misses during copying if $Mem[i]$ and $D[j]$ map to the same set, the copying may be done through an intermediate buffer of B words. Two such buffers may be needed; it suffices to extend Mem by two elements and use $Mem[C/2+1]$ and $Mem[C/2+2]$ as these buffers.

The (extended) array Mem occupies at most $T/2+1$ pages. If we assume that the TLB has only $T/2+1$ entries but that TLB replacement is done by an optimal off-line algorithm, the optimal off-line algorithm need never make more than 2 misses for each I/O performed by A—it can simply swap in the appropriate page for the D array, evicting a Mem page which will not be needed immediately, and swap back the Mem page just evicted at the expense of the D array page once the copying is complete. Since a LRU TLB with T entries never makes more than twice as many misses as an optimal TLB with $T/2+1$ entries on a given sequence of page accesses (Theorem 1), it follows that the emulation makes $O(I)$ TLB misses over the course of the emulation, if A performs I I/O operations. Combining this with the result of [Sen and Chatterjee 2000], we have:

THEOREM 2. *An algorithm A in the $\mathbf{I}(B, BC/2)$ model which performs I I/Os and t operations on data in main memory can be converted into an equivalent one A' in the $\mathbf{C}(a, B, C, P, T)$ model which performs at most $O(t + I \cdot B)$ operations, $O(I)$ cache misses and $O(I)$ TLB misses.*

The external-memory algorithm for LSB radix sort is the obvious one. The radix

r is chosen so that $2^r \leq M/(2B)$; this enables buffers of B keys to be maintained in main memory for each of the 2^r classes during the permute phase, while leaving space for auxiliary data structures such as the count array and the current source array block. We describe only the permute phase, and assume that the count array is already stored in main memory. The algorithm reads successive blocks from the source array (which is stored in secondary memory), and moves each key in the current source array block to its main-memory buffer. When the buffer is full, it is copied out into the appropriate block of the destination array (which is also stored in secondary memory). With a radix chosen as above and provided $n \geq M$, the algorithm performs $O(n/B)$ I/Os per pass. By Theorem 2 we get:

COROLLARY 1. *For any associativity $a \geq 1$, the emulation of external-memory radix sort with radix $r \leq \log(C/4)$ incurs $O(n/B)$ cache and TLB misses per pass in the worst case.*

In the future we refer to this as EBT (explicit block transfer) radix sort.

4.1 Implementing EBT Radix Sort

When implementing EBT radix sort, one may dispense with some of the steps in the emulation. The probability of conflicts between Mem and D in a block copy on random input seems (subjectively) so low that one should copy blocks directly, rather than through intermediate buffers. If pathological inputs are a concern then one should randomise the starting location of Mem as noted in [Sen and Chatterjee 2000].

We also limit the use of the emulation to critical parts of the algorithm. In effect, the algorithm we implement is normal LSB radix sort, but with the permute phase modified as follows. We maintain an array of 2^r buffers, all of which can hold $\Omega(B)$ keys. When moving keys in the permute phase, we do not do so directly, but instead move the key from the source array to the buffer corresponding to its class. When the buffer is full, all keys in it are copied as a block to their final locations in the destination array.

In practice one could probably use larger radix values such as $r = \log(C/2)$. In our case, this would correspond to using $r = 12$ instead of $r = 11$. Since the number of passes is not reduced for 32-bit data, we stayed with the smaller radix.

5. PRE-SORTING

In this section we discuss two algorithms both of which pre-sort the keys in small groups to increase locality. Unless stated explicitly otherwise, we take the word 'key' in this section to mean the r-bit digit being sorted in one pass of PLSB or EPLSB radix sort.

5.1 PLSB Radix Sort

One pass of PLSB radix sort with radix r works in two stages. First we divide the input array of n keys into contiguous segments of $s \leq n$ keys each. Each segment is sorted using counting sort (a *local* sort) after which we sort the entire array using counting sort (a *global* sort). In each pass the time for sorting each of the $\lceil n/s \rceil$ local sorts is $O(s + 2^r)$ time and the time for the global sort is $O(n + 2^r)$, so the running time for one pass of PSLB radix sort is $O(n + n(2^r)/s)$.

An obvious optimisation is to accumulate counts for the global sort at the end of the count phase of each local sort. So the structure for the algorithm is as follows, where `Input` and `Temp` are arrays of size n and `GlobalCount` and `LocalCount` are arrays of size 2^r and the segments are numbered from $0, \ldots, \lceil n/s \rceil - 1$:

```
1    initialise GlobalCount
2    for i = 0,...,⌈n/s⌉ − 1 do
2.1    initialise LocalCount
2.2    count frequencies of key values in segment i of Input
2.3    accumulate values in GlobalCount
2.4    prefix sum LocalCount
2.5    permute from segment i of  Input to segment i of  Temp
3   prefix sum GlobalCount
4   permute globally from  Temp to  Input
```

The intuition for the algorithm is that each local sort groups keys of the same class together and during the global sort we move sequences of keys to successive locations in the sorted array, thus reducing TLB and conflict misses between accesses to the destination array. We now give some more detailed intuition. We denote by γ the quantity $s/2^r$, and we will require that $\gamma \geq 2$. An immediate consequence is that one pass of PLSB radix sort runs in $O(n)$ time. To minimise the operation count, we would like to choose r as large as possible, but this pushes up s. To keep cache and TLB misses to a minimum in the local sorts, we would like the segments of the source and destination arrays in each local sort to fit into cache and TLB-addressable pages. This suggests that we must set $s \leq BC$ (the analysis below assumes that $s \leq BC/2$).

We move straightaway to the global permute phase now, since this was previously the source of most cache and TLB misses. An elementary observation is that an array of k consecutive items will span at most $\lceil k/P \rceil + 1$ pages or $\lceil k/B \rceil + 1$ blocks, and hence (if there are no pathological conflict misses) copying k consecutive items from the source array to consecutive locations in the destination array will incur at most this many cache and TLB misses in destination array accesses. If $k_{i,j}$ is the number of keys in class i in segment j, then the total number of destination array TLB misses would be roughly $\sum_{i=0}^{2^r-1} \sum_{j=0}^{n/s} \lceil k_{i,j}/P \rceil + 1 \leq O(n/P + (n/s) \cdot 2^r) = O(n/P + n/\gamma)$. Similarly, the number of cache misses would be about $O(n/B + n/\gamma)$. This suggests that we need $\gamma = \Omega(B)$ for an optimal number $O(n/B)$ of cache misses, and the number of TLB misses would then also be $O(n/B)$. This says that the radix r should satisfy $2^r = O(C)$, which is similar to the radix limitations of explicit block transfer.

5.1.1 *Worst-case cache and TLB analysis for PLSB.* We now give a worst-case cache and TLB miss analysis for PLSB. To determine the number of cache and TLB misses during the local sorts our approach is to describe our own cache and TLB replacement policies and determine the worst-case number of cache and TLB misses using these policies. Given that using an optimal replacement policy would make no more misses we calculate an upper bound on the cache and TLB misses given an LRU replacement policy.

Without loss of generality we assume the cache and TLB are empty at the start

of the local sorts, at the start of the prefix sum of `GlobalCount` and at start of the global permute.

THEOREM 3. *For $s = CB/2$ and $\gamma \geq 2$, the worst-case number of cache misses in one pass of PLSB when sorting n keys with an $a \geq 4$ ways associative cache is at most:*

$$\frac{C}{2\gamma} + \frac{2n}{B} + \frac{5n}{\gamma} + 1 + \frac{a}{a - \lceil a/2 \rceil - \lceil a/(2\gamma) \rceil} \left(\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \right)$$

where s is the number of keys in a local sort, $\gamma = s/2^r$ and r is the radix.

PROOF. During each local sort our replacement algorithm reserves sufficient ways to keep `LocalCount` in cache once it has been loaded. Our replacement algorithm also reserves sufficient ways to keep a segment of `Temp`, once loaded, in cache during step 2.5. It also reserves 1 further way for loading a segment of `Input` in cache during steps 2.2 and 2.5, it uses the same way to load `GlobalCount` during step 2.3. Since $C = Sa$ and $s = BSa/2$, `LocalCount` is $2^r/B = Sa/(2\gamma)$ blocks in size and our replacement algorithm requires $\lceil a/(2\gamma) \rceil$ ways for `LocalCount`. Each segment of `Temp` is $Sa/2$ blocks in size and our replacement algorithm requires $\lceil a/2 \rceil$ ways for a segment of `Temp`.

With our replacement algorithm, once `LocalCount` has been loaded into cache for the first local sort it will remain in cache for all further local sorts. An upper bound on the cache misses for the local sorts in one pass of PLSB using our replacement algorithm is derived as follows:

—$\lceil Sa/(2\gamma) \rceil \leq C/(2\gamma) + 1$ misses for loading `LocalCount` once.

—$\lceil n/B \rceil$ misses for loading `Input` into cache over all local sorts in step 2.2

—$\lceil 2n/(BSa) \rceil \cdot \lceil Sa/(2\gamma) \rceil \leq \frac{n}{B\gamma} + \frac{2n}{CB} + \frac{C}{2\gamma} + 1$ misses over all local sorts for loading `GlobalCount` into cache in step 2.3.

—$2\lceil n/B \rceil$ cache misses for loading `Input` and `Temp` into cache over all local sorts in step 2.5.

Over all local sorts our replacement algorithm uses $a^* = \lceil a/2 \rceil + \lceil a/(2\gamma) \rceil + 1$ ways and there are at most:

$$\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \tag{5}$$

cache misses. Using Theorem 1 we get that the number of cache misses in an a-way associative LRU cache is at most:

$$\frac{a}{a - \lceil a/2 \rceil - \lceil a/(2\gamma) \rceil} \left(\frac{C}{\gamma} + \frac{3n}{B} + \frac{n}{B\gamma} + \frac{2n}{CB} + 5 \right) \tag{6}$$

For the prefix sum of `GlobalCount` in step 3 and the global permute in step 4 we do not utilise the optimal cache. During the prefix sum of `GlobalCount` there are

$$\left\lceil \frac{Sa}{2\gamma} \right\rceil \leq \frac{C}{2\gamma} + 1$$

cache misses. For the global permute, we divide the memory accesses into $(n/s) \cdot m = n/\gamma$ *epochs*. In epoch (i, j), for $0 \leq i \leq m-1$ and $0 \leq j \leq n/s$, we move the $k_{i,j}$ keys with value i from local sort j to their final destination. In this epoch we access

at most $2(\lceil k_{i,j}/B \rceil + 1)$ blocks from the `Input` and `Temp` arrays in all, and one block from the count array. The number of misses is at most $2\lceil k_{i,j}/B \rceil + 3 \leq 2k_{i,j}/B + 5$. Summing over all epochs we get that the number of cache misses for the global permute is at most:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n/s} (2k_{i,j}/B + 5) = \frac{2n}{B} + \frac{5n}{\gamma} \tag{7}$$

Summing the cache misses for the local sorts, global prefix sum and global permute gives the worst-case number of cache misses for one pass of PLSB. □

COROLLARY 2. *If $\gamma = \Omega(B)$ then there are $O(n/B)$ cache misses in one pass of PLSB in an a-way associative cache with LRU replacement policy.*

THEOREM 4. *For $s = BC/2 \leq PT/2$ and $\gamma \geq 2$ the worst-case number of TLB misses in one pass of PLSB when sorting n keys using a TLB of size $T \geq 4$ is at most:*

$$\frac{T}{2\gamma} + \frac{2n}{P} + \frac{5n}{\gamma} + 1 + \frac{T}{T - \lceil T/2 \rceil - \lceil T/(2\gamma) \rceil} \left(\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \right)$$

where s is the number of keys in a local sort, $\gamma = s/2^r$ and r is the radix.

PROOF. During each local sort our replacement algorithm reserves sufficient TLB entries to keep page translations for `LocalCount`, once loaded, in the TLB during steps 2.1, 2.2, 2.4 and 2.5. Our replacement algorithm also reserves sufficient TLB entries to keep page translations for a segment of `Temp`, once loaded, in the TLB entries during 2.5. It also uses 1 further TLB entry for the page translations for a segment of `Input` during steps 2.2 and 2.5, it uses the same TLB entry to load `GlobalCount` during step 2.3. Since $s = BC/2 \leq PT/2$, `LocalCount` requires at most $2^r/P = T/(2\gamma)$ pages and our replacement algorithm requires at most $\lceil T/(2\gamma) \rceil$ TLB entries for `LocalCount`. Each segment of `Temp` is at most $T/2$ pages in size and our replacement algorithm requires at most $\lceil T/2 \rceil$ TLB entries for a segment of `Temp`.

With our replacement algorithm, once translations for `LocalCount` have been loaded into the TLB for the first local sort they will remain there for all further local sorts. An upper bound on the TLB misses for the local sorts in one pass of PLSB using our replacement algorithm, is as follows:

—$\lceil T/(2\gamma) \rceil$ misses for `LocalCount` in step 2.1 for the first local sort.

—$\lceil n/P \rceil$ misses for `Input` over all local sorts in step 2.2

—$\lceil 2n/(PT) \rceil \cdot \lceil T/(2\gamma) \rceil \leq \frac{n}{P\gamma} + \frac{2n}{PT} + \frac{T}{2\gamma} + 1$ misses over all local sorts for `GlobalCount` in step 2.3.

—$2\lceil n/P \rceil$ misses for `Input` and `Temp` over all local sorts in step 2.5.

Over all local sorts our replacement algorithm uses $T^* = \lceil T/2 \rceil + \lceil T/(2\gamma) \rceil + 1$ TLB entries and there are at most:

$$\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \tag{8}$$

TLB misses. Using Theorem 1 we get that the number of TLB misses using a LRU TLB of size T is at most:

$$\frac{T}{T - \lceil T/2 \rceil - \lceil T/(2\gamma) \rceil} \left(\frac{T}{\gamma} + \frac{3n}{P} + \frac{n}{P\gamma} + \frac{2n}{PT} + 5 \right) \tag{9}$$

For the prefix sum of `GlobalCount` in step 3 and the global permute in step 4 we do not utilise results for optimal TLB. There are

$$\left\lceil \frac{T}{2\gamma} \right\rceil \leq \frac{T}{2\gamma} + 1$$

TLB misses for the prefix sum of `GlobalCount`. For the global permute, we divide the memory accesses into $(n/s) \cdot m = n/\gamma$ *epochs.* In epoch (i, j), for $0 \leq i \leq m-1$ and $0 \leq j \leq n/s$, we move the $k_{i,j}$ keys with value i from local sort j to their final destination. In this epoch we access at most $2(\lceil k_{i,j}/P \rceil + 1)$ pages from the `Input` and `Temp` arrays in all, and one page from the count array. The number of misses is at most $2\lceil k_{i,j}/P \rceil + 3 \leq 2k_{i,j}/P + 5$. Summing over all epochs we get:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n/s} (2k_{i,j}/P + 5) = \frac{2n}{P} + \frac{5n}{\gamma} \tag{10}$$

Summing the TLB misses for the local sorts, global prefix sum and global permute gives the worst-case number of TLB misses for one pass of PLSB. □

COROLLARY 3. *If $\gamma = \Omega(B)$ then there are $O(n/B)$ TLB misses in one pass of PLSB.*

5.1.2 *Implementing PLSB Radix Sort.* Clearly, from the analyses above, we would like generally to get a large value of $\gamma = s/2^r$, as this tends to reduce cache and TLB misses in the global permute. The analysis above used $s = (BC)/2$, and we now review the local sort phase in order to find the largest 'practical' value of s.

To keep TLB misses low we want to ensure that the working set of a local sort can comfortably be held in TLB. On the Sun Ultra-II the TLB has size 64, so we should limit s to be at most kP for some integer k in the high 50s; this allows a couple of system TLB entries, a source page entry and a handful of count page entries all to fit in TLB. On pathological data, choosing $s > (BC)/2$ can give many conflict misses in the local sorts, even on a 4-way associative cache. However, by using randomisation, we ensure that no single input gives a bad running time, and so our algorithm has few expected conflict misses on any input. More precisely, by ensuring that each of the source and destination arrays start independently at locations which are uniformly distributed over the range $\{0, \ldots, BC - 1\}$, we get very few conflict misses in the local sorts even on direct-mapped caches, regardless of the input (this is easily shown using arguments similar to those of [Mehlhorn and Sanders 2000]). A side effect of the randomisation is that there are no pathological inputs for the global permute on a direct-mapped cache.

We use a value $s \approx 117,400$, corresponding to k between 57 and 58. This particular number is 'good' in some way for EPLSB radix sort and we choose it again in PLSB radix sort for convenience, although it has no significance for PLSB radix beyond that already discussed. Also, experimentally we determined that $r = 11$ is the best radix to use for sorting 32-bit data. This gives $\gamma \approx 57 \approx 3.5B$).

One effective low-level optimisation for PLSB is that during the global permute, we do not use the standard counting sort permute algorithm. Instead, before moving a record from `Temp` back to `Input` in the global permute, we first scan ahead in `Temp` to find a maximal set of k consecutive records which have the same key value as the next record to be moved. After moving these k records as a block we increase the relevant count array location by k. This saves significantly on instructions and also on writes to the count array (even though this is not part of our model, writes in general are a little expensive as they are handled by the L2 cache rather than the L1 cache). After this optimisation, the global permute is less than 30% slower than a straight copy from `Temp` to `Input`.

5.2 EPLSB Radix Sort

In EPLSB radix sort with radix r, each pass is again done in two stages. We use the term 'global key' to refer to the r-bit sort key for a particular pass, and we use the term 'local key' to refer to the r' most significant bits of the global key, where $r' \leq r$ is a parameter whose value we will choose later. If we choose $r' = r$ EPLSB radix sort essentially reduces to PLSB radix sort.

We assume again that $n \geq BC$. First, as in PLSB radix sort, we divide the input array of n records into contiguous segments of $s \leq n$ records each, and sort each segment. The local sorts now only sort according to the local key, but the global permute moves records to their final location according to the global key. During the global permute, we again process blocks of records from each segment, where a block consists of successive records with the same local key value. Although it is no longer the case that all records in a block are moved together, they belong to at most $\Delta = 2^{r-r'}$ different (global) classes, which limits the number of active locations in the destination array. Intuitively, we can let $\Delta = \Theta(T)$ and still keep TLB misses low, thus extending the radix size of PLSB by a factor of roughly T.

A more precise description of the algorithm follows. `Input` and `Temp` are arrays of size n. `Input` contains the input records, and at the end of the pass, `Input` will once again contain the records in sorted order with respect to the global key. `GlobalCount` and `LocalCount` are arrays of size 2^r and $2^{r'}$ respectively, and the segments are numbered from $0, \ldots, \lceil n/s \rceil - 1$.

```
1    /* perform all local sorts */
1.0  for i = 0,...,⌈n/s⌉ − 1 do
1.1    initialise LocalCount
1.2    count frequencies of local key values in segment i of Input
1.3    prefix sum LocalCount
1.4    permute records from segment i of Input to segment i of Temp
       /* Step 1.4 moves records according to local key value */
2   Initialise GlobalCount
3   Count frequencies of global key values.
4   /* global permute */
4.0  for j = 0,...,2^r' − 1
4.1    for i = 0,...,⌈n/s⌉ − 1
4.2       move all records with local key value j in segment i
          of Temp to their final location in Input.
```

```
/* Step 4.2 moves records according to global key value */
```

We now fix some parameter choices. We choose $s = (BC)/2$, $r' \leq \log C$ and $r \leq \min\{\log(BC/2), r' + \log(T/2)\}$. This ensures that $\gamma = s/2^{r'} \geq B/2$ and $\Delta \leq T/2$.

We now estimate the number of instructions, cache and TLB misses in one pass of EPLSB radix sort. We begin with instruction counts. Step 1 takes $O(n + n(2^{r'})/s) = O(n + n/\gamma) = O(n)$ time. Steps 2 and 3 take $O(n + 2^r) = O(n)$ time, since $2^r \leq BC \leq n$. Letting $k_{i,j}$ denote the number of keys with local sort key value j in segment i of the input, Step 4 takes $\sum_{j=0}^{2^{r'}-1} \sum_{i=0}^{\lceil n/s \rceil} (k_{i,j} + 1) = O(n + n/\gamma) = O(n)$ time, giving an overall bound of $O(n)$ time for one pass of EPLSB radix sort.

We now move to TLB misses. The analysis of the TLB misses for Step 1 is analogous to the analysis of Step 2 of PLSB, and we can conclude that there are $O(n/B)$ misses in this step. Steps 2 and 3 have negligible TLB misses (note that $2^r \leq BC/2$, so the count array requires at most $T/2$ pages). In Substep 4.2, note that we are moving elements belonging to Δ different global key classes in each iteration of the loop of Step 4.0. The count information for these classes is spread across at most two `GlobalCount` pages, as $\Delta \leq T \leq P$. We now calculate the number of misses made by an optimal TLB of size $\Delta + 3$. The optimal TLB would make no more than the following misses:

—By always evicting a page from `Input` when an access to `Input` causes a miss, at most $\sum_{j=0}^{2^{r'}-1} \sum_{i=0}^{\lceil n/s \rceil} \lceil k_{i,j}/P \rceil + 1 \leq n/P + 2n/\gamma = O(n/B)$ misses due to accesses to `Input`, where $k_{i,j}$ is as above.

—Similarly, accesses to `Temp` cause at most $O(n/P + \Delta \cdot n/s)$ misses. Since $\Delta n/s \leq 2\Delta n/(BC) \leq 2nT/(BC) = O(n/B)$, the total number of misses is again $O(n/B)$ (and should usually be considerably smaller, as $T \ll C$).

—At most $\lceil 2^r/P \rceil = O(n/P)$ misses for accesses to `GlobalCount`.

Hence the minimum size of the optimal TLB required to achieve $O(n/B)$ misses is no more than $\Delta + 3$. This is at most $2T/3$ if P and T are sufficiently large, so an LRU TLB of size T will also make $O(n/B)$ misses in the worst case.

We now turn to cache misses. As $s = (BC)/2$ and $2^r \leq C$, the number of misses in Step 1 is $O(n/B)$ in the worst case. For a direct-mapped cache this requires `Input` and `Temp` to be aligned so that corresponding segments of these arrays are mapped to distinct parts of the cache; we can dispense with this assumption for an a-way associative cache with $a \geq 4$.

In Step 3 the number of cache misses is $O(n/B)$ in the worst case if the cache is at least two-way associative (recall that $2^r \leq (BC)/2$ so the count array can occupy at most one way of each set).

In Step 4, again randomising the start of `Temp` and `Input` reduces the number of conflict misses between (i) `GlobalCount` and `Temp`; (ii) `Input` and `Temp` and (iii) `GlobalCount` and `Input` to negligible levels. However, this does not solve the problem of conflicts between the $T/2$ active locations in `Input`, which can conflict with each other. One solution to this problem is to randomise the start of each (global key) class, as suggested in [Mehlhorn and Sanders 2000]. If this is done, then it follows from the analysis there that the number of conflict misses should be small as $T/2 \ll C$ so there are $O(n/B)$ misses.

5.2.1 *Implementing EPLSB radix sort.* For the UltraSparc-II parameters, the maximum permissible value of r is 16 (since $BC/2 = 2^{16}$). As is the case for PLSB, it is beneficial to EPLSB to have a value of γ as large as possible. This means making s as large as possible and r' as small as possible. The smallest value for r' which gives few worst-case TLB misses is $16 - \log(T/2) = 11$. As in PLSB radix sort we randomise the start locations of the source and destination arrays; this reduces the number of cache conflict misses in Steps 1-3 to negligible levels (according to our model).

We again choose s to be about $58P$ to keep TLB misses in local sorts at a minimum. Here, however, the value of s also slightly affects the global permute, so s is fixed later.

We optimise the global counts by scanning the array `Temp` for the purpose of global counting, rather than `Input`. This improves performance by improving the locality of the global counts in a way that our model appears not to capture. A further improvement is to perform global counts on segment i of `Temp` right after the local sort for segment i, as most of `Temp` will be in cache.

We make no special optimisations to remove the main problem with the global permute, namely that EPLSB radix sort is susceptible to pathological data. The only possibility, as noted above, is to randomise the start of each class as in [Mehlhorn and Sanders 2000]. Using their approach directly would have a prohibitive space cost, but since there are only only $T/2$ active destination pointers in EPLSB radix sort, it is possible that one may not need to randomise by as much as suggested in [Mehlhorn and Sanders 2000]. It would be interesting to explore this aspect further.

As we will in the section on experimental results, in EPLSB radix sort worst-case inputs cause many TLB misses during the global permute when the number of active destination pointers $\Delta = T$. We also see that since $\Delta \ll C$ and the cache is physically mapped the theoretical worst-case input does not actually cause many more cache misses than random inputs. This justifies our design descision to make EPLSB radix sort robust against worst-case inputs for TLB misses but not for cache misses.

We do incorporate a second-tier optimisation for the global permute. The global permute for EPLSB radix sort makes several 'passes' over the `Temp` array (each pass corresponds to an iteration of the loop of Step 4.0). Each 'pass' moves a relatively small group of keys from a segment to their final destinations, before moving on to the next segment. As the end of a group may not lie on a cache boundary in general, it would be helpful if the block which contains the end of the group stays in cache for the next 'pass': this saves an extra miss for each block. For random data this can be significant: each group is typically only 3.5 cache blocks long, so the number of `Temp` array misses would be reduced by about 20%.

In practice the number of segments is small (less than 300 for $n = 32000000$) compared to C, so active `Temp` blocks could easily stay in cache between 'passes'. Intuitively the chances of this happening are improved if the starts of the segments were mapped to blocks which are roughly equally spaced in cache. Some values of s are better than others at achieving this: in [Rahman and Raman 1999] it is noted that values such as $s = (1 - 1/(9 + \phi^{-1}))BC \approx 117,400 \approx 57.3P$, where $\phi = (1 + \sqrt{5})/2$, are provably good for this. This is the value used in our code.

This optimisation does not always help, but it never hurts either.

6. ASYMPTOTIC NUMBER OF CACHE AND TLB MISSES

It has been shown [Aggarwal and Vitter 1988] that any algorithm for sorting n keys must make $\Omega(n/B \log_C(n/B))$ cache misses. A direct reduction from their argument shows that the number of TLB misses for sorting must be at least $\Omega(n/P \log_T(n/P))$.

These lower bounds, however, apply only when all $n!$ permutations of the input keys are possible. When (stably) sorting w-bit keys, the number of possible permutations is at most 2^{wn}, which could be smaller or greater than $n!$. In particular, if the radix r is chosen appropriately, then one pass of r-bit LSB radix sort can be accomplished in $O(n/B)$ cache misses, or $O(n/P)$ TLB misses. This has also been observed by [Matias et al. 2000], who also show that $\Omega((n/B) \log_C \min\{(n/B), 2^w\})$ cache misses are necessary to sort w-bit integers. Again by reduction from their argument $\Omega((n/P) \log_T \min\{(n/P), 2^w\})$ TLB misses are necessary to sort w-bit integers.

We now compare the asymptotic performance of the algorithms. Recall that we assume that $T \leq C$, $B \leq P$, $BC \leq PT$ and $B \leq C$. First, we consider the case when T is not too small, i.e., $\log T = \Theta(\log C)$. In this case it suffices for optimality that the algorithm makes $O(n/B)$ cache and $O(n/P)$ TLB misses per pass, where each pass sorts according to a radix which is at least $\Omega(\log T)$. We first note:

PROPOSITION 6. *Assuming an a-way associative cache for some $a \geq 2$, LSB radix sort makes an optimal expected number of cache and TLB misses on any input, if $\log T = \Theta(\log C)$.*

PROOF. We choose the radix of LSB radix sort as $r = (1 - 1/a) \log T - 1$. Since $r \leq \log(T/2)$, standard LSB radix sort makes $O(n/P)$ TLB misses per pass in the worst case. Also, $2^r \leq T^{1-1/a} \leq C^{1-1/a} \leq C/B^{1/a}$. If we assume that the algorithm uses randomised memory allocation as described in [Mehlhorn and Sanders 2000], it follows from the results there that if $2^r \leq C/B^{1/a}$, LSB radix sort makes $O(n/B)$ expected cache misses per pass on an a-way associative cache on any input. Observing that $r = \Omega(\log C)$ completes the proof. □

An analogue for worst-case behaviour follows:

PROPOSITION 7. *Assuming an a-way associative cache for some $a \geq 2$, one can sort incurring an optimal expected number of cache and TLB misses on any input, if $\log T = \Theta(\log C)$.*

PROOF. The algorithm is essentially PLSB radix sort modified to use EBT radix sort for the local sorts. Firstly, the input is divided into segments of size $PT/4$ and each segment is locally sorted with respect to the current r-bit digit using EBT radix sort, where $r = \log T - c \leq \log C - O(1)$ for some constant $c \geq 0$. By choosing c large enough we can ensure that all pages for each local sort fit into the TLB. In this case, EBT radix sort incurs $O(n/B)$ cache and $O(n/P)$ TLB misses, summed over all local sorts. The global permute then moves keys to their final destination as before. Since $\gamma = \Omega(P)$ here, the global permute also makes $O(n/B)$ cache and $O(n/P)$ TLB misses per pass, provided $a \geq 2$ by Theorems 3 and 4. As there are $\Theta(w/\log C)$ passes, the proposition follows. □

If $\log T = o(\log C)$ we do not know how to simultaneously get asymptotically optimal TLB and cache misses. EBT radix sort or PLSB radix sort with radix $\epsilon \log C$ for some constant $1 \geq \epsilon > 0$ makes $O(n/(BC^{1-\epsilon}))$ page misses per pass, while keeping the number of cache misses asymptotically optimal. However, the model does not rule out $P \gg BC$, so this can be non-optimal. Approaches based on choosing a radix of $\Theta(\log T)$ yield TLB-optimality but make too many cache misses.

7. EXPERIMENTAL RESULTS

The following table describes the algorithms that we tested and gives the names we will use for them. All algorithms were coded in C and all code compiled using `gcc 2.8.1`. Our experiments were run on a Sun Ultra II with 2×300 MHz processors and 1 GB of memory.

Name	Description
LSBr	LSB radix sort using r-bit radix with $\lceil 32/r \rceil$-tuple counting
FLSBr	LSB radix sort using r-bit radix with Friend's improvement
LSB556	LSB radix sort where a 16-bit count phase gathers count information for three permute phases. The first two of these permute phases use 5-bit radix and the third uses a 6-bit radix.
PLSBr	PLSB radix sort using r-bit radix
EPLSBr	EPLSB radix sort using r-bit local radix and 16-bit global radix
EBTr	EBT radix sort using r-bit radix
QSort	cache-tuned (*memory tuned*) Quicksort[LaMarca and Ladner 1999]
MSort	cache-tuned (*tiled*) Mergesort [LaMarca and Ladner 1999]

7.1 Random Inputs

We first tested the algorithms on uniformly distributed random integers for $n = i \times 10^6$, $i = 1, 2, 4, 8, 16, 32$. We avoided using values of n near powers of 2, as [Rahman and Raman 1999] suggests that LSB radix sort may perform slightly poorly at these values of n.

Figure 2 shows the time per key for one permutation pass of LSB radix sort. We see that using an r-bit radix, when $2^r < T$ the running times remain constant and when $2^r \geq T$ the running times increase as TLB misses increase (at $r = 4, 5, 6, 7, 8$ there are very few cache conflict misses). At $r = 11$ almost all accesses to the destination array will cause TLB misses and the increase in running time at $r = 16$ is due to increasing cache conflict misses.

Figure 3 shows the overall running times for LSB6, FLSB6 and LSB556. We see that since the radix is small 6-tuple counting gives a faster running time than using Friend's improvement. Since a permutation phase using a 5-bit radix has the same running time as using a 4-bit radix and is faster than using a 6-bit radix, we see that LSB556, which has in total four 5-bit and two 6-bit permute phases is faster than LSB6 which has effectively five 6-bit and one 4-bit permute phases.

Figure 4 shows the overall running times for LSB radix sort. We found that $\lceil 32/r \rceil$-tuple counting was faster than using Friend's improvement for radix sizes $r \leq 8$ bits, so we give results using $\lceil 32/r \rceil$-tuple counting for $r = 4, 5, 6, 7, 8$ bits and using Friend's improvement for $r = 11, 16$ bits. We see that LSB6 is the fastest

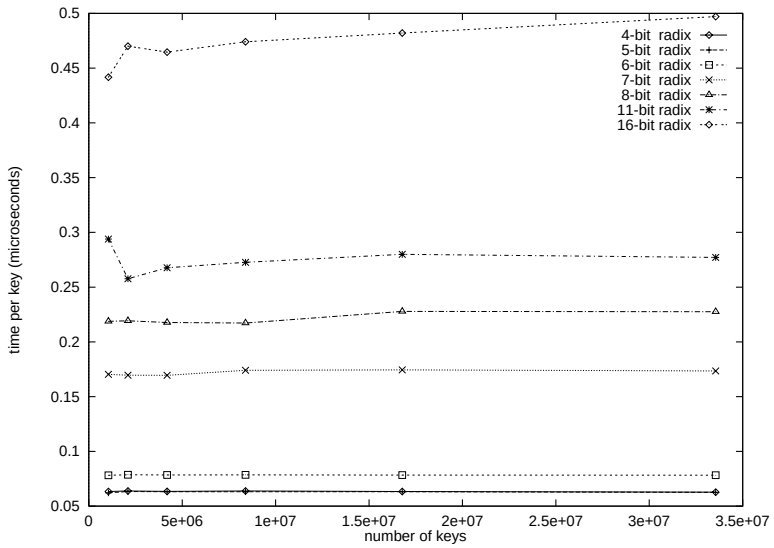

Fig. 2. Time per key for one permute pass of LSB radix sort using radix $4, 5, 6, 7, 8, 11$ and 16, at $n = 1M, 2M, 4M, 8M, 16M$ and $32M$. Keys are random 32-bit unsigned integers and M=10^6

Timings(sec)			
n	LSB556	LSB6	FLSB6
1M	0.57	0.64	0.67
2M	1.12	1.28	1.36
4M	2.20	2.56	2.72
8M	4.35	5.09	5.44
16M	8.57	10.22	10.84
32M	17.52	20.45	21.85

Fig. 3. Overall running times for LSB556, LSB6 and FLSB6 on random 32-bit unsigned integers, M=10^6.

Timings(sec)							
n	LSB 4	LSB 5	LSB 6	LSB 7	LSB 8	FLSB 11	FLSB 16
1M	0.74	0.68	*0.64*	0.93	0.95	0.90	0.92
2M	1.49	1.36	*1.28*	1.85	1.98	1.86	1.94
4M	2.99	2.73	*2.56*	3.70	3.86	3.86	4.08
8M	6.26	5.57	*5.09*	7.38	7.67	7.68	7.90
16M	11.94	10.94	*10.22*	14.75	15.79	15.23	15.99
32M	24.06	21.96	*20.45*	29.62	30.50	31.71	33.49

Fig. 4. Overall running times for LSB radix sort with varying radix on random 32-bit unsigned integers, M=10^6.

Timings(sec)								
n	EPLSB11	PLSB11	EBT11	LSB556	LSB6	FLSB11	QSort	MSort
1M	**0.46**	*0.47*	0.54	0.57	0.64	0.90	0.70	1.02
2M	**0.89**	*0.92*	1.09	1.12	1.28	1.86	1.50	2.22
4M	**1.74**	*1.82*	2.20	2.20	2.56	3.86	3.24	4.47
8M	**3.53**	*3.64*	4.49	4.35	5.09	7.68	6.89	9.71
16M	**7.48**	*7.85*	8.81	8.57	10.22	15.23	14.65	19.47
32M	**14.96**	*15.66*	17.55	17.52	20.45	31.71	31.69	41.89

Fig. 5. Comparison of TLB-tuned radix sorts: EPLSB11, PLSB11, EBT11, LSB556 and LSB6 versus cache tuned LSB11, QSort and MSort on random 32-bit unsigned integers, M=10^6.

on random inputs as it has very few cache and TLB misses. LSB7 has one less pass and about the same number of cache misses but almost half the accesses to the destination array cause TLB misses, so we see that it is almost 45% slower. LSB5 has very few cache and TLB misses but has one more pass than LSB6 and is slightly slower, however we will see later it is robust against worst-case inputs whereas LSB6 is not.

Figure 5 shows the overall running times for the TLB optimised radix sorting algorithms, EPLSB11, PLSB11, EBT11, LSB556 and LSB6 and the cache optimised algorithms LSB11, QSort and MSort. We see that EPLSB and PLSB radix sort outperform all the radix sort variants, getting speedups of 14+% and 10+% over the other TLB-optimised algorithms. These two algorithms obtain speedups of 52+% and 50+% over the cache optimised QSort, MSort and LSB11, for large n.

7.2 Testing robustness

We also tested the algorithms on the following input sequences:

—$0, \ldots, n-1$

—$0, \ldots, T-1$ repeated n/T times

with $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}, 2^{25}$, and where appropriate $s = BC/2 = PT/2$. We compared the running times for one pass of the algorithms LSB5, LSB6, EPLSB10, EPLSB11, PLSB11 and EBT11 with uniformly distributed random keys against the input sequences above. Since these experiments were for just one pass, the

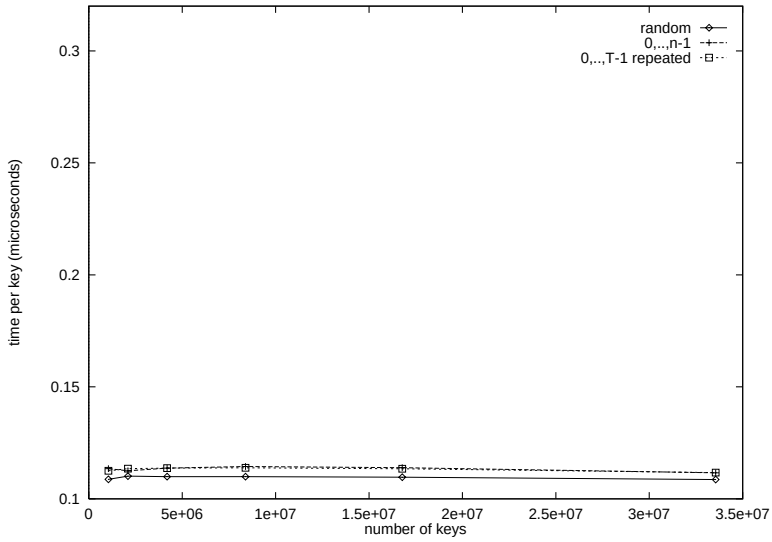

Fig. 6. Time per key for one pass of normal LSB with 5-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

algorithms were modified to gather counts for the first pass only, so one should not use these results to extrapolate the overall running times.

As stated in Proposition 3 the sequence $0, \ldots, n-1$ should cause worst-case cache misses at these values of n in LSB radix sort and one can easily show that it should also causes worst-case TLB when the radix $r \geq \log T$.

With these values of n and where appropriate s, the repeated sequence $0, \ldots, T-1$ should cause worst-case cache misses in LSB radix sort and EPLSB radix sort. This sequence should also cause worst-case TLB miss in LSB radix sort if radix $r \geq \log T$ and in EPLSB radix sort if the difference between the global and local radix lengths is at-least $\log T$.

Figures 6, 7, 8, 9, 10 and 11 summerise the results we obtained. We see in Figures 6 that in LSB5, where worst-case TLB misses are not in effect, there is only a slight, 3%, increase in running time with the above inputs versus the random keys. Similarly we see in Figure 11 with the repeated sequence $0, \ldots, T-1$ there is only a slight, 7%, increase in the running time of EPLSB11, where again worst-case TLB misses are not in effect. This is almost certainly due to the fact that the cache is physically mapped so we can not ensure that physical pages allocated to classes map to particular cache blocks. So we note that when 2^r is small we do not observe any significant increase in running time due to cache conflict misses even with worst-case inputs, this is in contrast to the results in Figure 2 when 2^r is large the increase in running times between radix 11 and radix 16 is almost entirely due to an increase in cache conflict misses.

We see in Figure 7 that in LSB6, where worst-case TLB misses are in effect, one pass with the above inputs takes almost 2.5 times as long as on the random keys. Similarly we see in Figure 10 that in EPLSB10 one pass with the repeated sequence

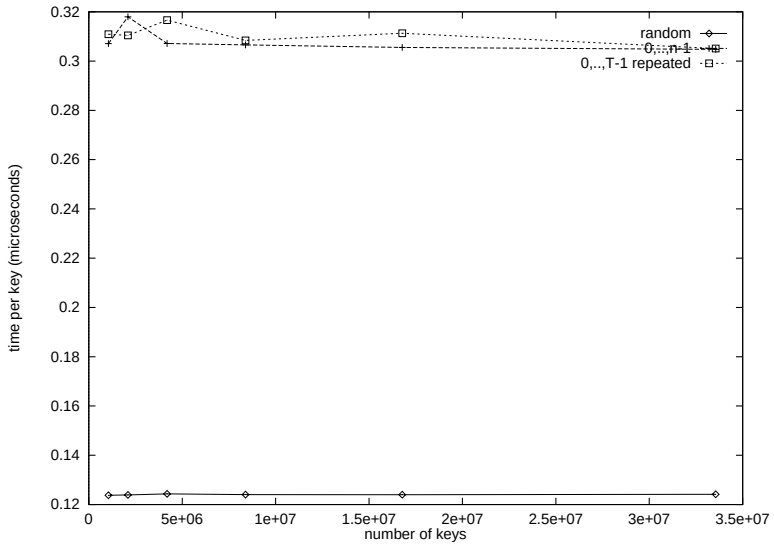

Fig. 7. Time per key for one pass of normal LSB with 6-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T - 1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

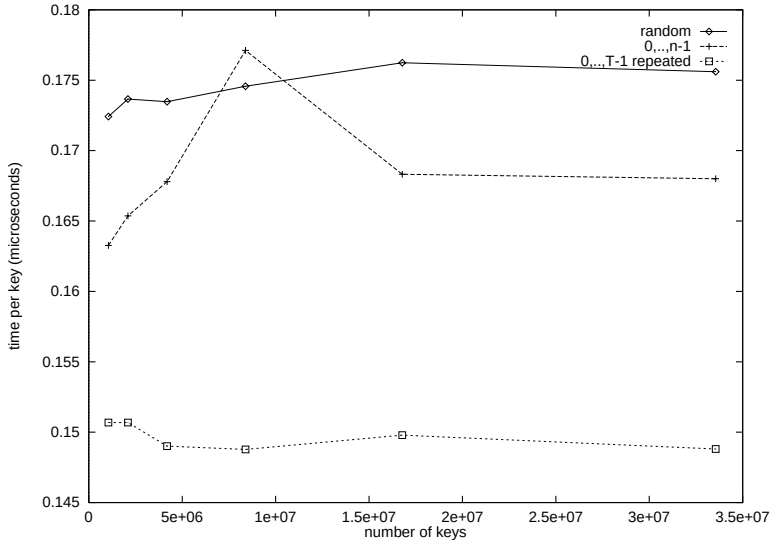

Fig. 8. Time per key for one pass of explict block transfer (EBT) with 11-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T - 1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

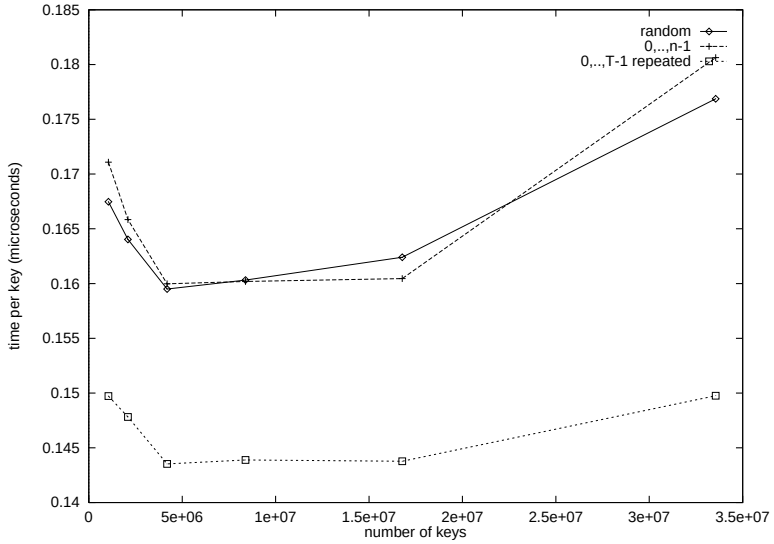

Fig. 9. Time per key for one pass of PLSB with 11-bit radix, on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T - 1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

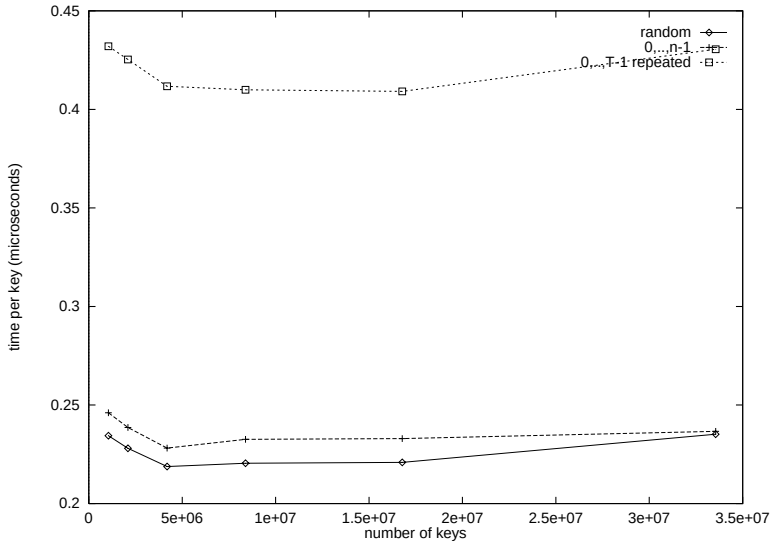

Fig. 10. Time per key for one pass of EPLSB with a 16-bit global key and 10-bit local key on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T - 1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

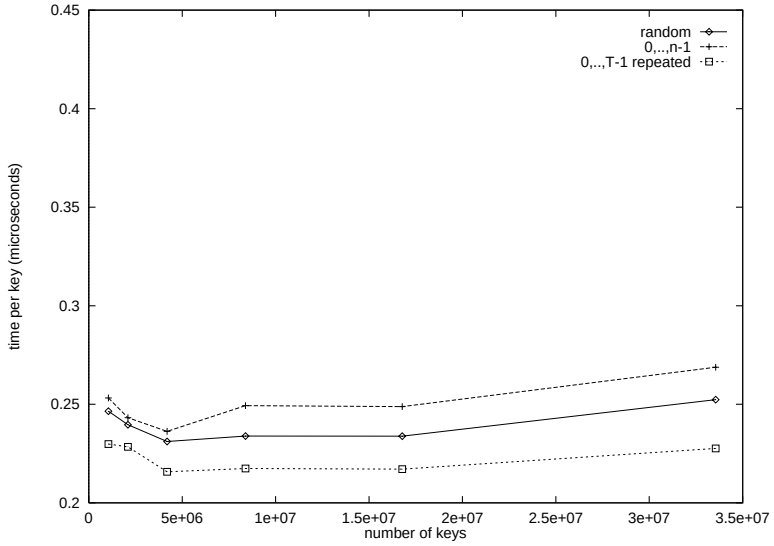

Fig. 11. Time per key for one pass of EPLSB with a 16-bit global key and 11-bit local key on random keys (random), on keys $0, \ldots, 2^n - 1$ (0,...n-1) and on repeated key sequence $0, \ldots, T-1$ (0,..,T-1 repeated) at $n = 2^{20}, 2^{21}, 2^{22}, 2^{23}, 2^{24}$ and 2^{25}

$0, \ldots, T-1$ takes almost twice as long as with the random keys, again worst-case TLB misses are in effect.

These results validate our design decision to tune EPLSB to prevent worst-case TLB behaviour but allow theoretically worst-case cache behaviour.

In theory PLSB should not have a worst-case behaviour, and we see in Figure 9 that the running time hardly changes with the above inputs versus random keys.

7.3 Sorted keys

We also tested the algorithms on sorted uniformly distributed random keys. We found that all the algorithms are faster or no slower with this input than with random keys and this is because they all gain from accessing successive locations in the count and destination arrays.

8. PERMUTING AN ARRAY

We consider the problem of permuting n integers from a source array to a destination array as specified in an additional permutation array. The naive implementation sequentially visits each element of the source and permutation array and places the value from the source array to some random location in the destination array as specified by the permutation array. If the permutation is random and if $n/P \gg T$ virtually every access to the destination array will cause a TLB miss and similarly if $n/B \gg C$ virtually every access to the destination array will cause a cache miss.

We have tested two approaches to permuting, both of which work in a 'most-signficant-bit first' fashion. Let k divide n for convenience, and define k equal-sized areas $D_1, \ldots, D_k$ in the destination array. All keys whose final destinations are in D_i are moved to *some* location in D_i in the first instance. We then recursively

Timings(sec)					
n	1M	2M	4M	8M	16M
Naive	0.30	0.66	1.40	2.89	6.03
PS	0.35	0.70	1.40	2.85	5.77
EBT	0.25	0.53	1.13	2.32	4.74

Fig. 12. Running times for naive, pre-sorting (PS) and explicit block copy (EBT) permutation algorithms for permuting 32-bit integer items. The permutations were random. M=2^{20}.

permute keys within D_i for $i = 1, \ldots, k$. The differences are in how the keys are moved to locations in D_i. One major disadvantage of both these approaches is that when we move an item to an intermediate location, we must move then the final destination of the item along with the item. This doubles the data to be moved. The improvements therefore are not as pronounced.

In each case we choose k to be $\min\{C/4, 4n/(CB)\}$, and distribute the keys into k areas. If the input size is small enough, then each D_i fits comfortably into cache and we finish the problem off with the naive algorithm, otherwise we recurse.

In each case, the distribution is similar to sorting the items according to the most significant $\log k$ bits of the destination address. In one case, we do this by moving items to an intermediate buffer as in the explicit block copy algorithm, and in the other, we do it by locally pre-sorting the items in segments according to which area they need to be moved to, and then moving them all in one global permute step. We omit the TLB and cache analyses of these algorithms, which are essentially the same as those of LSB radix sort variants, and merely give the experimental results, which show that the explicit block copying algorithm performs quite well in this context (see Fig 12).

9. CONCLUSIONS AND FUTURE WORK

We have shown the importance of minimising TLB misses in algorithms, even though it may require significantly more operations. We have shown the general technique of explict block transfer [Sen and Chatterjee 2000] minimises TLB misses and have applied it to radix sorting. We have also given two other techniques for reducing TLB misses in radix sorting, reducing working set size and pre-sorting.

We have given three radix sorting algorithms, pre-sorting LSB radix sort (PLSB radix sort), explicit block transfer LSB radix sort (EBT radix sort) and extended-radix PLSB radix sort (EPLSB radix sort), of which the first two have provably few cache and TLB misses, the last has provably few TLB misses.

The EBT technique is general, however, it does not always give the best practical performance. PLSB gives a fast running time and is robust. EPLSB has the advantage that it has fewer data moves than PLSB and EBT. Our current implementation of EPLSB can in theory suffer worst-case cache misses, even though this is not observed in experimental results, since it does not randomise the start of destination pointers during the global permute as this would add too much of an over-head. However since the number of random locations accessed is $T \ll C$ it would be interesting to determine if the algorithm can be made robust by using less randomisation than suggested by [Mehlhorn and Sanders 2000].

We have shown the pre-sorting and explict block transfer techniques applied

fruitfully to related problems such as permutations, but here more work could be done, including for instance work on specialised classes of permutations. Also the effect of applying pre-sorting and explicit block transfer to other algorithms such as MSB radix sort should be studied.

On a more speculative note, since pre-sorting and explict block transfer allow data to be moved in blocks we could further exploit the features available on most modern machine architectures, such as blocked copy operations between main memory addresses which by-pass the cache. Also since the local sorts in PLSB radix sort can be performed independently, on a machine with large *instruction-level parallelism* we could hope to speed that phase up by performing two or more local sorts simultaneously (by fusing the loops, for example).

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.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms*. MIT Press.

Friend, E. H. 1956. Sorting on electronic computer systems. *Journal of the ACM 3*, 134–168.

Frigo, M., Leiserson, C. E., Prokop, H., and Ramachandran, S. 1999. Cache-oblivious algorithms. In *Proc. 40th IEEE FOCS* (1999), pp. 285–298.

Gannon, D. and Jalby, W. 1987. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass Eds., *The Characteristics of Parallel Algorithms*. MIT Press.

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.

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*, pp. 370–379, 1997.

Matias, Y., Segal, E., and Vitter, J. S. 2000. Efficient bundle sorting. In *Proceedings of the 11th Annual SIAM/ACM Symposium on Discrete Algorithms* (2000).

Mehlhorn, K. and Sanders, P. 2000. Accessing multiple sequences through set-associative cache. Unpublished manuscript. Preliminary version in *Proc. 26th ICALP*, LNCS 1555, 1999.

Moret, B. and Shapiro, H. 1994. An empirical assessment of algorithms for constructing a minimum spanning tree. In *Computational Support for Discrete Mathematics*, Volume 15 of *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*, pp. 99–117. American Mathematical Society.

Rahman, N. and Raman, R. 1999. Analysing cache effects in distribution sorting. Manuscript submitted for journal publication. 1999. Preliminary version in *Algorithm Engineering: Proc. 3rd Workshop on Algorithm Engineering*, Number 1668 in LNCS (1999), pp. 184–198.

Sen, S. and Chatterjee, S. 2000. Towards a theory of cache-efficient algorithms (extended abstract). In *Proc. 11th ACM-SIAM SODA* (2000), pp. 829–838.

Sleator, D. D. and Tarjan, R. E. 1985. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 202–208.

Sun Microsystem. 1997. *UltraSPARC User's Manual*. Sun Microsystem.

VITTER, J. S. 2000. External memory algorithms and data structures: Dealing with MASSIVE data. To appear in *ACM Computing Surveys*.