Engineering Radix Sort for Strings*

Juha Kärkkäinen and Tommi Rantala

Department of Computer Science, University of Helsinki, Finland
{juha.karkkainen,tommi.rantala}@cs.helsinki.fi

Abstract. We describe new implementations of MSD radix sort for efficiently sorting large collections of strings. Our implementations are significantly faster than previous MSD radix sort implementations, and in fact faster than any other string sorting algorithm on several data sets. We also describe a new variant that achieves high space-efficiency at a small additional cost on runtime.

1 Introduction

Sorting is a fundamental problem in computer science that underlies a vast variety of computational tasks. When the sort keys are strings, it is possible to use any comparison based sorting algorithm but there are more efficient algorithms specialized for sorting strings. Among the best-known, simplest and fastest string sorting algorithms is the MSD (Most Significant Digit first) radix sort.

There are many possible ways of implementing the MSD radix sort. There exist extensive experimental studies on efficient implementation [2, 6] and recent new variants [7], but the possibilities have not been exhausted. We describe several new implementations, the best of which are significantly faster than any previous ones.

Radix sort and other string sorting algorithms tend to have irregular memory access patterns that are poorly suited for modern computer architectures with CPUs that are much faster than the main memory. Similar to several recent string sorting algorithms [7, 11, 12], our implementations reduce the number of slow memory accesses through better utilization of the cache memory. In addition, our algorithms reduce the *cost* of slow memory accesses by better utilization of the out-of-order execution capabilities of modern CPUs.

Some of our implementations are also very space-efficient. This is critical, for example, in several suffix array construction algorithms that rely on fast and space-efficient string sorting (see [9]).

Related Work. A seminal study on implementing MSD radix sort is by McIlroy, Bostic and McIlroy [6]. Andersson and Nilsson [2] describe more variants and provide another extensive experimental comparison. A recent, cache-efficient variant is by Ng and Kakehi [7]. Theoretical studies of radix sorting can be found in [1, 8].

* Supported in part by Academy of Finland grant 118653 (ALGODAN).

A. Amir, A. Turpin, and A. Moffat (Eds.): SPIRE 2008, LNCS 5280, pp. 3–14, 2008.

Two other fast string sorting algorithms are multikey quicksort [3] and burstsort [11, 12]. Like radix sort, both distribute strings into buckets based on a single character. However, multikey quicksort uses character comparisons to distribute the strings into just three buckets (smaller, equal and larger), while burstsort organizes the buckets into a data structure called burst trie.

There is also an extensive literature on radix sorting *integers* (see [4, 10], e.g.), but these usually involve *LSD* radix sort, and the issues are quite different.

2 Problem and Experimental Setup

We consider the problem of sorting a set of strings $R = \{s_1, s_2, \ldots, s_n\}$ over the alphabet $\Sigma = \{0, 1, \ldots, \sigma - 1\}$ into the lexicographic order. Besides n and σ, an important parameter of the problem is D, the total length of the distinguishing prefixes of the strings. The *distinguishing prefix* of a string s_i is the shortest prefix of s_i that separates it from the other strings. Thus, D is the minimum number of characters that need to be inspected, and provides a lower bound for the problem complexity. The best theoretical variants of radix sorting have time complexity $\mathcal{O}(D + \sigma)$ [8].

The experiments use the standard representation of strings in the C programming language. Thus, $\sigma = 256$ and each string is terminated with 0, which does not appear elsewhere in the strings. The task is to sort an array containing pointers to the beginning of the strings. The actual strings are stored contiguously in one array and are not moved during the sorting. Besides the sorting time, we are interested in the amount of space needed in addition to the input.

The data sets used in the experiments are described in Table 1. The initial order of the strings is random.

Table 1. Description of the test data. The datasets URL, Genome, and Unique are from [12] while Random A and Random B we have generated ourselves.

Name	n	D	Description
URL	10^7	3.1×10^8	URL addresses with the protocol name stripped
Genome	3×10^7	3×10^8	strings of length 9 over the alphabet $\{a, c, g, t\}$ from real genomic data
Unique	3×10^7	2.8×10^8	unique words collected from English documents
Random A	3×10^7	4.6×10^8	strings of single character with the length chosen uniformly at random from $[0, 30)$
Random B	3×10^7	1.2×10^8	strings of length 30 with the characters chosen uniformly at random from $[32, 255)$

The experiments were carried out on a machine with an Intel Core 2 processor model E6400 running at 2.13 GHz. The sizes of the processor's L1 and L2 caches are 32 kilobytes and two megabytes, respectively. The caches have 8-way associativity, and they use a block size of 64 bytes. The data TLB (Translation Lookaside Buffer) has two levels: DTLB0 with 16 entries (supporting loads