A study of the practical significance of word RAM algorithms for internal integer sorting

M.Sc. Thesis

Morten Nicolaj Pedersen

Department of Computer Science
University of Copenhagen, Denmark

2nd August 1999

Abstract

This thesis is submitted in partial fulfilment of the requirements of being awarded the Danish *Kandidatgrad* (*candidatus scientiarium*) at the Department of Computer Science, University of Copenhagen (DIKU). The initial study and choice of subject was made during the month of May 1998. The research was performed from October 1998 to July 1999 under the supervision of Jyrki Katajainen.

In recent years researchers have taken an interest in the *word RAM* (WRAM) machine model, which better reflects the capabilities of modern computers than the somewhat dated comparison-based model of computation. Briefly, the WRAM is a random-access machine with an unlimited number of memory cells containing w-bit unsigned integers, which supports (at least) comparison, addition, left and right shifts, and bitwise boolean instructions. This thesis is a study of the practical significance of WRAM algorithms for internal integer sorting.

The algorithms I focus on are "modern" in the sense of taking a radically different approach than classical radix-sorting algorithms, and classical comparison-based sorting algorithms. The modern sorting algorithms are generally designed as follows:

1. Use *range reduction* to reduce the problem of sorting a sequence $\mathcal{X}_b^n = \langle x_1, \dots, x_n \rangle$ of b-bit keys, $b \leq w$, to the problem of sorting (at most) n keys from a smaller range.

2. Solve the reduced problem of sorting short keys by *packing* several keys in one machine word in order to exploit the inherent bitwise *word-level parallelism* of machine instructions. Call this *packed sorting*.

3. Use the solution of the reduced problem to solve the original problem. This *range expansion* complements the range reduction.

The modern algorithms demonstrate how to sort n keys in $o(n \log n)$ time, which is an improvement over all the classical algorithms mentioned above. The modern algorithms are therefore theoretically important, but are they useful in practice? I try to answer this question.

I give a theoretical survey of WRAM algorithms for internal integer sorting, including radix-sorting algorithms. From a discussion of the practical value of the research presented, and a summary of existing empirical evaluations of it, it becomes evident, that packed sorting has no potential practical significance on the computers currently mass-produced ($w = 64$). I therefore choose to experiment with *exponential* range reduction and expansion combined with Adaptive Radixsort; an algorithm called KR Sort.

In comparison with programs, which are generally recognised as efficient, I find my careful implementation of KR Sort to perform unsatisfactorily. The comparison is made on three performance indicators, running time, number of cache misses, and number of instructions executed, since that is required to reason about the causes of the behaviour of a program. Out of necessity I synthesise a cache simulator, NEXT-GEN, from existing ones.

From the test results I conclude, that KR Sort loses due to bad memory behaviour; its implementation uses more space than the competing programs, and the memory access pattern is inherently random. This results in an excessive number of cache misses, which are expensive on a real computer. The WRAM model does not reflect this fact, and that is perhaps its biggest flaw.

Contents

1 Introduction

The topic of this thesis is an experimental study of internal integer sorting algorithms. Knuth [Knu98, Chapter 5] is a standard reference to a comprehensive survey of classical computer techniques for sorting (and searching). The algorithms I focus on are "modern" in the sense of taking a radically different approach than classical radix-sorting algorithms, and classical comparison-based sorting algorithms such as Mergesort, folklore Insertion Sort, Williams's Heapsort [Wil64], and Hoare's Quicksort [Hoa62]. The basis of the classical radix-sorting methods used in digital computers today is a mechanical tabulating and sorting machine invented by Hollerith; analogously the origin of merging is the mechanical IBM *collator*, cf., Knuth [Knu98, Section 5.5] for details.

The "modern" algorithms agree with classical radix-sorting methods on the underlying *word RAM* (WRAM) model of computation, which is not comparison based, but differentiate themselves from radix-sorting algorithms by applying *word-level parallelism* (WLP); cf., Section 1.2 for an introduction to WLP. The WLP-algorithms are mainly from the nineties, and thus modern in that respect as well. They are important theoretically, as they demonstrate how to sort n numbers in $o(n \log n)$ worst-case time, which is an improvement over all the classical algorithms mentioned above. Note, that the $o(n \log n)$ bound is derived on the WRAM.

However, as the theoretical notions of the modern algorithms are rather new, little experimental work has yet been performed with programs implementing them. I adopt a pragmatic view on computer science, and that was the basis of my choice of subject for this thesis: the modern algorithms should also be judged on their performance in practice.

The purpose of this opening chapter is to introduce the basic concepts, define more precisely the scope of the thesis, and give some incentive for it. In Section 1.1 I relate the definition of the theoretical WRAM model of computation. I also discuss the WRAM in the context of alternative models, and remark on how well it reflects real computers. In Section 1.2 I then further explain my point of departure and intent. This is also where a reading guide for the complete thesis can be found.

1.1 Models of computation

This section presents the word RAM (WRAM) model of computation in a historical context. Section 1.1.1 deals with the comparison-based model of computation. Section 1.1.2 describes the random-access machine (RAM). Section 1.1.3 presents the WRAM.

1.1.1 The comparison-based model of computation

An asymptotic lower bound on the running time of a sorting algorithm such as Insertion Sort, Mergesort, Heapsort, or Quicksort, which uses comparison of elements as the basic means of sorting, is conveniently derived in the comparison-based model of computation. To determine the pairs of elements to be compared, the model allows a sorting algorithm to use $O(1)$ free operations of the control flow or element exchanging variety per comparison. Indeed, the model requires all reordering of elements to be deducible from the comparisons decided upon. It downright forbids the use of arithmetic. In this framework, an asymptotic lower bound of the number of comparisons made by an algorithm, is regarded an asymptotic lower bound of its running time.

As explained by, for instance, Cormen *et al.* [CLR90, Theorem 9.1, p. 174], it can, by a simple decision-tree argument, be proved, that $\Omega(n \log n)$ comparisons are necessary to sort n elements in the worst case. On the other hand, Mergesort, for example, demonstrates, that $O(n \log n)$ comparisons are sufficient. Thus, the complexity of sorting n elements in the comparison-based model is $\Theta(n \log n)$.

The comparison-based model is also known as the *decision-tree model.* Note, that the model is oblivious to the exact nature of the "elements" to be sorted, as is any sorting algorithm, which can be formulated within the framework of the model. It is only required, that the comparison operator defines a *linear order*, cf., Knuth [Knu98], on the elements, and that elements can be exchanged.

The comparison-based model is acceptable as a first approximation of a theoretical model for sorting. However, a simple, practical algorithm such as Bucketsort, cf., Section 3.3.1, does not fit into its framework. Accordingly, replacements of the comparison-based model have been suggested.

Such new candidates for a model are imperative. Any branch of (natural) science utilises modelling as the stepping stone of much research. The world around us is so complex, that simplifying assumptions are inevitably made. Consider, for example, meteorology or behavioural psychology. In computer science there is a golden opportunity to apply precise models though. Engineers design and build computers, so we have exact knowledge about how "reality" behaves. It would be a sin to not exploit this in the definition of refined models.

1.1.2 The random-access machine

The *random-access machine* (RAM) model of computation was suggested by Shepherdson and Sturgis [SS63], and has been widely accepted by the research community. For example, it is the main model of computation in the textbook by Cormen *et al.* [CLR90]. The RAM is actually a class of models, as its exact details vary between different authors. I next describe the version of the RAM, that I use. Such an explicit characterisation is valuable for the discussions in the thesis, as it eliminates the RAM concept as a source of misunderstandings.

I first relate the *successor*-RAM (SRAM) as presented by Jones [Jon97, Section 7.5, pp. 118–120]. The control flow is imperative, and a program is a finite sequence of labelled instructions. At any time, the state of computation is uniquely represented by the label of the next instruction to be executed, and the exact contents of the store. The store, in turn, is unlimited in terms of both register size and number. This is a major idealisation, as will be elaborated on below. Each register contains a non-negative integer of arbitrary size, whose value is zero initially, unless the register is used for input; the input, a limited number n of integers, is assumed to be present in the first n registers (the input area), when a program starts executing on the RAM. It is also assumed, without loss of generality, that the value of n is somehow known to the program. Analogously, the output, m integers, is presented in the first m registers (the output area) upon termination, and m is somehow communicated to the "environment". The numbers n and m may be equal.

Each instruction can be executed within one time unit, and the instruction repertoire is:

0. Conditional and unconditional jumps.

1. Copying one register into another.

2. Direct and indirect addressing of registers.

3. Elementary operations on one or more registers, i.e., adding or subtracting the constant 1, comparing to 0, and so on.

The SRAM can be regarded a generic idealised model of a computer. As Jones demonstrates, the complexity class *PTIME*, usually written $\mathcal{P}$, is independent of a host of models, including the Turing machine (actually a class of models), cf., Turing [Tur36]. This statement [Jon97, Theorems 18.2.4 and 18.3.3], often phrased "robustness of polynomial-time computation", is stronger than the *Church-Turing thesis*, which merely *conjectures*, that all "reasonable" models of computation are equivalent in terms of decidability, or computability, as it is sometimes called.

The RAM differs from the simpler SRAM by allowing general purpose comparisons, and additionally a set of general purpose arithmetic instructions. By "general purpose" is signified, that the arguments can be arbitrary, i.e., not just a limited number of immediate operands like 1 and 0. For example, the contents of two arbitrary registers can be added, and the result stored in a third register. The three registers need not be distinct. The issues pertaining to the addition of the arithmetic instructions are also valid for the WRAM model, which is introduced below in Section 1.1.3. Hence, I discuss those issues there.

Mehlhorn [Meh84, Theorem 1, p. 87] has proved, that the worst-case time complexity of sorting (integers) on a RAM like the one presented above, and supporting addition, multiplication, and the "monus" operator[1], is $\Theta(n \log n)$. The result also holds (in the expected running time sense), if a "coin-tossing" instruction, i.e., non-determinism, is allowed.

However, Mehlhorn [Meh84, Theorem 2, p. 92] also demonstrates, that if division and bitwise boolean operations are allowed in unit time on operands of arbitrary size, then a RAM can sort n keys in worst-case $\Theta(n)$ deterministic time. Kirkpatrick and Reisch [KR84], and Paul and Simon [PS82], have

[1] Monus behaves like standard subtraction with the exception, that a negative result is converted to zero. This is natural when, like on the RAM, negative integers are not allowed.

similar, constructive proofs. The crux of these proofs is to abuse, that the RAM model sets no limit on the size of the non-negative integers held in registers. In the proof in [Meh84], for instance, a bit string of length $(k+2)n^2$ is repeatedly divided by the number $2^{(k+2)n}$, where k is the length of the binary representation of the maximum key.

The attentive reader will object, that bitwise boolean operations, which are needed by the $\Theta(n)$ time algorithm just mentioned, have no meaning on the RAM as presented above, because it has not been assumed, that the register contents are represented in the binary fashion. This is true, and it illustrates the point made earlier, that the RAM is class of models; different authors employ different versions of the random-access machine, which can be confusing on the first reading of new research. The important thing to notice is, that not bounding the register size significantly increases the power of the model, if each instruction is still assigned a unit cost. $\mathcal{P}$ is *not* the same for the SRAM and the RAM.

Some RAM models differentiate between registers and main memory cells in an attempt to closer reflect the architecture of real machines. The notion is, that all operands for a instruction are held in one of a small, limited number of registers, and this is also where the result of the computation is placed. Load and store instructions copy from and to main memory cells, of which there is an infinite number. This copying is usually implicit in the instruction syntax. It is important to remember the copying though, if a detailed instruction-cost scheme is to be employed, see Katajainen and Träff [KT97] for example. In their scheme, using an immediate operand, also called a literal constant, is cheaper than operating on a variable (if it is not already present in a register). The term "variable" is used here to denote a particular memory cell, which is associated with a certain entity, e.g., the ith key of the sorting problem instance.

The word RAM (WRAM), which is the topic of the next section, and the underlying model of computation in this thesis, does not distinguish between registers and main memory cells. Hence, neither do I. I therefore use "cell" below when referring to the units of the flat, i.e., one-level, memory "hierarchy" of the WRAM. As its memory model is flat, an *external* sorting algorithm cannot be analysed within the framework of the WRAM; keys can only reside in working memory, i.e., the sorting must be *internal*.

1.1.3 The word RAM

The word RAM (WRAM), a term used by Hagerup [Hag98] among others, specialises the ordinary RAM from above by specifying, that memory cells are *words* consisting of a certain number w of bits. A word is interpreted as the binary representation of an unsigned integer in the range $\{0, \ldots, 2^w - 1\}$. The least significant bit is the rightmost one, also called the first bit, or bit no. 1. Analogous remarks apply to the most significant bit. There is not complete consensus in the literature as far, as the number of memory cells is concerned. However, if a is the number of bits in an address, and n is the number of keys to be sorted, it is natural to assume:

$$w \geq a \geq \log n \tag{1}$$

i.e., the complete set of keys can be held in working memory.

The WRAM is a better model than the RAM, because the WRAM prohibits that the arguments of instructions are excessive in size; arguments are always w-bit words. Furthermore, bitwise boolean operations and shifts have meaning in the WRAM context.

Researchers have not yet agreed on, which additional instructions should be allowed for the WRAM. Hagerup [Hag98] has a good classification, which I reuse, of the extensions suggested.

The restricted instruction set The *restricted* instruction set consists of (a subset of the) instructions, which are available on a typical real computer:

- The bitwise boolean operations **not** (ones complement), **and** (conjunction), **or** (inclusive disjunction), and **xor** (exclusive disjunction). They are all realistically assumed to be of the unit-time kind.
- The zero filling left and right shifts, **left** and **right**. A unit-cost assumption for the shifts might seem unrealistic, because the displacement can be arbitrary. However, a circuit of constant, i.e., independent of w, depth, can compute all the w possible results in parallel. The size in gates of such a circuit is linear in w.
- Addition and subtraction in unit time. It is discussed below, why this is realistic.

Miltersen [Mil96] coined the term Practical RAM for a WRAM supporting addition, shifts, and bitwise boolean operations.

The AC^0 instruction set Two extensions of the restricted instruction set appear in the literature. One is the AC^0 instruction set. The class AC^0, by definition, contains the operations, that can be implemented by a polynomial (in w) sized circuit of constant depth, when unbounded fan-in to gates is allowed.

All the instructions from the restricted set belong to AC^0; they are therefore also called the "standard" AC^0 instructions. The membership of AC^0 of addition (and analogously subtraction) is perhaps not obvious. However, even though carry-lookahead addition, which is a textbook technique, see Cormen *et al.* [CLR90, Section 29.2], requires a $\Theta(\log w)$ depth circuit, a depth three circuit for addition is possible, if boolean gates are allowed to have unbounded fan-in [CLR90, Exercise 29.2-9].

As Hagerup [Hag98] points out, there is no compelling practical reason to concentrate on AC^0 instructions, as unbounded fan-in is considered expensive in microprocessor design in terms of area, power consumption and operational speed. Still, as a unit-cost measure is theoretically natural for AC^0 instructions, they are sometimes singled out as interesting.

A WRAM algorithm making use of non-standard AC^0 instructions should not be regarded as practical though. Only instructions, that are supported by the majority of machine codes, are practical.

The multiplication instruction set The second extension, which has been suggested, of the restricted instruction set is the *multiplication* instruction set, which adds only multiplication to the restricted repertoire. This extension is controversial; multiplication is *not* in AC^0, so a unit-cost measure is theoretically invalid. Cf., Beame and Håstad [BH89], who show that a multiplication circuit has depth $\Omega(\log w / \log\log w)$. As multiplication *can* be performed in $O(\log w)$ time with Wallace-tree multipliers [CLR90, Section 29.3], a logarithmic price is more to the point. As the references of this thesis *do* assign a unit cost to multiplication, but are careful to emphasise when multiplications is actually used, I do so as well.

Randomisation Some of the WRAM sorting algorithms I relate in Chapter 3 are *randomised*; they require the possibility of generating random bit patterns, or, using the proper WRAM terminology, words. I write **rand** for a unit-cost instruction, which takes no arguments, and returns a number picked uniformly at random from the range $\{0, \dots, 2^w - 1\}$. A $\Theta(1)$ time cost, i.e., several time units, is probably more realistic, because **rand** will normally be implemented by a constant number of the previously defined basic instructions. Note, that **rand** can be satisfactorily implemented with the restricted instructions set, cf., Section 7.2.

There are two types of randomised algorithms, cf., Motwani and Raghavan [MR95, Section 1.2]. For a *Las Vegas* algorithm, a fluctuation in time and space complexity between different invocations (on the same input) is the only effect of using **rand**. A *Monte Carlo* algorithm additionally might not produce the correct result. For a randomised algorithm of either type to be useful, it must behave in a certain way *with high probability*. It is not enough, that some performance can be *expected* in the statistical sense, as this does not rule out huge deviations. See Section 4.1 for a discussion of the practical implications of allowing randomness.

A Las Vegas algorithm is usually preferable over a Monte Carlo algorithm for the same problem, due to the risk of error of the latter.

Remark 1 ([MR95, Exercise 1.3]). Let a Monte Carlo algorithm A for a problem Π, with expected running time at most $T(n)$ on an instance of size n be given, and suppose that it produces a correct solution with probability at least $\gamma(n)$. Further assume that a $t(n)$ time verification method exists. Then a Las Vegas algorithm A' for Π exists, such that the expected running time is at most $\frac{T(n)+t(n)}{\gamma(n)}$. The reason of course is that $1/\rho$ is the expected value of the geometric distribution with parameter ρ.

Complexity measures The time complexity $\mathbb{T}$ of a WRAM algorithm is the number of instructions executed, and the space usage $\mathbb{S}$ is the number of working memory cells *allocated*. In this thesis, an asymptotic complexity bound is deterministic and for the worst case, unless otherwise stated.

It is not enough to count for $\mathbb{S}$ the number of different locations *accessed*. Indeed, some WRAM algorithms use *sparse* tables; they allocate huge tables of size s, where $s \gg n$, but access only $O(n)$ of the entries, cf., for instance, Section 3.5.1. It would be gravely wrong to attribute a mere linear space usage to such algorithms. Note, that if randomisation and multiplication is allowed, a sparse table can be implemented in linear space, in the "allocated" sense, by using hashing, cf., Section 2.4. With hashing, a table access still takes constant time, but in the expected sense.

Alternatively, the space consumption could be defined using the minimum value of a from Equation 1 on p. 5, for which the algorithm would still work. Then, $a = \log n + O(1)$ would correspond to linear space.

Discussion The WRAM model does not reflect the multi-level memory hierarchy of a real computer (registers, caches, main memory, paging, and finally disk storage), cf., Hennessy and Patterson [HP96]. This is perhaps its biggest flaw, as locality of reference is of monumental importance in practice. Rahman and Raman [RR99] estimate that "today", a cache hit is serviced within 1-3 CPU cycles, whereas a cache miss may require more than 60 CPU cycles. Furthermore, as the trend in hardware design is, that CPU clock rates increase faster, than main memory latencies decrease, the hit/miss gap is widening. Neither does the WRAM model take pipelining, nor parallel execution of instructions into consideration. One should not hope or aim for a model, that encompasses all these concepts, as the magnitude of the technicalities would probably cloud the general view, and weigh down the usefulness of the model.

1.2 Scope and overview

Recall, that the topic of this thesis is an experimental evaluation of the practical significance of "modern" internal integer sorting algorithms, i.e., algorithms employing *word-level parallelism* (WLP), and derived in the WRAM model of computation. At the core of WLP lies the technique of "packing" several keys in one machine word in order to use the inherent bitwise parallelism of instructions in somewhat ingenious ways. Cf., Section 2.3 for a precise definition of packing, and see Figure 5 on p. 26 for an illustration of, how several keys can be compared with a constant number of instructions when packed in one word.

Before getting to the real matter of interest, I first, in Chapter 2, touch on preliminaries such as notation and basic concepts.

Chapter 3 is a survey of non-comparison-based sorting algorithms for the WRAM from the literature. This includes both classical radix-sorting algorithms, a recent radix-sorting algorithm, and the modern algorithms that exploit WLP. My focus is internal sorting of single-precision integers, i.e., b-bit integers, where $b \leq w$. For the methods I present though, I emphasise, if they can be extended to multiple-precision integers. I do not deal with "parallel" algorithms in the traditional sense of the word, i.e., algorithms demanding several processors, cf., Albers and Hagerup [AH97], and Andersson *et al.* [AHNR98]. The majority of the new algorithms are, in my opinion, much harder to understand than the classical ones. Furthermore, for many of the new ideas it is not immediately obvious, to me at least, how they are more than pure theoretical notions. The question is, which ones of the new ideas can be faithfully programmed at all such that the theoretical resource bounds still hold?

In Chapter 4 I try to answer this question. I discuss the properties an algorithm should have, and not have, in order to be "practical" for implementation. Based on the selection criteria resulting from that discussion, I choose the modern techniques, which seem to be most valuable in terms of practicality of implementation and potential for efficiency.

Chapter 5 details the design of my empirical evaluation of those promising techniques. It explains which techniques that have been implemented, and the method used to assess the performance of the programs. The chapter starts with a summary of existing experimental work with WLP. None of those studies are conclusive as far as the practical value of WLP is concerned. Nor should this be expected of them. An exhaustive exploration is a giant effort, which many researchers must take part in, if it is to be completed. This thesis should be regarded such a stepping stone.

Chapter 6 concerns the actual, careful programming of the techniques, I have decided to evaluate.

The experiments are documented in Chapter 7. The tools used for measuring performance indicators are described along with the suite of test instance, and the testing framework. Plots of the results and a discussion of them are also presented in that chapter.

Chapter 8 is an overall conclusion.

Each chapter begins with a more detailed outline of its contents.

2 Preliminaries

In this chapter I introduce some notation and a few concepts and data structures, which are used throughout the thesis. Section 2.1 deals with technicalities. The *trie* data structure is discussed in Section 2.2. Section 2.3 introduces the notion of packing several "short" integers in one WRAM word. Section 2.4 is a treatment of hashing.

2.1 Technicalities

At this point I define some general notation; some context-specific notation is introduced along the way. The same symbol might be used at different places in the thesis to denote separate entities, because I use the pedagogical technique, of using symbols as mnemonics. Ambiguity will never be allowed, the exact meaning of a symbol will always be clear from the context.

The instance size is denoted by n, and $\mathcal{N} = \{1, \ldots, n\}$. By "instance size" I mean, for example, the number of keys to be sorted, or the number of elements to be organized in a data structure. I adhere to the customary convention of assuming integer divisions to have no remainder, writing n/d instead of $\lceil n/d \rceil$ or $\lfloor n/d \rfloor$. This is no loss of generality, and is merely done to better communicate the overall design of various algorithms, focusing on the ideas and not the mathematics. $\mathcal{X}_b^i = \langle x_1, \ldots, x_i \rangle$ denotes a sequence of i unsigned b-bit integers. The term "sequence" does not disallow random access, which is guaranteed on the WRAM. $\log n$ is the base two logarithm of n, and I write $\log^a n$ instead of the less ambiguous, but more cumbersome $(\log n)^a$. Let $\mathrm{kb} = 2^{10}\mathrm{b}$, and $\mathrm{Mb} = 2^{20}\mathrm{b}$. I use the term "key" to denote an integer of *at most* w bits.

2.2 Tries

Tries are closely tied to algorithms for searching, which are not comparison based, cf., Knuth [Knu98, Section 6.3]. In fact, the name alludes to the word "re*trie*val". The significance of the trie concept for this thesis lies in its strong connection with radix sorting. Nilsson [Nil96] elaborates at great length on the connection, and below I follow his presentation of tries. First a definition:

Definition 2.1. Let Σ be a finite alphabet of size $m = 2^\ell$, and suppose that a set $\mathcal{S}_\ell^n = \{s_1, \ldots, s_n\}$ of n strings over Σ is given. No string s_i may be empty. An *alphabetic trie* representing $\mathcal{S}_\ell^n$ is a tree $\mathcal{T}$ with the following properties:

1. If $n = 0$, $\mathcal{T}$ is the empty tree.
2. When $n = 1$, $\mathcal{T}$ consists of a single node representing s_1.
3. For $n > 1$, the root of $\mathcal{T}$ is an m-ary node. For each character $c \in \Sigma$, the root has a child, which is an alphabetic trie representing the suffixes of the strings $s_i \in \mathcal{S}_\ell^n$ that start with c.

Here "suffix" is taken to mean all but the first character. This recursive definition is illustrated in Figure 1, which shows a 26-ary trie over some English words. The set $\mathcal{S}_\ell^n$ is assumed to be *prefix free*, such that s_i is not a prefix of s_j for $i \neq j$. Specifically, no two strings are the same. Children of a common parent are *siblings*.

Figure 1: An alphabetic trie. Nilsson [Nil96, Figure 1-2].

Let v be the leaf of $\mathcal{T}$ corresponding to the string s_i. Each node on the path from the root to v (both endpoints included) corresponds to a *distinguishing character* of s_i. These are the characters of s_i it is absolutely necessary to inspect, in order to determine the rank of s_i in $\mathcal{S}_\ell^n$. Together they make up the *distinguishing prefix* of s_i. The length of the distinguishing prefix is denoted by c_i, and $C = \sum_{i \in N} c_i$, $\bar{C} = C/n$, and $C_{\max} = \max_{i \in N} c_i$.

For the purposes of this thesis, the case $m = 2$ is of special interest, because a WRAM word is a binary string of length w. When $m = 2$, it is natural to use the terminology distinguishing *bits* of s_i, and denote by b_i the length in bits of the distinguishing prefix of s_i. The use of b_i instead of c_i is supposed to emphasise *b*its as opposed to *c*haracters. The definitions of B, $\bar{B}$, and $B_{\max}$ are analogous.

When $m = 2$, the binary strings are sometimes viewed as originating from a (virtual) alphabet $\Sigma' = \{0, \ldots, m' - 1\}$ of size $m' = 2^c$. The (virtual) characters comprising Σ' are bit patterns of length c. The m'-ary alphabetic trie which corresponds to regarding the binary strings in the set $\mathcal{S}_\ell^n$ as strings over Σ', is called a *c-digit* trie, and is denoted by $\mathcal{T}_c'$. For binary strings, c_i and b_i (plus the associated quantities) both have meaning: the former corresponds to c-bit characters, and the latter to bits. When I use "c" for the length of a bit pattern in this thesis, the pattern is a *chunk* of bits from a larger binary string.

It is common to *path compress* an alphabetic trie by removing all internal nodes with only one non-empty child. The resulting path-compressed trie (PCT) is guaranteed to have at most $2n - 1$ nodes: if $m = 2$, at most $n - 1$ binary internal nodes are needed to connect all the n leaves (indirectly) to the root, and if $m > 2$, fewer internal nodes may be sufficient. I write $\mathcal{T}_{PC}$

for the path-compressed version of $\mathcal{T}$.

2.3 Packed representation of short keys

The notation introduced in this section is borrowed from an article by Albers and Hagerup [AH97]. Given k and d obeying the inequality $k(d+1) \leq w$, one can view a WRAM word as composed of k *fields* of $d+1$ bits each[2] (with some bits to spare unless $k(d+1) = w$). The *packed* (k, d)*-representation* of a sequence $\mathcal{X}_d^k = \langle x_1, \ldots, x_k \rangle$ of d-bit integers is illustrated in Figure 2. The figure shows the x_i packed together in a single word $X_{k,d}$. Note, that the least significant field holds the first number, and that field and number indexes agree in general. I use the notation $X_{k,d} = [0 : x_k| \cdots |0 : x_1]$ for packing. The parameters k and d will sometimes be omitted, if they can be deduced from the context. A sequence $\mathcal{X}_d^n = \langle x_1, \ldots, x_n \rangle$ of d-bit integers, $n > k$, is packed in n/k words $X^1, \ldots, X^{n/k}$, where $X^i = [0 : x_{ki}| \cdots |0 : x_{k(i-1)+1}]$. *Reversing* a *packed word* X means doing so with the order of the fields it contains.

Figure 2: Packed (k, d)-representation of the sequence $\mathcal{X}_d^k = \langle x_1, \ldots, x_k \rangle$ in a WRAM word $X_{k,d}$.

The extra bit of a field is called the *test bit*. In Figure 2 all k test bits are zero. The "modern" WRAM algorithms, which use the packed representation in order to apply word-level parallelism, maintain the invariant: if a packed word contains a non-zero test-bit value, the word is currently participating in a yet unfinished series of computations. In such a series of calculations the test bits are used as *flags* to distinguish between fields of interest. How, will become apparent below, cf., Section 3.4. When I use "d" for the bit length of integers in this thesis, the integers should be regarded "short", i.e., they can be (k, d)-packed.

Some special constants are convenient. One is the *test-bit mask* $T_{k,d} = [1 : \bar{0}| \cdots |1 : \bar{0}]$, where "$\bar{0}$" denotes a vector of length d of reset bits. The constant $A_{k,d} = [0 : k-1| \cdots |0 : 1|0 : 0]$ makes explicit the assumption, that $d \geq \log k$. Both $T_{k,d}$ and $A_{k,d}$ can be computed with the restricted instruction set in $O(\log k)$ time by a "repeated doubling" approach: a constant number of instructions suffices to double the number of fields currently occupied by their final value, cf., Hagerup [Hag98]. The constants are called *native*, because it is common to assume, that they are available for free, i.e., they can be loaded in constant time with special instructions. Hence, the

[2] Perhaps denoting the number of *f*ields by f would be more suggestive, but f is ordinarily used to denote a function.

$O(\log k)$ term is ignored when tallying up the asymptotic running time, although theoretically it could dominate. Native constants do introduce some *weak* non-uniformity, a phrase coined by Ben-Amram and Galil [BAG97], into algorithms, as they are nothing but "magic numbers".

2.4 Hashing

This section briefly introduces an incarnation of *hashing*, which is relevant for later developments. See, for example, Knuth [Knu98, Section 6.4] for a thorough and general treatment of classical hashing.

The *dictionary problem* concerns representing a data structure for a subset $\mathcal{S}$ of a totally ordered universe $\mathcal{U} = \{0, \ldots, U-1\}$ of keys. I focus on the *dynamic* dictionary problem; the case where $\mathcal{S}$ is initially empty, and must be represented efficiently under the updates $\textsc{Insert}(x)$ and $\textsc{Delete}(x)$, and the query $\textsc{Find}(x)$, where $x \in \mathcal{U}$. The symbol for the subset $\mathcal{S}$ is a mnemonic stressing, that it is sparse, meaning that $|\mathcal{S}| \ll U$. Thus, it is unsatisfactory to simply utilise a table of size U of markers in a straightforward manner to solve the problem.

A *hash table* solves the dynamic dictionary problem in $O(|\mathcal{S}|)$ space, but with a time-complexity tradeoff. It entails a *hash function* $h : \mathcal{U} \to \mathcal{T}$, where $\mathcal{T} = \{0, \ldots, T-1\}$ and $T = O(|\mathcal{S}|)$. The idea is, to use a *bucket table* of size T. An entry in the bucket table, also called a bucket, is an appropriate search structure, i.e., a list. The hash-table primitives update or search the bucket with index $h(x)$ when given x as argument. As $T \ll U$, since $T = O(|\mathcal{S}|)$, the existence of pairs $(x, y) \in \mathcal{U}^2$ such that $h(x) = h(y)$ is to be expected. The number of such *collisions* is a central issue in choosing the hash function.

Dietzfelbinger *et al.* [DHKP97] are responsible for an attractive solution when $U = 2^k$ for some $k > 0$, which I proceed to describe. Let $\ell \in \{1, \ldots, k\}$. Define the hash function $h_a : \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^\ell - 1\}$ using integer division and remainder:

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell}, \quad 0 \le x < 2^k \tag{2}$$

Let $\mathcal{H}_{k,\ell} = \{h_a | 0 < a < 2^k \text{ and } a \text{ is odd}\}$. It follows that:

Lemma 2.1 ([DHKP97, Lemmas 2.4 and 2.6]). *Let k and ℓ be integers with $1 \le \ell \le k$. Choose $h_a \in \mathcal{H}_{k,\ell}$ uniformly at random. If $x, y \in \{0, \ldots, 2^k - 1\}$ are distinct:*

$$\Pr[h_a(x) = h_a(y)] \le \frac{1}{2^{\ell-1}} \tag{3}$$

If $\mathcal{S}$ is a subset of $\{0, \ldots, 2^k - 1\}$:

$$\Pr[h \textit{ is 1-1 on } \mathcal{S}] \ge 1 - \frac{|\mathcal{S}|^2}{2^\ell} \tag{4}$$

The cardinal virtue of the hash functions in $\mathcal{H}_{k,\ell}$ is, that they are fairly cheap to evaluate. There is of course the potentially expensive multiplication ax, but the remaining power of two integer arithmetic can be realised with the instructions **and** and **right** on the WRAM, provided that $k \leq w$. Equation 3 guarantees, that inserted keys can be expected to be evenly distributed among the buckets. If ℓ is chosen, such that $2^\ell = O(1) \cdot |\mathcal{S}|$, all hash-table primitives can be expected to run in $O(1)$ time.

3 Integer sorting algorithms for the word RAM

In this chapter I first define, in Section 3.1, in no uncertain terms the sorting problem I consider. The definition is accompanied by some reflections supporting it. Section 3.2 describes how WRAM algorithms can be partitioned into two groups on their asymptotic complexity. It also contains a characterisation of the general design of a "modern" WRAM sorting algorithm. A reading guide for the rest of this chapter can be found there as well.

3.1 The conservative value integer-sorting problem

The *conservative value integer-sorting problem* (CVISP) is defined as:

Definition 3.1. Given a sequence $\mathcal{X}_b^n = \langle x_1, \ldots, x_n \rangle$ of n *unsigned* b-bit integers, or *keys*, the CVISP is to permute the keys into non-decreasing order in the WRAM model of computation.

I use the notation CVISP(n, b) below. Note the deliberate use of the word "key". Thorough comments on the definition are needed. That is the subject of the rest of this section.

In the survey of existing research below, cf., Sections 3.3–3.6, I follow the convention of adopting the restricted instruction set, when nothing else is explicitly mentioned. Thus, I take care to emphasise, when multiplication is used.

The restriction to unsigned integers is no loss of generality. The transformation of an instance, in which negative keys occur, to one where it is not the case, is a simple addition of $2^{\frac{b}{2}}$ to all keys. It is nice to not have to worry about the bit-pattern representation of negative numbers, i.e., is it *signed magnitude*, *one's* or *two's complement*, or *excess* 2^{b-1}? Cf., Tanenbaum [Tan90, Section A.4].

"Conservative" indicates that instructions are prohibited from operating on numbers, which cannot be represented by w bits. This is guaranteed in the definition by insisting on the WRAM, and is believed to significantly increase the time complexity of sorting. It forbids "stretching" the unit-cost assumption for instructions. As already noted in Section 1.1.2, such stretching is central in proofs from the literature of how to sort deterministically, but non-conservatively, in linear time on the RAM. Currently, sorting in $\Theta(n)$ (expected) time on the WRAM is only known to be possible, due to Andersson *et al.* [AHNR98], under the assumption $w \geq (\log n)^{2+\epsilon}$, and with the aid of multiplication and randomisation, cf., Signature Sort in Section 3.6.2. Thorup [Tho97] complements this result with an algorithm, which runs in $O(n \log \log n)$ expected time on the Practical RAM, cf., Cluster Sort in Section 3.6.2.

Remark 2. A *stable* sorting algorithm maintains the relative order of non-distinct keys. An *in-place* algorithm uses only a constant amount of auxiliary memory cells in addition to the space needed to store the keys.

There is a difference between *rank* and *value* sorting. The former calls for computing the rank of each key; the rank of a key x' is defined as the cardinality of the set $\{i \in \mathcal{N} : x_i \leq x'\}$. To value sort is to permute the keys into non-decreasing order, as requested in Definition 3.1.

Whether rank sorting is required or not, is closely coupled with the presence or absence of *satellite data*. If such data exists, for instance, if the integers to be sorted are database-record keys, reordering the keys imply reordering the complete records. Thus, value sorting is not satisfactory in the presence of satellite data; rank sorting is preferable. Alternatively, an indirect value sorting, that reorders records implicitly by reordering references to them could be applied. Note, that the question of stability is void for value-sorting algorithms in the absence of satellite data, in which case there is no way to distinguish keys of equal value.

Assume for this paragraph, that all keys are distinct. Of course, a rank-sorting algorithm can be transformed into a value sorter: the position in the output of a key is its rank, and the WRAM provides random access to the output area. The reverse is true as well; the capability to value sort implies a rank-sorting algorithm. Assume that $n = 2^p$. The trick is to label each key by means of its index: replace x_i with $x'_i = nx_i + (i-1)$ by shifting p positions to the left and adding $i-1$. This makes each key unique, but two words are now required to represent each x'_i. The idea then is to first value sort the x'_i on the low-order words, and subsequently use a stable Bucketsort, see Section 3.3.1, to sort the result on the high-order words. The latter action takes $\Theta(n)$ additional time and space, as the input to Bucketsort is n numbers of $p = \log n$ bits each. Afterwards, key no. r in the output from Bucketsort contains in its p low-order bits (of the low-order word) the index (minus one) of the key with rank r. The assumption $n = 2^p$ is no loss of generality. The demonstration above of the equivalence between rank and value can be found in a paper by Andersson *et al.* [AHNR98]. The equivalence holds even though keys are not distinct, as can be verified with a little effort.

WRAM algorithms for the CVISP are applicable to other types of data than integers. The IEEE 754 floating-point standard [Tan90, Section B.2] was designed such that larger reals are represented by bit patterns corresponding to larger integer values. A set of uniformly distributed reals does not map to a set of uniformly distributed bit patterns though. Most character sets, e.g., ASCII, code the sequence of letters "a, b, ... , z, A, B, ... , Z" as a strictly increasing sequence of integer values. Thus, integers, reals and letters can readily be sorted, but what about strings and multi-precision numbers? Radix-sorting algorithms, cf., Section 3.3, are feasible for, indeed designed for, such data, whereas the applicability of the "modern" WRAM algorithms using word-level parallelism is not obvious, as will become apparent. Analogous remarks apply to records, which are multi-word structures.

3.2 Classification and characterisation

WRAM algorithms can be split into two groups on the basis of their asymptotic complexities:

1. An algorithm belongs to the first group, if its running time or space

demands depend on w. Fredman and Willard [FW94] call such algorithms *trans-dichotomous*. The term "first-generation" has also been used, as the earliest examples appeared before any algorithm of the second type. Traditional radix-sorting algorithms, cf., Section 3.3.2, fall in this catagory, as does KR Sort, cf., Section 3.6.1.

2. Naturally then, the second group encompasses the algorithms, whose asymptotic bounds depend on n alone. Fredman and Willard [FW94] dub those *strongly trans-dichotomous*, and were also the first to develop a $o(n \log n)$ sorting algorithm of this type, cf., Section 3.7. As algorithms in this group are from the nineties, they have been called "second-generation". I have used the term "modern" so far, and will continue to do so.

The two groups complement each other. For large w, an algorithm of the second type may be more efficient, whereas it might pay to switch to one of the first type, as w decreases. In my opinion, it makes sense to consider w to be constant for any fixed machine. I therefore recommend substituting the actual size of keys in bits b, for w, when interpreting the "switching" statement. The key size b, which is assumed to be less than or equal to w, *can* be regarded mutable. Indeed, the "modern" sorting algorithms are characterized by the fact, that they operate in three consecutive stages:

Range reduction The original problem CVISP(n, b) is reduced to sorting at most n *short* keys from a smaller range $\{0, \ldots, R-1\}$, where $R \ll 2^b$. Frequently R is a power of two as well, e.g., $R = 2^d$, with $d \leq b$.

Packed sorting Those short keys are sorted in a *packed* fashion by using *word-level parallelism* (WLP), which is enabled by using the packed (k, d)-representation, cf., Section 2.3.

Range expansion The sorted sequence of short keys thus produced is utilised to construct the final sorted output of the orginal b-bit keys $\mathcal{X}_b^n = \langle x_1, \ldots, x_n \rangle$.

The first- and last-mentioned steps are mutually dependent. The middle one can be considered a plug-in, that can freely be changed, although it can be an advantage to waive this property for reasons of efficiency of implementation, cf., Section 6.2. The reduction/expansion method justifies regarding the key size b as mutable, as promised.

All the WLP-sorting algorithms presented in Section 3.6.2, follow such a blueprint. Furthermore, they are strongly trans-dichotomous. They achieve this by decreasing d as n increases.

The rest of this chapter is a survey of the body of research papers published in recent years on WRAM algorithms for the conservative value integer-sorting problem (CVISP). Section 3.3 deals with radix-sorting algorithms; both classical, endian ones, and Forward Radixsort, which was concocted recently by Andersson and Nilsson [AN94b]. Section 3.4 concerns packed sorting, and Section 3.5 is devoted to range reduction and expansion. Finally, Section 3.6 summarises how researchers have combined the techniques presented in its two preceeding sections to construct WLP algorithms in accordance with the three-step design above.

3.3 Radix sorting

Radix-sorting methods were among the first algorithms developed, that did not fit into the comparison-based model of computation. The presentation in this section concerns radix sorting of n strings $\mathcal{S}_\ell^n = \{s_1, \ldots, s_n\}$ of ℓ-bit characters from an alphabet Σ of size $m = 2^\ell$; it thus demonstrates the applicability of radix-sorting algorithms to strings in general, when the alphabet size m is a power of two. However, as the focus of this thesis is integer sorting, the reader should recall the notion, which was introduced in Section 2.2, of viewing the binary representation of a b-bit number as a string of length $k = b/c$ of c-bit (virtual) characters from an alphabet Σ' of size $m' = 2^c$.

Andersson and Nilsson [AN94b] have recently invented Forward Radixsort, which is also described in Nilsson's Ph.D. thesis [Nil96]. I relate this algorithm in Section 3.3.3, after summarising Nilsson's survey of classical radix-sorting algorithms in Section 3.3.2. See also Knuth's book [Knu98] for such a survey. To begin with though, I explain Bucketsort. It is generally assumed that $\ell = \Theta(w)$, such that a character can be inspected in constant time.

3.3.1 Bucketsort

Central to radix-sorting algorithms is *bucketing* on ℓ-bit characters. I describe the bucketing concept by relating Bucketsort. The algorithm sorts a sequence $\mathcal{X}_\ell^n = \langle x_1, \ldots, x_n \rangle$ of ℓ-bit keys in $\Theta(n+m)$ time and space, where $m = 2^\ell$. Bucketsort revolves around a table of size m. Each table entry is a *first-in-first-out* (FIFO) data structure referred to as a *bucket*. The FIFO property is trivially seen to be maintainable, for instance, by using a list to represent a bucket, within a $\Theta(1)$ time bound on insertions into, and extractions from, buckets.

The sorting is done in three steps:

1. The bucket table is initialised in $\Theta(m)$ time.

2. In $\Theta(n)$ total time each key x_i is inserted into the bucket corresponding to the ℓ-bit value of the key; x_i is used to index the bucket table. In machine code terms, this is offset addressing.

3. In $\Theta(m)$ time the buckets are traversed in ascending order. For each bucket, all keys are removed in FIFO order and written to the output area, which hence is accessed sequentially only. It takes $\Theta(n)$ time to process the contents of all buckets.

If satellite data exists, references to the records containing the keys are of course placed in the buckets instead of the keys x_i themselves. In this case it is worth noticing, that Bucketsort as presented is stable, due to the FIFO property of buckets.

In fact, a counter, instead of a data structure, suffices to represent a bucket, which reduces insertion and extraction to increment and decrement of a counter. With this modification it is sensible to call the bucket table the *counter* table. "Radix table" is sometimes used as a collective name. After Step 2 the counter table can be interpreted as a histogram of the sequence of keys $\mathcal{X}_\ell^n = \langle x_1, \dots, x_n \rangle$. Bucket sorting is therefore also known as *sorting by distribution*. The histogram can be used to organise the keys, or the references to the records containing them if satellite data exists, into sorted order. This permutation can be effected in-place, but unstably, in linear time $\Theta(n+m)$. I have used this technique in my implementation, cf., Section 6.1, of an algorithm modelled after Adaptive Radixsort, see the next section. If auxiliary space for a copy of the keys/references is available, the permutation can be effected in a stable fashion. Because of Equation 1 on p. 5, two references can be exchanged in constant time on the WRAM. The references are stored in a table.

In the context of radix sorting strings of ℓ-bit characters, Bucketsort is applied repeatedly to sort the strings on a character *position* p, with p either increasing or decreasing as the radix-sorting algorithm proceeds. Note, that character positions are numbered analogously to bit positions; the rightmost one is no. 1. Do not confuse the first character *position* with the first *character*, which is the leftmost one. For each p, the characters in positions different from p can be regarded satellite data during the Bucketsort. Hence, when, as in the general case, the strings, i.e., arbitrary English words coded according to the ASCII standard, are too large (in either length or character size, or both), for each string to fit into a constant number of machine words, it is normal to sort the strings indirectly by identifying each one with a reference to it, and permuting the references (with Bucketsort) during the radix sort. In the special case, where a "string" is an integer from $\{0, \dots, 2^w - 1\}$, strings can obviously be exchanged in constant time, and indirection is unnecessary; the integer can be interpreted as a reference to itself. The presentation below of radix sorting adopts the general view of permuting references. As they correspond to strings, I take the liberty of not consistently distinguishing between these two types of entities.

3.3.2 Classical radix-sorting algorithms

LSD Radixsort *Least significant digit* (LSD) Radixsort sorts n strings of equal length k by in turn applying a stable version of Bucketsort to character position $p = 1, \dots, k$ (in that order). After the ith application of Bucketsort, the strings are sorted according to their i rightmost characters. For example, 64-bit integers ($b = 64$) can be sorted by eight phases ($k = 8$) of bucketing on 8-bit characters ($c = 8$). The drawback of LSD Radixsort is, that it unconditionally inspects all the characters of the strings; the time complexity is $\Theta(k(n + m))$, where m is the alphabet size. The space complexity is $\Theta(n + m)$.

MSD Radixsort *Most significant digit* (MSD) Radixsort starts from the opposite/leftmost end, and inspects at most C, cf., Section 2.2, characters, i.e., at most the distinguishing characters. The algorithm does not require the strings to have equal length, but for convenience of presentation I assume that all the strings have length k.

The algorithm works as follows:

1. Use Bucketsort to sort the n strings on character position k, i.e., the leftmost one. This permutation of (the references to) the strings partitions them into m (the alphabet size) *groups* arranged in ascending order. A group is a maximal subset of strings from $\mathcal{S}_\ell^n$ which agree on character position k. The n references are now organized such that the character value of position k of the strings they point to is non-decreasing. To each group thus corresponds a distinct subrange of the references, and a group is conveniently represented by the indexes of the endpoints of this subrange. The endpoints can be calculated from the histogram used in Bucketsort.

2. In ascending order use recursion separately on each group containing more than one string, now regarding $k-1$ as the leftmost character position. This calls for extending the representation of a group with the position of the "leftmost" character of the strings in it.

It is usual to implement this recursive recipe iteratively by means of an explicit stack of remaining groups, using the same (allocated once only) radix table for each recursive invocation of Bucketsort. Repeatedly, a parent group is popped from the stack, and after partitioning it with Bucketsort, the resulting children groups containing more than one string are pushed onto the stack. In the context of radix sorting a table (of references to) strings, the order of the pushes does not matter, as the order of the recursive sortings of groups/subranges is irrelevant.

Each string must be inspected once, and furthermore it is accessed once per distinguishing character of it. Taken together, this implies a $\Theta(n+C)$ contribution to the running time. Initialisation (resetting) and traversal of the radix table takes place at each non-leaf node of the m-ary trie corresponding to the complete set of n strings $\mathcal{S}_\ell^n$. This trie can be as large as $\Theta(C)$, hence the overall cost of using the radix table is $O(mC)$. Therefore, the total time complexity is $O(n+mC)$. Note, that the trie is not constructed by MSD Radixsort; the concept is merely used in the analysis. The space consumption of MSD Radixsort is $\Theta(n+m)$, since it inherits such a space demand from Bucketsort, and the size of the stack is $O(n)$.

For small groups of size n', where $n' \ll m$, the use of the radix table is expensive. Therefore a *hybrid* algorithm, which switches to a comparison-based algorithm, for instance, Insertion Sort, on groups smaller than some threshold n_0, can advantageously be used. In addition to eliminating some of the radix table traversals, this optimisation reduces the number of stack records to $O(n/n_0)$. The optimisation does not decrease the asymptotic time bound, and if the insertion-sorting algorithm is careful to compare character positions and not the complete strings, the optimisation does not increase the bound either. The hybrid concept is thus a practical matter. Knuth [Knu98, p. 380] recommends Insertion Sort with $n_0 \approx 25$.

Adaptive Radixsort This algorithm is a modification of MSD Radixsort. Even though the characters are ℓ bits long, the partitioning of a group of size g is done on $\log g$ bits of the underlying coding of the strings, which implies an associated radix table of size $2^{\log g} = g$. The point is, that the cost of using (resetting and traversing) a radix table of size g for partitioning a group of size g, matches the cost of reading distinguishing bits, which is necessary for the partitioning, from each of the strings in the group. The time complexity of Adaptive Radixsort is therefore $\Theta(n+B)$, where B is the number of distinguishing bits. However, if at least ℓ bits are inspected at a time, the theoretical performance is at least as good as for MSD Radixsort. Adaptive Radixsort uses only $\Theta(n)$ space. Tamminen [Tam83] has shown that Adaptive Radixsort runs in $\Theta(n)$ expected time on a large class of input distributions, including uniformly distributed data. The practical paradigm of switching to Insertion Sort is valuable here as well.

3.3.3 Forward Radixsort

Forward Radixsort combines the advantages of LSD and MSD Radixsort, and suffers from none of the drawbacks of either of those two methods; the algorithm only traverses the bucket table once per character position, and it inspects only distinguishing characters. Below, lists are assumed to be doubly linked.

Basic Forward Radixsort I first describe the *basic* version of Forward Radixsort. The algorithm is list oriented; it works on a list L of (references to) the strings. It partitions *groups* in *passes*. A group contains a sublist of L. To begin with, there is only one group, and it contains all the strings $\mathcal{S}_\ell^n = \{s_1, \ldots, s_n\}$ of ℓ-bit characters. After the jth pass the strings have been partitioned such that exactly all strings with the same prefix of length j belong to the same group, and all groups are maintained in a group list G in ascending order of these prefixes. A group has a number associated with it, which indicates the rank in $\mathcal{S}_\ell^n$ of the lexicographically smallest string in the group. Call this number the *label* of the group. Note that by definition the groups are also maintained in sorted order of labels. A group is *finished* after pass j, if it contains only one string, or if all strings in it are at most j characters long. The "switching" paradigm is useful in practice to finish directly groups of size less than some appropriate threshold n_0. The unfinished groups are maintained in sorted order in an additional, auxiliary group list U.

The algorithm employs a *first-in-first-out* (FIFO) *bucket* table, cf., Section 3.3.1, of size $m = 2^\ell$. For $j = 1, \ldots, C_{\max}$, where $C_{\max}$ is the length of the longest distinguishing prefix, the jth pass of basic Forward Radixsort works as follows:

1. Traverse the unfinished groups in ascending order, and bucket each unfinished string s_i on its jth character from the left. Insert with each string the *tag* of its group.

2. Go through the buckets in ascending order, and remove from each bucket in FIFO order the strings in it, and put them back in their groups by means of the tags.

3. Process each group separately in ascending order of group labels. If the rth string in a group labelled g differs in its jth leftmost character from its predecessor in the group, then split the group in two immediately before the rth string. This split can be performed in constant time, since lists are doubly linked. The "bottom" half inherits the label g. The "top" half is assigned the new label $g + r$ in accordance with the rank condition on labels, and the splitting/processing recurs on this half.

Furthermore, the algorithm checks, before each pass j, if all the characters to be inspected during the pass are equal. In this case, the pass is skipped, and j is increased. See Figure 3 on p. 22 for a helpful illustration. The thing to note is, that a tag can be shared between several strings.

Basic Forward Radixsort runs in $O(C + n + m \cdot C_{\max})$ time: it inspects only distinguishing characters, and uses a size m bucket table for each of the $C_{\max}$ passes. The running time is also bounded by $O(C+n+m^2)$: first, when there are more than m unfinished strings in a pass, the cost of using the bucket table matches the cost of reading characters from the strings; and second, when less then m strings remain in the unfinished groups, there can be at most m passes left. This implies:

Theorem 3.1 ([Nil96, Theorem 2.1]). *A sequence of n strings from an alphabet of size m can be sorted on the WRAM in $O(C+n+m\cdot\min(m, C_{\max}))$ time with the restricted instruction set, where C is the total number of distinguishing characters, and $C_{\max}$ is the length of the longest distinguishing prefix.*

Complete Forward Radixsort Theorem 3.1 can be improved upon by making sure not to visit empty buckets. That is the subject of the rest of this section. The *complete* version of Forward Radixsort of strings $\mathcal{S}_\ell^n = \{s_1, \ldots, s_n\}$ in brief description works as follows:

1. Create an empty list P.

2. Basic Forward Radixsort with a modification, which I proceed to describe, is run on the strings. Before pass j, a list B of bucket indexes is initialised. The index of a bucket is inserted in B the first time (only) the bucket is used in pass j. Only used buckets, and not the complete bucket table, are initialised, cf., Section 3.5.1, for an explanation of how this is possible. B is used to visit only non-empty buckets, although not in sorted order, when putting strings back in their respective groups. When a group is subsequently split on a character value c in pass j, the pair (j, c) is inserted in P. With the modification, the invariant that groups are kept in ascending order is not maintained, and group labels do not obey the rank condition.

Figure 3: A pass of Forward Radixsort. [Nil96, Figures 2-3–2-5].

3. First, sort the list P of pairs (j, c) on their second coordinate in time $\mathbb{T}_P(n, m)$ and space $\mathbb{S}_P(n, m)$, see below. Then, use Bucketsort to sort the result stably on the first coordinate of the pairs in $\Theta(n)$ time, cf., Nilsson [Nil96, p. 23], for details, and for an explanation of, how the size of P can be reduced to $O(n)$ before the two sortings of P.

4. Another variant of Basic Forward Radixsort is run on $\mathcal{S}_\ell^n = \{s_1, \ldots, s_n\}$. This time, in pass j, there is no list of activated buckets B, but the trick of not initialising the complete bucket table is still used. The sorted list P is used in all passes to visit *only* non-empty buckets, and *in alphabetical order*. In all, P is traversed once only.

As the cost of using (traversing and resetting in whole) the bucket table has been eliminated, Complete Forward Radixsort proves the following:

Theorem 3.2 ([Nil96, Theorem 2.3]). *A sequence of n strings from an alphabet of size $m = 2^\ell, \ell \leq a$, can be sorted in*

$$O(C + n) + \mathbb{T}_P(n, m) \tag{5}$$

time with the restricted instruction set on the WRAM, where $\mathbb{T}_P(n, m)$ is the time to sort n integers from the set $\{0, \ldots, m - 1\}$.

The implied bound $2^\ell \leq 2^a$ stems from the fact, that all buckets must be addressable, cf., Equation 1 on p. 5. The space complexity is: $\Theta(n + m) + \mathbb{S}_P(n, m)$. I use "$P$" as subscript to signify the correspondence with sorting the list P on the second-coordinate character values, which are ℓ bits wide. See Section 3.5.2 for a discussion of the impact of this theorem on sorting integers. In this thesis "Forward Radixsort" means the complete version, unless qualified with "Basic".

3.4 Packed sorting

The bedrock of the packed sorting algorithm, I present in this section, is a comparison-based method discovered by Batcher [Bat68] for sorting *bitonic* sequences. It is presented admirably by Cormen *et al.* [CLR90, Chapter 28] in the context of *comparison networks*. I take that approach, since it is perfect for discussing word-level parallelism, because it emphasises which comparisons it is possible to make in parallel.

3.4.1 Bitonic Sort

A comparison network is a limited number of *comparators* interconnected by *wires*. A comparator is a device with two inputs, x and y, and two outputs, $x' = \min(x, y)$ and $y' = \max(x, y)$. A comparison network has n input wires $a_1, \ldots, a_n$, which transmit the values to be sorted into the network, and n output wires $b_1, \ldots, b_n$, on which a permutation of those values is eventually presented. If the permutation is into sorted order, the comparison network has sorted the input. The time complexity of a comparison network is the maximal depth of a comparator in it, since a comparator is assumed to compute its output values in constant time, when the (signals on the wires of) input become(s) valid.

I report the *zero-one principle*:

Theorem 3.3 ([CLR90, Theorem 28.2]). *If a comparison network with n inputs sorts all 2^n possible sequences of zeroes and ones correctly, then it sorts all sequences of arbitrary numbers correctly.*

A *bitonic* sequence is defined as the concatenation of a non-decreasing and a non-increasing sequence[3], or a cyclic shift of such a sequence. Batcher's algorithm is recursive in nature. Given a bitonic sequence $\mathcal{X}_d^n = \langle x_1, \ldots, x_n \rangle$ of d-bit integers as input it performs the following steps (cf., Albers and Hagerup [AH97]):

1. Let $h = n/2$. For $i = 1, \ldots, h$, let $m_i = \min(x_i, x_{i+h})$ and $M_i = \max(x_i, x_{i+h})$.

2. In turn recursively sort the sequence $m = \langle m_1, \ldots, m_h \rangle$ of minima, and the sequence $M = \langle M_1, \ldots, M_h \rangle$ of maxima.

3. Return the concatenation of the results of the two recursive invocations.

The correctness of this method needs to be established. Bitonic Sort fits into the framework of comparision networks. Thus, Theorem 3.3 facilitates limiting the proof to binary sequences. A bitonic, binary sequences adheres to the regular expression $0^*1^*0^*$ (taking ones complement yields an analogous case, which is ignored). Two key observations suffice: $m_i \leq M_j$ for $i, j \in \{1, \ldots, h\}$, and m and M are themselves bitonic sequences, so the recursion is permissible. To see this, consult Figure 4 on p. 25.

In the comparison-based model of computation the time complexity of Bitonic Sort is $\Theta(n \log n)$: a logarithmic number of stages of recursion is necessary, and h comparisons must be made at each stage. The significance for word-level parallelism (WLP) is, that for each stage, the batch of h comparisons can be carried out in parallel on the WRAM. Indeed, Albers and Hagerup [AH97] show how to effect such a batch with a constant number of WRAM instructions from the restricted set, provided that the keys are appropriately short. The rest of this section reports their method.

3.4.2 Packed Sorting

As hinted at in Section 2.3, where the packed (k, d)-representation was introduced, it is helpful to understand the packing concept in terms of parallel computing: each of the k fields corresponds to a processor, and the test bits program the processors. This is clear from Figure 5 on p. 26, which illustrates how Bitonic Sort of a packed word $X_{k,d}$ can be carried out using a constant number of WRAM instructions at each of the $\log k$ stages. It has been established:

[3] Allowing for duplicates in each of the two parts.

Figure 4: Proof of the correctness of Bitonic Sort. Cormen *et al.* [CLR90, Figure 28.8].

Figure 5: A copy of [AH97, Figure 1]. Packed bitonic sorting of the sequence $\langle 8, 10, 15, 12, 9, 4, 2, 7 \rangle$ packed in a word $X_{k,d}$ (top row). The idea of comparing in this packed fashion was originally concocted by Paul and Simon [PS82]. $k = 8$ and $d \geq \lceil \log 15 \rceil = 4$. The smallest power of two w obeying the inequality $k(4+1) \leq w$ is $w = 64$. With this choice of w, $d = 7$ is natural. For convenience the figure uses $l = d + 1$ and "$\uparrow$" for left shift. The instructions shown correspond to stage $t = \log k - 1, \dots, 0$, but the machine words in the middle column only illustrate the first, or topmost, stage ($t = \log k - 1$). Empty fields contain no set bits. The words A and B are constructed from X by means of the test-bit mask $T_{k,d} = [1 : \bar{0}| \cdots |1 : \bar{0}]$ and another native constant $A_{k,d} = [0 : k-1| \cdots |0 : 1|0 : 0]$ (cf., Section 2.3): at stage t, a field value belongs in A, if the $(t+1)$th rightmost bit of the corresponding field in $A_{k,d}$ is set. Analogously for B. B is subtracted from A to effect the $h = k/2 = 4$ comparisons, call the result M. The test bits of M (the fourth row from the top) are set for the fields, for which the value in B is less than the corresponding value in A. Define $\textsc{CopyTestBit}(Y) = Y - (Y \downarrow d)$, where Y must be a packed word containing field values of zero only; the test bits of the fields in Y can be arbitrary. The symbol "$\downarrow$" (analogously) denotes right shift. This function copies the test bits of Y into the corresponding fields, and resets the test bits. For example, $\textsc{CopyTestBit}(T_{k,d}) = [0 : \bar{1}| \cdots |0 : \bar{1}]$. Returning to the figure, M' is computed as $M' = \textsc{CopyTestBit}(M \text{ AND } T_{k,d})$. The rest of the figure explains itself.

Theorem 3.4. *A sequence $\mathcal{X}_d^k = \langle x_1, \ldots, x_k \rangle$ of d-bit numbers packed in a word $X_{k,d}$ can be sorted in worst-case $\Theta(\log k)$ deterministic time and constant space on the Practical RAM.*

The real value of packed bitonic sorting is, that it facilitates packed merging of two packed sorted sequences of length k represented by $X_{k,d}$ and $Y_{k,d}$: reverse X, then compare (subtraction) each field value in X with the corresponding one in Y, and construct in X the packed bitonic sequence of minima, and in Y the packed bitonic sequence of maxima, before finally using packed Bitonic Sort on X and Y in turn. The total time complexity is $\Theta(\log k)$. A few trivial modifications of this technique imply, that two packed sorted sequences of length k', where $k' \leq k$, can be merged in time $\Theta(\log k')$ using the (k,d)-representation: if, for instance, $k' = k/2$, the topmost stage of each of the recursive packed bitonic sortings is unnecessary.

Of course, the ability to merge sequences of length k paves the way for merging sequences of arbitrary length n, where $n > k$, as elaborated on by Nilsson [Nil96]: let two non-decreasing sequences $\mathcal{X}_d^n = \langle x_1, \ldots, x_n \rangle$ and $\mathcal{Y}_d^n = \langle y_1, \ldots, y_n \rangle$ of d-bit keys packed in words $X^1, \ldots, X^{n/k}$ and $Y^1, \ldots, Y^{n/k}$ be given. Merge X^1 and Y^1 obtaining Z^1 and Z^2. Then Z^1 contains the k smallest of the $2n$ keys, and the merging problem has efficiently been reduced to one less word. If the last/largest key of Z^2 is x_k, then the procedure continues by merging Z^2 with Y^2, analogously for y_k. The total time complexity is $\Theta((n/k)\log k)$.

Merging implies sorting: Albers and Hagerup [AH97, Corollary 1] have a result in a parallel setting, a sequential version of which essentially reads:

Theorem 3.5. *A sequence $\mathcal{X}_d^n = \langle x_1, \ldots, x_n \rangle$ of d-bit keys, where d is chosen such that $d = \Theta(w/(\log n \log\log n))$, can be sorted deterministically in worst-case linear time and space on the Practical RAM.*

The proof is an algorithm which works as follows:

1. Let $k = w/(d+1)$, such that $k = \Theta(\log n \log\log n)$. Obtain n/k packed sorted sequences $X_{k,d}^i$ of length k as follows. For $i = 1, \ldots, n/k$: in Step $j = 1, \ldots, \log k$ merge two words each containing $k/2^j$ packed sorted sequences of length 2^{j-1} in time $\Theta(j)$ by using packed merging. As $1 + \ldots + \log k = \Theta(\log^2 k)$, the total time complexity (all $X_{k,d}^i$) is:

$$\Theta(\frac{n}{k}\log^2 k) = \Theta(n) \tag{6}$$

2. Merge, as detailed above, the n/k packed sorted sequences $X_{k,d}^i$ into exponentially increasing longer packed sorted sequences. Here, the number of levels is $\log(n/k)$ and Theorem 3.4 on p. 27 is applied n/k times at each level. It costs

$$\Theta\left(\log(n/k)\frac{n}{k}\log k\right) = \Theta(n) \tag{7}$$

3.5 Range reduction and expansion

I now spell out various techniques to reduce and later expand the range, or length in bits, of keys, cf., Section 3.2 for an explanation of this concept in the context of algorithms using word-level parallelism. I abbreviate "range reduction and expansion" as "range redex". See Section 3.6 for examples of how to combine the building blocks of range redex and packed sorting.

3.5.1 Exponential range redex

Kirkpatrick and Reisch [KR84] (KR) were the first to introduce the notion of range reduction. Theirs is an exponential scheme, which reduces b-bit keys to $b/2$-bit keys in linear time. At the core of the approach lie two bucket tables H and L of size $2^{b/2}$. For the moment assume, that they have both been reset leaving all buckets empty. Buckets must obey the *first-in-first-out* (FIFO) condition. Buckets actually used during the range reduction are called *active* to separate them from the ones, which do not have keys inserted into them. One *reduction* of the KR scheme progresses like so:

1. The input keys are bucketed in H on their $b/2$ high-order bits. Simultaneously, the minimal key inserted is remembered for each active H-bucket.

2. For each active bucket in H, move each of the keys in it, except for the minimal one, to L by bucketing on the low-order $b/2$ bits of the key.

3. Present as output the indexes of the active buckets in L and H. These indexes are $b/2$ bits wide.

In Step 3 at most n $b/2$-bit keys are produced. To see this, consider that Step 1 gives rise to at most n active buckets in H. Keys that wind up alone in a H-bucket of their own, do not induce active buckets in L during Step 2, as they are unconditionally minimal. Analogously, if an activated H-bucket contains several keys after Step 1, one of them is left behind during Step 2. This reduces the potential number of active L-buckets by one, which pays for the activation of the H-bucket. The extra administration of maintaining the minimal element can be seen as a slight modification of the FIFO property of buckets.

How this effectively reduces CVISP(n, b) to CVISP$(n', b/2)$, where $n' \leq n$, is best illustrated by regarding the accompanying *expansion*. Suppose the half-sized keys of Step 3, of which there are at most n as just argued, have been sorted, i.e., CVISP$(n', b/2)$ have been solved, and divided into H- and L-indexes. This division is not difficult, and can be realized in linear time in a plethora of ways, for instance, by checking if the corresponding H- and/or L-bucket is active. First, the active L-buckets are visited *in sorted order* by means of the sorted indexes. For each such L-bucket, the contents are moved back into H-buckets on the high-order bits, taking care to insert "after" the minimal elements that were left behind in H during Step 2. The keys from L are *last-in* when compared to these minimal keys, but between the keys from L the FIFO property is maintained during their reinsertion into H. Second, the active H-buckets are next traversed in sorted order, and the contents, the original b-bit keys, are taken out of the buckets and presented as output in FIFO, and thus sorted, order.

The KR method is similar to radix sorting in many respects. Like MSD Radixsort it begins with grouping on the leftmost ($b/2$-bit) character of the keys, *but* it avoids the coupling with each such group of a separate bucket table for the rightmost character. That coupling is exactly the drawback of MSD Radixsort. Instead, the KR method uses L for all groups, bearing a closer resemblance to LSD Radixsort in that respect.

For the time complexity to be $\Theta(n)$, a trick must be applied to facilitate traversing active H-buckets in time linear in the amount of them during Step 2 of the reduction. Traversal is not a problem for the expansion, as the indexes of all active buckets are assumed available in sorted order due to the solution of the reduced sorting problem. As it turns out, the trick also means, that the bucket tables do not have to be initialised.

The trick is accomplished by employing three tables B for "buckets", C for "checking", and I for "initialised" of size $2^{b/2}$ to represent a bucket table. B contains (references to) the actual FIFO buckets. C stores indexes into I, and vice versa. A counter a for "active" is needed as well. Its value is zero to begin with. The invariant is maintained, that the active buckets are precisely those with indexes $I[j], j < a$, so they can be traversed in linear time $\Theta(a)$. Not in sorted order, but in the order of their initialisation. This is sufficient for Step 2.

The trick is a double checking strategy; insertion of x into the bucket with the $b/2$-bit index i proceeds as follows. If $C[i]$ is less than a, $B[i]$ might already have been activated. This is only the case however, if $I[C[i]] = i$, and then bucketing of the b-bit key x inducing the index i reduces to insertion of x into $B[i]$. Otherwise, a is increased by one before the assignments $C[i] = a$ and $I[a] = i$ are carried out, prior to subsequently initialising the bucket $B[i]$ before inserting x into it. Deletion must, of course, also maintain the invariant. The reader can work out the details. Lookup in the bucket table represented by B, C, and I is a simple lookup in B.

Note that the trick is generally applicable to the initialisation in constant time of a table at the cost of asomptotically trebling the space usage of its representation. The time overhead is constant, which trivially is at least as good as any other method, i.e. search structures, of keeping track of the active buckets. See also Aho, Hopcroft and Ullman [AHU74, Problem 2.12].

As promised, KR range reduction (and expansion) employing the trick runs in time linear in n, as can easily be verified. The space complexity, with or without the trick, is $\Theta(n + 2^{b/2})$. If l levels of reductions are used before an intermediate packed method and l subsequent expansions, the contribution of the KR method to the overall complexity is $\Theta(ln)$ time, and $\Theta(\sum_{i=1}^{l} n + 2^{b/2^i}) = \Theta(ln + 2^{b/2+1})$ space. For large b the space demand is a turnoff. The natural solution is to use hashing to represent the sparse tables needed. This reduces the space usage to $\Theta(ln)$ at the cost of an expected constant slowdown and the use of multiplication. See Section 4.2.1 for a discussion of the practical usefulness of KR range reduction, which I will also refer to below as *exponential* range reduction. Section 6.2 describes my implementation of range reduction in the spirit of KR in the context of an algorithm for sorting w-bit keys. The algorithm is KR Sort, cf., Section 3.6.1.

3.5.2 Alphabetic range redex

Regarding a b-bit integer, or binary string, as composed of $k = b/c$ (virtual) characters of bit length c, a corollary of Theorem 3.2 on p. 23 reads:

Corollary 3.1 ([Nil96, Corollary 2.4]). *On the WRAM, using the restricted instruction set, for any positive integer $c \leq a$, the problem of sorting n b-bit keys, $c \leq b \leq w$, can in $O(B/c+n)$ time be reduced to sorting n c-bit keys, where B is the number of distinguishing bits.*

As Equation 1 on p. 5 explains, a is the length in bits of an address on the WRAM. In constructive terms, the range reduction/expansion of the corollary is performed by the first/second invocation of Basic Forward Radixsort in Complete Forward Radixsort. Due to the bucket table used by Forward Radixsort, this range reduction demands $\Theta(n + 2^c)$ space. Since sorting b-bit binary strings takes $\Omega(n + B/w)$ time in the worst case, the reduction is asymptotically optimal. With $c = b/2$ the reduction is similar[4] to KR exponential range reduction, but as the size in bits c of the reduced keys can be chosen freely, it is more flexible. See Section 3.6.1 for a theoretical application of the corollary to integer sorting, and cf., Section 4.2.1 for a discussion of its practical significance. Forward Radixsort also leads to improved bounds on general, as opposed to binary, string sorting, see [Nil96, Section 2.7].

Observe that the alphabetic range reduction is readily applicable to multi-precision integers; in $O(n + N)$ time it can reduce the problem of sorting n multi-precision integers held in a total of N words to that of sorting at most n single-precision integers of w bits each. To see this, consider a character to be a complete w-bit word. The number of distinguishing characters is then trivially at most N, and the time bound follows. But *is* this entirely true? It would require $a > w$, compare with Equation 1. Otherwise, the bucket table of size 2^w alone takes up the whole addressable storage. Alternatively, hashing could be used to give an affirmative answer. Using, for example, Nilsson's Sort, cf., Section 3.6.2, on the reduced problem yields an overall $O(N + n \log \log n)$ time multi-precision sorting algorithm, cf., [AHNR98, Corollary 4.1].

3.5.3 Signature-based range redex

Andersson *et al.* [AHNR98] have invented a Monte Carlo randomised range reduction using multiplication, which in $\Theta(n)$ worst-case time and space reduces the length of n b-bit keys by a factor k with probability at least $1 - 1/n^2$, where $b \leq w$, and k must be chosen such that n keys of $\Theta(k \log n)$ bits each can be sorted in linear time. In view of Theorem 3.5 on p. 27 let $k = w/(\log^2 n \log \log n)$. Choose[5] $d = \Theta(\log n)$, the hidden constant is

[4] $B/c = O(nb/c) = O(n)$.

[5] This d has nothing to do with the notation in Theorem 3.5.

elaborated on below, and let $p = d + 1$. In this section I use the notation $\mathcal{X}_b^n = \langle X_1, \ldots, X_n \rangle$ as opposed to $\mathcal{X}_b^n = \langle x_1, \ldots, x_n \rangle$ for the keys, since they are consistently regarded as containing *chunks* or *fields*, for both of which I use lowercase letters.

The range reduction regards the keys X_i as k *chunks* of $c = b/k$ bits each. I use "chunk" here instead of "field" to not confuse the applied view of the X_i with the packed view introduced in Section 2.3; there is no test bit in a chunk. The range reduction starts with computing the *concatenated signature* Y_i of each key X_i in accordance with the recipe:

1. Choose a hash function h_a randomly from the class $\mathcal{H}_{c,d}$ introduced[6] in Section 2.4.

2. For each X_i, apply h_a to each c-bit chunk of a copy of X_i, and call the result Z_i. Subsequently, for $1 \leq j \leq k$, the d least significant bits of the jth c-bit chunk z_j of Z_i contains the *signature* $z_j = h_a(x_j)$ of the corresponding chunk x_j in X_i. The application of h_a to X_i can be accomplished in constant time with shifts, bitwise boolean operations, and two multiplications with a; the trick is to compute even and odd numbered chunk signatures separately.

3. Prune the superfluous most significant bits from the chunks of Z_i such that a kp-bit essence remains. This essence is placed in S_i, which should be regarded in the packed (k, d)-way. Remember that $p = d+1$, the extra bit is the test bit associated with the d-bit value. The pruning is performed as suggested by Fredman and Willard [FW93], and uses multiplication. See Figure 6 on p. 32. The pruning requires that $kp \leq c$, but if that is not the case, things are actually easier, as will be explained shortly. Call S_i the *concatenated* signature of X_i.

The length of the significant low-order part of S_i is $kp = \Theta(k \log n) = \Theta(w/(\log n \log\log n))$, and by Theorem 3.5 on p. 27 n such keys can be sorted in linear time and space (this is where the test-bits are used). Let $\mathcal{Y}^n = \langle Y_1, \ldots, Y_n \rangle$ denote the sorted sequence of concatenated signatures. From $\mathcal{Y}^n$ a 2^d-ary path-compressed trie $\mathcal{T}_{PC}^{\mathcal{Y}}$ over the concatenated signatures can be constructed in linear time and space by regarding the field values of Y_i as d-bit characters. See [Hag94] for details. The construction requires $\Theta(n)$ multiplications and is performed such that each internal node of $\mathcal{T}_{PC}^{\mathcal{Y}}$ is marked with the length of the corresponding common prefix, and holds a direct reference to a leaf below it, and such that each leaf of $\mathcal{T}_{PC}^{\mathcal{Y}}$ holds a reference to the corresponding input key X_i. Since $d = \Theta(\log n)$, and the trie has $\Theta(n)$ 2^d-ary nodes, a node should not contain a table of size 2^d of references to its children; this would imply an $\Omega(n^2)$ space complexity. Never fear, the construction of $\mathcal{T}_{PC}^{\mathcal{Y}}$ can be done using a linked list for each node to store references to its non-empty children. This also means that these children can be visited in linear time in the amount of them.

Consider the probability that the chosen h_a is 1-1 on the cardinality at most kn set $\mathcal{C}$ of c-bit chunk values present in the original keys X_i. By

[6] On conferring p. 12 substitute k for c, and ℓ for d.

Figure 6: Pruning, $k = 4$. The top row shows a key X, and the row beneath it shows the corresponding Z. A multiplication of Z with a suitable native constant N (third row) places the signatures adjacently in the result $M = N \cdot Z$. N is k adjacent copies of a bit pattern, which has only the least significant bit set, of length $(k-1)p$. To avoid interference of signatures with each other in the computation of M, the inequality $kp \leq c$ must be obeyed. The bottom row is the concatenated signature S. [AHNR98, Figure 1].

Lemma 2.1 on p. 12 the probability is at least $1 - |\mathcal{C}|^2/2^d$. Assume $k \leq n$ without loss of generality. Choosing $d = 6 \log n$ then makes the probability at least $1 - 1/n^2$.

Assume that h_a *is* 1-1 on $\mathcal{C}$. $\mathcal{T}_{PC}^{\mathcal{Y}}$ is then isomorphic to the path-compressed trie $\mathcal{T}_{PC}^{\mathcal{X}}$ corresponding to the original keys (viewed as strings from a c-bit alphabet). The only difference is the order of sibling edges. Going from $\mathcal{T}_{PC}^{\mathcal{X}}$ to[7] $\mathcal{T}_{PC}^{\mathcal{Y}}$ changes the rank of a child within its set of siblings in accordance with a permutation, which depends on a, the choice of the hash function h_a.

The original sorting problem thus reduces to undoing the effect of this permutation; for each internal node of $\mathcal{T}_{PC}^{\mathcal{Y}}$ the outgoing c-bit chunk/character values of non-empty children of the corresponding node in $\mathcal{T}_{PC}^{\mathcal{X}}$ must be sorted. That is the reduced problem. For the range expansion the sibling edges of $\mathcal{T}_{PC}^{\mathcal{Y}}$ must be rearranged in accordance with the result of that sorting. Afterwards, the original keys X_i can be traversed in sorted order during a linear time traversal of $\mathcal{T}_{PC}^{\mathcal{Y}}$.

Above it was explained how $\mathcal{T}_{PC}^{\mathcal{Y}}$ was constructed such that each node holds some auxiliary information. It is easy to see, that this information can be used to generate the recursive sorting problem of outgoing c-bit values in linear time, such that each value is *tagged* with the identity of the parent

[7]The transition is conceptual. $\mathcal{T}_{PC}^{\mathcal{X}}$ is not constructed, but only used in the description of the range reduction.

node, and with its d-bit signature. As there are $\Theta(n)$ internal nodes in $\mathcal{T}_{PC}^{\mathcal{Y}}$ (and $\mathcal{T}_{PC}^{\mathcal{X}}$), and $d = \Theta(\log n)$, such a tag is conveniently $\Theta(\log n)$ bits stored in $\Theta(1)$ extra w-bit machine words. $\mathcal{T}_{PC}^{\mathcal{Y}}$ has e edges, where $n \leq e \leq 2n-2$, but by excluding the minimum outgoing character value for each set of sibling edges, the number of c-bit values in the recursive sorting problem is at most n.

As $c = b/k$, a factor k reduction in key length has happened in linear time and space with probability at least $1 - 1/n^2$. The recursive sorting problem must regard the tags as satellite data of the c-bit values. When it has been solved, $\Theta(1)$ invocations, using a total of $\Theta(n)$ time and space, of stable Bucketsort of the c-bit values of the recursive problem on their tag values suffice to group siblings such that they occur in sorted order within the groups. That is what the expansion requires.

Recall, that Step 3 above is impossible if $kp > c$. Since $kp > c \Leftrightarrow b/(kp) < k$, the signature-based range reduction is not applicable if $b/(kp) < k$. However, in that case one can view a b-bit key as divided into $b/(kp)$ chunks; the resulting kp-bit chunk values of the reduced problem can be sorted directly in linear time (Theorem 3.5), and no further range reductions are necessary.

3.5.4 Cluster-oriented range redex

Thorup [Tho97] has shown:

Theorem 3.6 ([Tho97, Theorem 1.2]). *A sequence* $\mathcal{X}_{w/\lambda}^{n} = \langle x_1, \ldots, x_n \rangle$ *of* (w/λ)*-bit keys can be semisorted in* $O(n + n \log\log n/\lambda)$ *expected time and linear worst-case space on the* Practical RAM *with a probability of success of at least* $1 - 1/n$.

Here to "semisort" means to permute the keys such that equal-valued ones are placed adjacently to each other. Cf., Section 3.6.2 for a discussion of the impact of this theorem on range reduction. I have used "(w/λ)" as opposed to "b" in the theorem to make that discussion easier. When arguing the correctness of the theorem two cases are considered:

A. $\log w = \omega(\log\log n) \Leftrightarrow w = (\log n)^{\omega(1)}$

B. $w = (\log n)^{\Theta(1)}$

Do not confuse the Greek letter "ω" with the Latin letter "w". I only describe the algorithm underlying Case A, as the developments involved clearly illustrate the ideas, which also make for supporting Case B, although the latter case is more complicated. In this section shifts are assumed to be cyclic; one cyclic shift can be simulated by two zero-filling ones followed by **and** of the results.

Assume Case A then. Let a sequence $\mathcal{X}_b^n = \langle x_1, \ldots, x_n \rangle$ of b-bit input keys be given. Suppose that the keys have length $b = \omega(w/k)$, where $k = \log n \log\log n$, i.e., that Theorem 3.5 on p. 27 cannot immediately be applied to sort them, and let $c = b/k$. View a b-bit key as k *chunks* of c bits each.

The algorithm computes a c-bit *signature* of each key, since Theorem 3.5 *can* be used to sort c-bit keys. Let α be a random b-bit string. Define $\mathtt{Sign}_\alpha : \{0, \ldots, 2^b - 1\} \to \{0, \ldots, 2^c - 1\}$ by Algorithm 1; $y\langle j \rangle_{/k}$ denotes the jth c-bit chunk of y, "$\bigoplus$" is used for **xor**, and "$\wedge$" is used for **and**.

$y = x \wedge \alpha$
$z = \bigoplus_{j=1}^{k} y\langle j \rangle_{/k}$
return z

Algorithm 1: $\mathtt{Sign}_\alpha(x)$

By a clever implementation, $\mathtt{Sign}_\alpha$ can be applied to all keys x_i in total linear time. The trick is to use a continued halving approach to compute the **xor**-summation; repeatedly "fold" y around the "middle". This takes $\Theta(\log k)$ time per key, because there are k chunks in a key. As $\log k = \omega(1)$, folding each key in isolation is insufficient. Instead, the computation of $\mathtt{Sign}_\alpha$ must progress on all y_i simultaneously. Because the intermediate results become steadily shorter, they can be packed in fewer and fewer words. The running time is therefore a geometric progression, and is dominated by the cost of the first folding, which computes the **xor** of the "top" and "bottom" $(b/2)$-bit halves of each y_i, and then packs the n results in $n/2$ machine words.

Next, the sequence $\langle (\mathtt{Sign}_\alpha(x_i), i) | x_i \in \mathcal{X} \rangle$ is sorted in linear time by Theorem 3.5 and a subsequent invocation of stable Bucketsort, cf., the discussion of Signature Sort in Section 3.6.2 for details. The sorted result facilitates partitioning $\mathcal{X}$ into *clusters*. A cluster is a maximal subset of $\mathcal{X}$ of keys with a common signature. With $r = 3k \log n$, Thorup shows that with probability at least $1 - 1/n$, there is no pair of keys (x_i, x_j) belonging to the same cluster such that $\|x_i \oplus x_j\|_1 > r$. Success in this matter is assumed below. Here, $\| \cdot \|_1$ denotes the *Hamming norm*, i.e., the number of ones in the binary representation.

Duplicate keys cannot reside in different clusters, but a cluster does not necessarily contain duplicate keys only. What now remains is thus to *bust* each cluster $\mathcal{Y}$. Set $\mathcal{Z} = \{y \oplus y_0 | y \in \mathcal{Y}\}$, where $y_0 \in \mathcal{Y}$ is chosen arbitrarily. Because "$\oplus$" is idempotent, the busting reduces to finding duplicates in $\mathcal{Z}$. This is intuitively easier, as $\forall z \in \mathcal{Z} : \|z\|_1 \leq r$, a statement which does not have an analogue for $\mathcal{Y}$.

The method for busting is to compute for each key $z \in \mathcal{Z}$ a second kind of c-bit signature. A family $\mathcal{F}$ of functions is used. A function $f \in \mathcal{F}$ has the following properties:

1. For any $z \in \mathcal{Z}$, the computation of $f(z)$ errors with probability at most $1/2$.
2. The 1-1 condition is obeyed on the subset of keys of $\mathcal{Z}$ on which f is successful.
3. $f : \{0, \ldots, 2^b - 1\} \to \{0, \ldots, 2^c - 1\}$

The recipe for busting is to pick a function at random from $\mathcal{F}$, send all $z \in \mathcal{Z}$ through it, and recur on the subset of failures with a new function from $\mathcal{F}$. The first property ensures, that the total number of function values, which must be computed, is expected to be less than $2n$. The sequence $\langle(\mathcal{F}(z_i), i)|z_i = x_i \oplus y_0, x_i \in \mathcal{Y}\rangle$, where $\mathcal{F}(z_i)$ is the $\mathcal{F}$-fingerprint of z_i, can be sorted in linear time, and the result used to identify duplicates within $\mathcal{Y}$. Note, that there is one such sorting for each cluster $\mathcal{Y}$, but the total time used on the sortings is $O(n)$.

It remains to elaborate on $\mathcal{F}$. Now regard each of the k chunks of a b-bit key as broken down further into r^2 *pieces*, making for a total of $p = r^2k$ pieces of length $l = b/p$ per key. The idea is to "hash" the non-empty pieces, i.e., those containing set bits, into the (at most) c least significant bits of a machine word.

In order for the functions in $\mathcal{F}$ to be injective, for each key all non-empty pieces are first labelled in total constant time by using the native mask $A_{p,l} = [0 : p-1|\cdots|0 : 1|0 : 0]$; by Case A, each label can be represented by at most l bits. This doubles the lengths b and l, but is asymptotically insignificant. The function value $f(z)$ is computed as follows:

1. For each chunk of z perform a random, cyclic shift of the r^2 pieces within it (shift distance a multiple of l). The k displacements are the same for each application of f, but change when another function $f' \in \mathcal{F}$ is picked. Such selection in turn merely consists of choosing k new displacements, one for each field. $\mathcal{F}$ is thus only represented implicitly.

2. Compute and return the bitwise disjunction **or** of the k c-bit chunks of the result of the preceding step.

Remember, that $\|z\|_1 \leq r$. Hence, there are at most r non-empty pieces in a $z \in \mathcal{Z}$. The probability that any two of these *collide*, i.e., wind up at the same offset within their chunk(s), during Step 2 is therefore at most $\binom{r}{2}/r^2 < 1/2$. If there is no such interference, f is 1-1 on z. Thus, $\mathcal{F}$ fits the bill. Thorup implements the computation of $f(z)$ efficiently enough to obtain the bounds in Theorem 3.6.

3.6 Using the building blocks

With the fundamental building blocks now described, I next give an overview of how they have been combined to yield a algorithms for sorting a sequence $\mathcal{X}_b^n = \langle x_1, \ldots, x_n\rangle$ of b-bit integers. The reader might want to confer Section 3.2 for an explanation of the subsection headings.

3.6.1 First-generation algorithms

KR Sort Kirkpatrick and Reisch [KR84] apply their exponential range reduction $l = \log(b/\log n)$ times and sort the resulting keys of $b/2^l = \log n$ bits each with Bucketsort. This KR Sort is a $\Theta(ln)$ time and $\Theta(ln+2^{b/2+1})$ space algorithm. As Section 3.5.1 explains, the $2^{b/2+1}$ term in the space bound can be reduced to $\Theta(ln)$ by means of hashing. This makes the time bound expected, as opposed to worst-case. Analogous remarks apply below, when exponential range reduction is used. Cf., Section 4.2 for an explanation of why (a slightly modified version of) KR Sort was chosen for implementation, cf., Section 6.2.

With alphabetic range reduction Using Corollary 3.1 on p. 30 and KR Sort as subroutine, for any integer $k \geq 2$ sorting can be carried out in $\Theta(kn + n \log(c/\log n))$ time and $\Theta(2^c + n \log(c/\log n))$ space, where $c = b/k$. Cf., Nilsson [Nil96, Theorem 2.8].

3.6.2 Second-generation algorithms

These are the algorithms, which I have also called "modern".

AHNR Sort and Nilsson's Sort Andersson *et al.* [AHNR98] suggests using $2 \log \log n$ levels of exponential KR range reduction, and sorting the resulting $b/(\log^2 n)$-bit keys by Theorem 3.5 on p. 27. The complexity of the range reductions dominates here, so the overall performance is $\Theta(n \log \log n)$ time and $\Theta(n \log \log n + 2^{b/2+1})$ space. For lack of a better name I call the algorithm AHNR Sort.

Nilsson have obtained the same bounds. Let $k = \log n$. After half as many, namely $\log \log n$, exponential range reductions, the resulting keys of $b/\log n$ bits each are split into n/k groups of size k, each one of which is sorted by comparison (Mergesort). This takes $(n/k)\Theta(k \log k) = \Theta(n \log \log n)$ time in total. Therewith, the stage is set for packed merging of the length k sorted sequences. This takes $\Theta(n \log \log n)$ time too, as argued by the left-hand side of Equation 7 on p. 27. I call this algorithm Nilsson's Sort.

Signature Sort Recall, that signature-based range reduction, cf., Section 3.5.3, reduces the length of b-bit keys by a factor $k = w/(\log^2 n \log \log n)$ with probability at least $1 - 1/n^2$ in linear time and space using multiplication. For $b \leq w$, the key length b must drop by a factor $\Theta((b/w) \log n \log \log n)$ before Theorem 3.5 can be used to terminate the sorting in linear time and space. Such a drop is realized by $O(1 + \log \log n / \log k)$ factor k reductions; to see this use the base k logarithm. As each reduction uses linear time and space it follows:

Theorem 3.7 ([AHNR98, Theorem 3.1]). *On the WRAM, n integers of b-bits each, $b \leq w$, can be sorted in $O(n + n \log \log n / \log k)$ time and space, where $k = w/(\log^2 n \log \log n)$, with probability at least $1 - 1/n$ using the multiplication instruction set.*

The corresponding algorithm is called Signature Sort. The signature-based range reduction requires the algorithm to be used for the reduced problem to be able to handle satellite data. The packed algorithm (corresponding to Theorem 3.4 on p. 27) underlying Theorem 3.5 can do this, provided that the size of the satellite data associated with each short key is within a factor $O(1)$ of the short key length. The trick is to pack the satellite data as well. The *tags* associated with the keys in the signature-based range reduction are $\Theta(\log n)$ bits long. Therefore, Theorem 3.5 cannot be used on $o(\log n)$-bit values to solve the reduced problem; Bucketsort is used instead.

A corollary follows:

Theorem 3.8 ([AHNR98, Corollary 3.1]). *If $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, we can sort n integers of b-bits each, $b \leq w$, on the multiplication WRAM in linear time and space with probability of error at most $1/n$.*

Cluster Sort Why does the cluster-oriented method from Section 3.5.4 embody a range reduction? I cannot say. Thorup [Tho97] claims it implies "Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations". As far as I can tell, his arguments are:

1. n w-bit keys can be grouped on their $\beta = \sum_{i=1}^{2 \log \log n} w/2^i$ most significant bits in $O(n \log \log n)$ expected time and linear worst-case space by means of $2 \log \log n$ applications of Theorem 3.6 on p. 33. This is true, and can readily be verified. There is a probability of error of at most $1/n$ associated with each application. I do not elaborate on the implication of this, i.e., derive a Las Vegas version of the grouping.

2. Each of the resulting groups can be deterministically sorted on the "remaining" least significant bits of the keys in it in total (all groups) linear worst-case time and space on the Practical RAM by Theorem 3.5.

The problem, as I see it, is, that Step 1 does not organise the groups in sorted order on the β most significant bits.

3.7 Final remarks

Use of the techniques employed to derive the "modern" sorting algorithms has also lead to theoretically remarkable results on search trees, priority queues, and tries. The *fusion search tree* of Fredman and Willard [FW93] was the first data structure to support $o(\log n)$ time primitives, and thus $o(n \log n)$ sorting, and the ideas supporting it were further developed in [FW94]. Andersson [And95, And96a, And96b] has built on and improved those ideas. Thorup [Tho96, Tho98] and Raman [Ram96] have concentrated on priority queues. All the research just mentioned has obvious, and less obvious, cf., Thorup [Tho96, Tho97], connections with sorting. A more detailed account of this is outside the scope of this thesis, however. Andersson and Nilsson's [AN94a] *level-compressed trie* does not rely on word-level parallelism, but it is derived in the WRAM model of computation. The *stratified search tree* of van Emde Boas *et al.* [vEBKZ77, vEB77] is significant for some of the work mentioned in this section.

4 Discussion of potential practical significance

The survey in Chapter 3 sets the stage for a discussion of the potential practical value of research published so far, which in turn makes for the selection of the WRAM algorithm(s) and/or data structure(s), which seem most likely to be useful in practice. The structure of this chapter is as follows. In Section 4.1 I discuss what characterises a practical algorithm. Next, in Section 4.2, I use these criteria to select among the ideas from the survey the candidate(s) for implementation.

4.1 Selection criteria

I now line up the basis of deciding what to implement. The question is, which properties are crucial, and which ones are desirable, but can be waived? And vice versa, which properties should be avoided, but not at all costs, and which ones are entirely unacceptable. The guidelines below, which answer these questions, are the basic rules of *performance engineering.*

Simplicity Always! This is a goal in itself. A simple algorithm has several advantages. It is easier to communicate. Both to ones peers by means of publications, and to the computer by means of a programming language. There is also the added benefit that the size of the object code will be smaller for a simpler algorithm, thus reducing the number of instruction cache misses during its execution. As discussed on p. 1.1.3, good memory behaviour is of utmost importance for the efficiency of an algorithm on modern computers. See also the items on memory usage below.

Constant factors All asymptotic expressions ($O(\cdot), \Theta(\cdot), \Omega(\cdot)$) should be *tight.* For example, $O(n)$ is not interesting if it really means "at most $1000n$ for $n \gg 2^{20}$". An $O(n \log n)$ bound covering for "at most $4n \log n$ for $n \geq 2^{10}$" would be preferable for most values of n. Tightness is usually implied by simplicity.

Memory demands Really huge memory demands, e.g., $\Theta(n2^{\epsilon w})$, should be scorned without reservations. A logarithmic factor, i.e., a $\Theta(n \log n)$ memory footprint, should be avoided, at least if the order of magnitude of n is millions. On the other hand, there is no reason to hold out for linear space at all costs. In particular, a simple and time-wise efficient algorithm using $\Theta(n+2^{\epsilon w})$ space is not to be rejected, if ϵ is small enough, that the constant, independent of n at least, overhead $\Theta(2^{\epsilon w})$ is manageable. Of course, the hidden constants influence whether or not that is the case.

Locality of data reference A somewhat large space complexity is easier to tolerate, if *spatial* or *temporal* locality of data reference is predominant. This is due to the memory hierarchy of real machines, as discussed on p. 7. An almost uniformly random access pattern causes lots of misses at all levels of buffers, which kills performance.

For instance, the WORKSTATION 500AU, cf., Section 5.2.2, which I have used for the experiments for this thesis has an on-chip cache of size 96kb, and an 8Mb board-level cache. If the n keys are accessed in a predominantly sequential[8] fashion, the ones "currently" used will be present in the board-level cache at least, and the on-chip cache can be used for auxiliary data structures, i.e., a bucket table.

In light of the points just made, packed techniques gain one immediate advantage; they need to read and write fewer machine words, simply because each word contains several keys. The question is, whether this upper hand is waived due to properties of other parts of the word-level-parallel algorithm, in which the packed method partakes. Here I am alluding to the reduction to and expansion from short keys.

Arithmetic Integer multiplication, division, and remainder calculations should be avoided if possible. As mentioned in connection with the introduction of the WRAM model, cf., Section 1.1.3, multiplication is theoretically harder than, for instance, addition. The real difference in complexity in terms of execution time will vary from machine to machine, and multiplication, as well as division and remainder calculation, is often very efficient in hardware. However, it is a good rule of thumb to consider these instructions expensive. For the WORKSTATION 500AU at least this maxim is true; Section 7.1.2 details how using right shifts (**right**) and bitwise boolean conjunctions (**and**) instead of divisions and remainder calculations crucially sped up NEXT-GEN, a cache simulator of my own making.

Addition and subtraction can be used freely, as these instructions are efficient in hardware. Indeed they must be, if the possibility of using tables, which are a core building block in programming, is to exist. The reason of course is, that indexing a table uses addition, perhaps implicitly by means of an offset addressing capability of the underlying machine. If such a capability does not exist, the addition explicitly uses a machine instruction before the lookup (yet an instruction). I here take "indexing" to mean non-sequential access. Note that indexing a multi-dimensional table implies a hidden use of multiplication.

As regards floating-point numbers, analogous remarks can be made. This perhaps is less interesting in the context of processing integers. Cf., Andersson and Thorup [AT96] however, for an ingenious way of computing integer logarithms by means of integer to floating-point conversion.

Randomisation A stochastic influence can be simulated by means of quasi- or pseudo-random number generator. As Section 7.2 explains, modern generators can efficiently generate numbers uniformly distributed over $\{0, \ldots, 2^w - 1\}$ without using the likes of multiplication. Using such a generator for **rand** makes it entirely agreeable to draw $O(1)$, $O(\log n)$, or even $O(n)$ random word values. Thus, in terms of the stochastic influence, for instance hashing à la Dietzfelbinger *et al.* [DHKP97], see Section 2.4, and Quicksort with random selection of the partitioning element, cf., Cormen *et*

[8] "Sequential" as opposed to "random", not "sequential" as opposed to "parallel".

al. [CLR90, Section 8.3], are practical algorithms. Suspicion of simulations of intricate, non-uniform distributions by means of **rand** is in order.

Uniformity Native constants, cf., Section 2.3, are totally harmless. This statement suffices for the purposes of this thesis, since algorithms, one would seriously consider to implement, are *strongly* uniform in the sense of Ben-Amram and Galil [BAG97].

4.2 Darwinian selection

So, which algorithm to implement? In this section the fittest one from Section 3.6 is chosen based on the above discussion.

4.2.1 Reducing key length

A range reduction and complementing expansion from Section 3.5 must be decided on.

Exponential range redex Straightforward exponential range redex as pioneered by Kirkpatrick and Reisch [KR84] (KR) seems the more attractive of the lot. It is conceptually simple and tight. It *is* greatly disadvantaged by its huge memory demand and random access pattern. As mentioned in Section 3.5.1, the KR method uses $\Theta(ln + 2^{w/2+1})$ space (and $\Theta(ln)$ time) for l levels of reductions of keys of initial length w. This is excessive for real machines; substitute, for instance, 64 for w. Accordingly, exponential range redex must be implemented using hashing to represent the sparse tables needed, at least until the key size in bits is reduced enough, that the bucket tables can be directly allocated on a real machine. Hashing implies the use of multiplication for computing the hash function, and Las Vegas randomisation for choosing it, and an expected constant factor slowdown on table lookups, cf., Section 2.4. Thus, it is not obvious, that the KR method has practical significance, but it is the one among the candidates, which seems least displeasing, as I elaborate on below. As such, I choose it for implementation, cf., Sections 5.2 and 6.2.

Alphabetic range redex The alphabetic range redex, cf., Section 3.5.2, embodied by Forward Radixsort, cf., Section 3.3.3, uses only $\Theta(n+2^c)$ space, where c is the length in bits of the reduced keys. The bucket table of size $\Theta(2^c)$ is in the nature of things accessed randomly, but this is bearable since c can safely be regarded small. The group list is accessed sequentially at all unfinished groups, when groups are split after the bucketing.

In the context of alphabetic range reduction of binary strings, i.e., machine word integers, the list L of (references to) the strings is of course realised as a list of the integers themselves. L might even be implemented as a table. A doubly linked list of "references" is still needed to represent a group, however. The links impose a considerable relative space overhead, when the data to be sorted is not general strings.

Also, the alphabetic range reduction can be said to perform the same job twice; the complete version of Forward Radixsort runs the basic one (modified) two times on the same input. The first run collects sibling edges of the underlying trie, i.e., the c-bit reduced keys, and the second run, i.e., the expansion, uses the intermediate sorting of those keys to output the original b-bit keys in sorted order.

As Nilsson [Nil96, p. 35] points out, the basic version of Forward Radixsort, which employs straightforward traversals of the $\Theta(2^c)$ size bucket table, is more important in terms of practicality than the complete version, which is *only* interesting *because* it effects the alphabetic range reduction, which has theoretical importance, cf., Section 3.6.

The constant c, and hence the bucket table size, can be chosen freely with Forward Radixsort, whereas $c = b/2$ is mandatory for the KR method. This speaks in favour of alphabetic range reduction over exponential range reduction, but I still choose to implement the latter. This decision is backed by the fact, that Nilsson himself uses Nilsson's Sort (incorporating the KR method), cf., Section 3.6.2, and not Forward Radixsort to sort integers in his Ph.D. thesis [Nil96, Section 4.2].

He *has* programmed Basic Forward Radixsort of general strings, which turns out to be worthwhile [Nil96, Section 4.1], but that is outside the scope of my work.

Signature-based range redex The signature-based range redex of Andersson *et al.* [AHNR98] from Section 3.5.3 is not very inviting in the context of sorting single-precision integers. It divides keys into k fields, where $k = w/(\log^2 n \log\log n)$, to reduce the length of b-bit keys to $c = b/k$. For example, $n = 2^{20}$, which is an entirely realistic instance size, implies $k \approx w/1729$, and hence the necessity of a very large word size indeed, in order for the reduction to have any meaning.

A downside too is, that the reduction uses multiplication. Furthermore it is a Monte Carlo method; each factor k reduction in key length errors with probability at most $1/n^2$.

The requirement of the signature-based range redex of being able to sort short integers, i.e., the concatenated signatures, fast, is not a problem in practice, even though Theorem 3.5, which is important in deriving the theoretical resource bounds for Signature Sort, cf., Section 3.6.2, might not necessarily be used for this purpose. All in all however, in terms of practicality the reduction does not qualify itself. Rejected.

Cluster-oriented range redex Thorup's [Tho97] cluster-oriented range redex of Section 3.5.4 is not attractive, as I next elaborate. It is rather complicated, especially for short words, i.e., in Case B of p. 33, which I have not pursued. Furthermore, it is a Monte Carlo method with a probability of error; at most $1/n$. What really disqualifies it though, is the impossibility of choosing between the two different cases for a given value of n.

4.2.2 Sorting short keys

With exponential range redex chosen, it remains to decide, which method to use to sort short keys. To begin with, the question "How long is a 'short' key?" must be answered. Let $w = 64$, the word size of mass-produced computers today. The algorithms presented in Section 3.6 use at least $\log\log n$ exponential range reductions to shorten b-bit keys, $b \leq w$. For $n = 2^{20}$, this means that "short" is at most $d = 64/2^{\log 20} = 2$ bits.

Bucketsort, cf., Section 3.3.1, should be used to sort such d-bit keys. It is a simple, tight, deterministic algorithm, which does not use multiplication. Bucketsort employs a bucket[9] table of size $\Theta(2^d)$, which is used heavily (random access) during the permutation of the input. The permutation can be effected accessing the input and output sequentially with 1 and 2^d pointer(s) respectively; this is also true of the in-place version of Bucketsort. Real machines have on-chip caches large enough to make Bucketsort entirely feasible, when $d \approx 10$, cf., Section 6.1.

One should not consider using packed sorting, cf., Section 3.4. The packing and unpacking of the n keys requires two tight $\Theta(n)$ traversals alone, which, as just explained, is basically all that is needed for Bucketsort to finish. Even disregarding that, packed sorting is not recommendable. The actual packed merging is based on packed bitonic sorting, see Figure 5 on p. 26, which uses quite a lot of instructions to sort a packed word $X_{k,d}$. In Section 5.1 I explain how Rahman and Raman [RR98] have demonstrated experimentally, that packed sorting is not worthwhile, which supports my decision not to use it.

4.2.3 And the winner is ...

From the preceding two sections it is clear, that from the set of algorithms in Section 3.6, KR Sort should be chosen for implementation, and I have done so. As KR Sort is a first-generation WRAM algorithm, this implies, that I have not actually experimented with any of the "modern" algorithms; I have deemed them practically insignificant based exclusively non-empirical evidence, and rightfully so I think.

Section 6.2 introduces my `KR_sort` program. It uses `contender` to sort the keys resulting from the exponential range reduction. The `contender` program implements a small variation on Adaptive Radixsort, which I have called Contender. This choice of name is explained in Section 5.2.1, and Section 6.1 describes the implementation. Using `contender` in `KR_sort` enables me to experiment with the number of reduction levels, since the reduced keys do not have to be "short" in order to be sorted by Contender.

[9] Each bucket is conveniently a counter and a pointer.

5 Method of empirical evaluation

In this chapter I first summarise existing tests from the literature in Section 5.1. The purpose of the summary is to spell out, together with the observations of Section 4.2, the basis of my decision on what to try out in practice. I present my decision and argue for it in Section 5.2, hence that is where an outline of the contribution of this thesis to the area of empirical studies of WRAM sorting algorithms can be found. As it will appear, and as revealed in Section 4.2.3, I have not experimented with algorithms employing word-level parallelism, i.e., packed sorting, because doing so would bring about no knowledge of their value, which is not already present in the literature. Instead, I have tested KR Sort, cf., Section 3.6.1, against Contender, cf., Section 5.2.1, which is modelled after Adaptive Radixsort. In the empirical evaluation I concentrate on full length keys, i.e., let $b = w$.

5.1 State of affairs

This section is a summary of existing empirical work with word-level parallelism in the context of internal integer sorting known to me.

Nilsson [Nil96] compares implementations of Heapsort, Mergesort, Quicksort and Adaptive Radixsort by other researchers, see [Nil96, p. 67], with his own "unoptimised" realization of Nilsson's Sort, cf., Section 3.6.2. He tests with input created with the random number generator presented by Park and Miller [PM88], cf., Appendix C, and with 32-bit integers constructed from four consecutive 8-bit ASCII characters of various texts. The largest value he uses for n is 2,000,000. The results are "virtually identical" for the two types of input; he finds Nilsson's Sort to be faster than Heapsort and Mergesort, but slower than Quicksort and Adaptive Radixsort, which are known to be efficient in practice. The problem with his efforts is, that he only executes the programs on machines with $w = 32$, which does not provide the capability to pack many short keys in a machine word, thus disadvantaging packed sort.

Rahman and Raman [RR98] avoid that deficiency of Nilsson's experiments by *simulating* machines with word lengths greater than 32 bits. This is done by means of a `C++` class `Longint`, which implements the concept of arbitrary word length; the value to use for w in the simulations must be hardwired into `Longint` at compile time. The `Longint` class is suitably overloaded and maintains "counts of all operations performed on instances of class `Longint`." With minor modifications a program written for real machine words can be converted to one processing `Longint`s.

Rahman and Raman [RR98] have two orthogonal types of experiments:

- Try to determine, if KR Sort, cf., Section 3.6.1, is competitive to LSD Radixsort, cf., Section 3.3.2, when using 11-, 16-, or 22-bit characters in the latter. These radix sizes "minimised (locally) operation counts for $w = 64$ and were retained for larger values of w." Here "operation counts" denotes the per-item, i.e., divided by n, "total of all the operations on `Longint`s and `int`s." The number of "random" keys in the input ranges from 10,000 to 8,000,000, and $w = 64, 128, 256$. It is not clear what "random" means. For each configuration Rahman and Raman find the best of the three LSD radix sortings to use approximately 11 operations per item per character position, whereas KR range reduction uses about 31 operations per item per level ("except for small values of n"). As the first couple of levels of range reduction achieve the effect of several invocations of Bucketsort, these results recommend a *hybrid* algorithm: use exponential range reduction until the keys are small enough, that LSD Radixsort will finish faster, and then switch. The simulations highlight the need for hashing, as some configurations failed, because the amount of virtual memory was limited to 400Mb. *Real-time* experiments ($w = 64$) on a SUN ULTRASPARC, which has a machine word size of 64 bits, were performed to evaluate the impact of using hashing for the sparse tables needed for exponential range reduction. With hashing Rahman and Raman on the basis of the running times measured for those tests expect KR Sort and the hybrid algorithm to outperform LSD Radixsort for $w = 256$ and $w = 128$ respectively.

- Investigate the practical importance of packed merge, cf., Section 3.4. The input to packed merge is two packed sorted sequences of size n, where n ranges from 512,000 to 4,000,000, containing "random" 7-bit keys factor 8 packed in 64-bit words. The opposition is standard, non-packed merge of two sorted sequences of n "random" 64-bit integers each[10]. Operation counts are accumulated by processing `Longints` (with $w = 64$), *and* real running times are measured on the SUN ULTRASPARC. The total operation count is about 9.8 times larger for packed merge than for non-packed merge. This does not exactly build confidence in packed sorting. However, in terms of real time measurements, the performance of packed merge is worse by only approximately 50% (when including the time to pack). I think Rahman and Raman are on to something, when they credit this discrepancy to the higher locality of reference, which is a direct consequence of packing. In any event, the lesson to learn is to not judge the performance of an algorithm on operation counts alone.

5.2 The master plan

Any shortcomings of the experiments outlined above are to be expected, as they are the first reported in the literature. My empirical exercises should be seen as an extension of existing research attacking the core of its drawbacks. In this section I give an overview of what I have done.

5.2.1 Challengers of KR Sort

As the experiments of Rahman and Raman [RR98] show, the main advantage of KR Sort over LSD Radixsort lies precisely the exponential range

[10]Non-packed merge was also tested on two skewed sequences $\langle 3, \ldots, 3, 0\rangle$ and $\langle 2, \ldots, 2, 1\rangle$. One can ask oneself why there are no corresponding packed experiments.

reduction. Consider a doubling of the word length; indeed this seems to be the evolution of mass-produced hardware. The doubling effectively means, that the first range reduction level eliminates twice as many LSD Radixsort character positions as before. This exponential growth in the value of KR Sort over LSD Radixsort is exactly why Rahman and Raman believe that "the *hybrid* KR-radix sort will outperform radix sort on 128-bit integers even if we use a hash table."

In my opinion it is more interesting to compare KR Sort with a Radixsort other than LSD, which unconditionally inspects all (virtual) characters of the input. Furthermore, LSD Radixsort accesses the complete input area randomly during *every* Bucketsort pass, which is bad for cache performance.

Therefore I have implemented a more clever, MSD-oriented Radixsort algorithm, referred to below as Contender, and compared it with KR Sort. As a benchmark I have also included `sort` from the `C++` STANDARD TEMPLATE LIBRARY (STL) in the comparison. The `sort` program can, by its inclusion into STL, be considered state-of-the-art for general purpose sorting. See Chapter 6 for an introduction to the `KR_sort` and `contender` programs.

For the compiler I have used, cf., Section 7.1.4, `sort` implements Introsort, cf., Musser [Mus97]. Introsort is a hybrid of a version of Quicksort, which uses a median-of-three selection strategy for the partitioning element, and Heapsort. On "random" (Musser's phrasing) input Introsort is a little faster than Quicksort. Introsort's raison d'être is that it switches to Heapsort, when it detects that partitioning is tending towards quadratic behaviour of Quicksort, thus achieving a $\Theta(n \log n)$ *worst-case* asymptotic running time. Here "faster" denotes that the sum of the number of comparisons, assignments, iterator operations, and distance operations is smaller. The last-named quantities have to do with the *iterator* concept of STL.

5.2.2 Performance indicators

As exemplified by Rahman and Raman's [RR98] experiments with packed merge, due to the memory hierarchy, operation counts alone do not give a clear picture of the performance of an algorithm on a real machine. This is painstakingly evident in the work of LaMarca and Ladner [LL99] too. They compare, among other things, memory-tuned versions of Heapsort, Mergesort, and Quicksort with a "highly optimised" version of LSD Radixsort with the radix chosen to minimize cache misses (and incidentally the instruction count as well). They find, that "radix sort has the lowest instruction count, but because of its relatively poor cache performance, its overall performance is worse than the memory optimised versions of mergesort and quicksort." This conclusion demonstrates, that in order to provide a clearer picture of the relative merits of the algorithms experimented with, it is imperative to compare them based on *all* three of: real running time, number of instructions executed, and cache behaviour. I have done so.

The memory hierarchy of a computer continues beyond the caches and main memory as discussed in on p. 7. However, the latency induced by these additional levels of the hierarchy does not show in the running time reported by `clock`, cf., Section 7.4, or in the instruction count reported by `dyninst`, cf., Section 7.1.1. As such the virtual memory used by a program is considered internal by these two performance indicators.

Realising a simulator of future machines with large word lengths, which pays attention to all three performance indicators is outside the scope, at least in terms of man hours, of this thesis. I have therefore conducted experiments on a real machine with “long” words. At DIKU, the Department of Computer Science at the University of Copenhagen, Denmark, this means a Digital Personal WorkStation 500au[11] from Compaq running Digital Unix (also known as DEC OSF/1 and Compaq Tru64 Unix) version 4.0D[12]. The CPU is a 64-bit 500 MHz Alpha 21164 Microprocessor[13]. It has two on-chip data caches: an 8kb first-level cache for data only, and a 96kb second-level shared instruction and data cache. It has an 8kb first-level pure instruction cache as well. At DIKU the 500au machines are configured with an 8Mb off-chip, board-level cache and 128Mb internal working memory. The board-level cache is the bottleneck of the three cache levels in the context of accessing the input, when it consists of millions of keys. Hence, I have chosen to gather statistics of behaviour for the board-level cache exclusively. Thus, I detail the metrics for the board-level cache only; it is direct-mapped, write-back, write-allocate with 64-byte blocks. Note, that for a direct-mapped cache, the question of which replacement policy to use is void.

Although I have decided to not compare on operation counts only, Musser’s way of comparing algorithms illustrates an important prerequisite for fairness; all programs should be implemented in the same spirit. It does not make sense to compare an abstract interface oriented implementation, which employs a fair amount of indirection[14], and is intended for a general purpose library, with an implementation which only accepts, for instance, an array in its interface. Hence, STL’s `sort` has served as model for my programming of Contender and KR Sort, which has also been carried out in the `C++` programming language. Stroustrup [Str97] gives two alternatives for `sort`:

```
template<class Ran> void sort(Ran first, Ran last);
template<class Ran, class Cmp>
    void sort(Ran first, Ran last, Cmp cmp);
```

The first unconditionally uses the, possibly overloaded, less-than operator “`<`” to compare elements. That suffices for my purposes; I have passed *random access iterators* (`Rans`) of a `vector` of machine words to the two-arguments version of `sort` without overloading “`<`”. This completely eliminates any indirection overhead of element comparison in `sort`. As the input is passed to `KR_sort` and `contender` by means of two `Rans` as well, the “same

[11] `http://www.workstation.digital.com/`

[12] `http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V40D_HTML/LIBRARY.HTM`

[13] `http://ftp.digital.com/pub/Digital/info/semiconductor/literature/dsc-library.html`

[14] The slowdown caused by indirection decreases when more functions are inlined by the compiler.

spirit" condition is obeyed. It is, of course, harder to ensure, that the same amount of work has gone into implementing different algorithms, both in terms of efforts to optimise a program, and the difficulty of programming the underlying algorithm in the first place.

I have tested my programs not only on uniformly distributed data, which is sometimes the case in published research, but also on data intended to provoke worst-case performance in each of the algorithms experimented with.

5.2.3 Discussion

Note, that as decided in Section 4.2.3 I have not actually experimented with word-level parallelism (WLP), specifically packed sorting, which is the source of the theoretical improvement over comparison-based sorting in all of the research related in Section 3.6.2. This decision is supported by the fact, that the experiments of Rahman and Raman [RR98] related in Section 5.1, in my opinion, demonstrate, that packed merge cannot compete with non-packed merge for $w = 64$, if only the "standard", or "least common denominator" instruction set of modern computers is available.

Several vendors have extended this set with their own host of *single instruction multiple data* (SIMD) type instructions intended to speed up multi-media applications, cf., Yang *et al.* [YSL98]. Some of these instructions can also be useful in the context of WLP. The "Packed Compare for Greater Than" instruction of the INTEL MMX instruction set is one such example, as pointed out by Rahman and Raman. Experimenting with packed merge/sorting employing SIMD instructions is thus a potential path to follow. As the ALPHA 21164 does not provide SIMD instructions, this is not an option for me. Running packed merge on a real machine with a word length greater than 64 bits would also be justifiable, but again, I am not in a position to do so.

6 Implementation

In this chapter I describe briefly the source code listed in Section B.1 of Appendix B. The focus is on `KR_sort` and `contender`, which are both implemented in the `CVISP` namespace, cf., Sections B.1.9 and B.1.10. I have tried to heed the observations of Section 4.1 in the development of the source code. Sections 6.1 and 6.2 elaborate on my design and implementation, or rather performance engineering, of Contender and KR Sort respectively. The programs are not innovative in the sense of introducing a new idea or programming trick.

I emphasise however, that I have adopted a cache-conscious point of view, the importance of which follows from the efforts of LaMarca and Ladner [LL99], cf., Section 5.2.2, and from observing, that when millions of keys are to be sorted, the `vector` of keys will interfere in the caches with any auxiliary data structures, i.e., a bucket table, hash tables, and/or an explicit stack, as well as with itself. Unfortunately, as explained in Section 7.1.5, I have had no powerful tuning tools at hand, which *profile* cache behaviour. Therefore, the one technique for memory optimisation available to me has been mental analysis completed by simulation, cf., Section 7.1.2, of complete program runs. The coarse grain of the information thus obtained necessitates a trial-and-error method for memory tuning. Due to the time-consuming nature of such an approach, I have not been able to follow it through. There is no guarantee then, that my source code would not benefit from further memory tuning. See Lebeck and Wood [LW94] for a description of standard tricks like fusing loops, merging arrays, padding and aligning structures, packing data, and blocked processing. Note, that the heightened emphasis on the memory hierarchy does not eliminate the need for traditional, instruction-count minimising optimisations. The two should complement each other.

The programs sort word-sized integers in internal memory, i.e., I take $b = w$. I have based the implementations on tables as opposed to lists, i.e., the internal representation of the keys is a table (STL `vector`) of machine words. For general strings the one alternative cannot be said to be decisively better than the other; cf., McIlroy *et al.* [MBM93], who find that an array (of pointers to strings) does not consistently beat a linked list. However, for machine words the per-key overhead of storing them in a linked list is immediately discouraging. It seems sensible to follow LaMarca and Ladner [LL99], and as far as I can tell Rahman and Raman [RR98, RR99], and choose the table representation. In any event, the point is to implement all the algorithms using the same representation and in the same spirit in order to achieve fairness of comparisons of the programs; as mentioned in Section 5.2.2, the functions `contender` and `KR_sort`, like STL `sort`,actually take random access iterators (`Rans`), as opposed to tables in the guise of `C++` arrays, as arguments.

The source files `NMSPC.h`, `Debug.h`, `Global.h`, and `Global.cpp` are self explanatory. `isaac.hpp` and `driver.cpp` are explained in Sections 7.2 and 7.4 respectively. In general, the `C++` source code in Section B.1 is object oriented in the sense, that I have tried to, in the words of Gamma *et* *al.* [GHJV95], "encapsulate the concept that varies." This maxim is central in designing components, e.g., `C++` classes, which can be reused in a context different from the one for which they were first written.

6.1 `contender`

Below I use the terminology "group" to denote a set of input keys, which are placed in the same bucket in a Bucketsort pass, cf., Section 3.3.1. The "index" of a group is the corresponding index of the bucket in the bucket table, i.e., the value of the common bit pattern of the keys in the group occurring in the relevant (virtual) character position. I refer with "tallying" to the counting of the number of keys belonging to each group, i.e., Step 2 of Bucketsort.

`contender` implements the Contender algorithm, which is a small variation on Adaptive Radixsort, cf., Section 3.3.2; Contender prohibits using radix sizes larger than a constant, `max_radix`, for bucketing in recognition of the fact, that the bucket table will be used heavily, and must therefore preferably fit into the cache. When sorting millions of keys, $\log n$, which Adaptive Radixsort uses as radix, is at least 20, so the first Bucketsort pass needs a bucket table that occupies at least 2^{20} `struct` { `int; Ran;` }s, i.e., at least 16Mb on the WORKSTATION 500AU, twice the size of the 8Mb board-level cache. An upper bound on the radix size is therefore imperative in Contender. Based on experiments I have defined `max_radix = 9;`. This rather small value furnishes a clue to the fact, that the sizes of the on-chip caches, and not the size of the board-level one, determine `max_radix` on the WORKSTATION 500AU.

Other methods exist to alleviate the potential burden of the bucket table in Adaptive Radixsort. The most powerful is to divert to (for instance) Insertion Sort for small subproblems. Practical experience with string sorting shows, that the threshold should be chosen between 10 and 30, cf., Andersson and Nilsson [AN], who get a 50% reduction in running time, or 10 and 50, cf., McIlroy *et al.* [MBM93], for satisfactory results. `contender` switches to STL `__insertion_sort` on groups of size at most `__stl_threshold`. These identifiers are not part of the STL specification, but they are defined and used internally by the implementation of STL, which comes with the compiler I have used, cf., Section 7.1.4.

There is also the possibility of maintaining in sorted order the indexes of non-empty groups during each Bucketsort invocation in order to speed up bucket table traversals. The *span heuristic* does this partially; the minimum and maximum group index is maintained during tallying, and in Step 3 of Bucketsort only buckets in the corresponding subrange are visited. The overhead of the span heuristic is small enough to not slow down Adaptive Radixsort on uniformly distributed data, for which few empty buckets can be expected. Contender uses the span heuristic.

According to McIlroy *et al.* [MBM93] more intricate, but yet standard, methods for maintaining in sorted order the non-empty groups do not have this property. The more intricate methods typically slow down performance on random data by a factor between 1/2 and 1/3, whereas the span heuristic lends a 10% improvement to the hybrid algorithm, which switches to Insertion Sort. These results are measured in the context of string sorting, but must be valid for integer sorting as well. A simple argument that supports this statement is, that the cheaper it is to inspect and move the entities to be sorted — and machine words are at least as cheap as anything else in this respect — the larger the impact, that overhead has on overall performance.

The question is, if the implementation should be specialised to deal with input, which is distributed in a certain way; from knowledge of the distribution it might be possible to derive some instance-type-specific way of not visiting empty buckets. The answer is clearly "no", as `contender` is to be compared with STL `sort`, which is a general-purpose library routine.

It is important to avoid reallocating the bucket table for each "recursive" invocation of Bucketsort. An explicit stack is therefore necessary to make `contender` iterative. A stack record represents a group to split, and must therefore contain a range (two `Rans`), and the position (an `uchar`) in the keys in the group of the most significant bit, which has not yet been inspected as part of a virtual character. If all three components are gathered in a `struct`, the compiler uses excessive padding, as can be verified with `sizeof`. I have therefore used two parallel stacks. I have not used the strategy of splitting the largest group resulting from a Bucketsort pass last, even though that might reduce the maximal size of the explicit stack.

I use in-place permutation, cf., Knuth [Knu98, Solution 5.2-13] and Section 3.3.1, because it is nicer cache wise, and has a smaller memory footprint, than a two-table method for permutation. The in-place choice means however, that `contender` is not stable.

6.2 `KR_sort`

As is evident from the fact, that the vast majority of the source code implements KR Sort, this algorithm was much harder to program than Contender. `KR_sort` uses `contender` to sort short keys. Thus, it remains for me to describe how I have implemented exponential range redex, cf., Section 3.5.1. With $w = 64$, it is infeasible to allocate tables of size $2^{w/2}$; thus sparse tables are needed. In the namespace `Sparse_table`, cf., Sections B.1.5 and B.1.6, I have implemented hashing. The most interesting class in `Sparse_table` is the abstract class `Creator_interface`, which implements the Factory Method *design pattern* described by Gamma *et al.* [GHJV95]. Subclasses of `Creator_interface` must implement a *policy* for choosing an implementation of the abstract concept of sparse tables. In the context of exponential range redex the intention is, that hash tables should be chosen for large universe sizes ($\log|\mathcal{U}| >$ `sep_width`), whereas a straightforward, directly allocated table suffices for small ranges. My own implementation of hash tables based on $\mathcal{H}_{k,\ell}$ from Section 2.4 is chosen for large universes by the subclass `Own_impl_creator`, whereas the STL implementation of hash tables, namely `hash` and `hash_map`, is chosen for this purpose by the subclass `STL_impl_creator`. Both these subclasses choose a simple array for small universe sizes. STL hashing turned out to be faster than my own implementation, although only by a small margin.

STL_impl_creator is therefore used in KR_sort as template, policy argument in the instantiation of the class KR, which is a subclass of the abstract class RR_interface, where "KR" stands for Kirkpatrick and Reisch, and "RR" abbreviates "range redex". Both these classes belong to the Range_redex namespace, cf., Sections B.1.7 and B.1.8. KR implements exponential range redex as suggested by Rahman and Raman [RR98]; this approach resembles MSD Radixsort with exponentially decreasing virtual character lengths $c = w/2, w/4, w/8, \dots$, but the input is only permuted during the range expansion, into increasingly larger groups, which is facilitated by using *promotion histories*. KR::sort takes an argument RR_steps, which determines the number of redex levels. KR_sort uses 1, the minimum meaningful value, for RR_steps, as one level was experimentally found to be optimal; this alludes to the fact, that exponential range redex does not perform well in practice, cf., Section 7.5.1. Note, that KR_sort is not an in-place program, as KR uses a an auxiliary vector of size n.

7 Experiments

In this chapter I detail the design, execution and results of the empirical evaluation of the `contender` and `KR_sort` programs, cf., Chapter 6. Section 7.1 concerns tools for development and measuring performance indicators. Section 7.2 deals with the choice of a random number generator for constructing test instances of uniformly distributed keys. The actual test suite of sorting problem instances, which comprises instances designed to provoke worst-case behaviour in each of the programs, is presented in Section 7.3. The program runs are documented in Section 7.4, and the results are discussed in Section 7.5.

7.1 Tools

I have used the ATOM *instrumentation* toolkit from DIGITAL. Instrumenting a program means, that the object code is modified. In the context of tools for measuring performance that typically means, that instructions, which collect and process information on the performance, are added. ATOM facilitates creation of *custom* tools for instrumenting applications as described by Srivastava and Eustace [SE94]. They also explain how ATOM achieves "pristine behaviour", which means, that instrumentation does not change data addresses in the program; stack and heap addresses are the same as before. Naturally, instrumentation alters instruction layout, but if an analysis routine of an instrumentation tool created with ATOM needs a program counter value, the original one can be provided. See Section 9.2.6 of the DIGITAL UNIX Programmer's Guide[15] for an outline explanation of how to ensure this. Hence, instrumentation with ATOM can truthfully record or analyse both data and instruction references.

In this section I first touch on some ready-to-use instrumentation tools, which are supplied with ATOM, see Section 7.1.1. I then, in Section 7.1.2, describe NEXT-GEN, an instrumentation tool for cache *simulation* I have created with ATOM. This is followed by Section 7.1.3, which describes some powerful tools for *profiling* the memory usage and cache behaviour of programs. As it will appear, I have unfortunately been precluded from using them. Section 7.1.4 briefly summarises the development tools I have used. In Section 7.1.5 I discuss the value of the various tools.

7.1.1 Prepackaged atom tools

The ATOM toolkit, which is distributed with DIGITAL UNIX (the command line `man atom` brings up the manual page), also consists of a set of *prepackaged* tools. In the following short description and discussion of the relevant subset of those tools, text within "quotation marks" is quoted directly from the `man` pages:

`dyninst` "Provides fundamental dynamic counts of instructions, loads, stores,

[15] `http://www.unix.digital.com/faqs/publications/base_doc/DOCUMENTATION/V40D_HTML/APS30DTE/TITLE.HTM`

blocks, and procedures." I report the count of instructions, one of the three performance indicator chosen in Section 5.2.2, directly.

`hiprof` "Produces a flat profile of an application that shows the execution time spent in a given procedure, and a hierarchical profile that shows the execution time spent in a given procedure and all its descendents." Here "time" is per default actually number of instructions. Note, that `hiprof` does not use sampling. The profiles are exact and reproducible.

`gprof` "Displays pc-sampling or `hiprof` call graph profile data." Running a `hiprof`-instrumented program produces as a side-effect a profile data file. `gprof` generates from such a file the final report in plain text format suitable for inspection by human beings. Note, that `gprof` alternatively takes as input profile data files resulting from running programs compiled with the `-pg` switch. In that case, real timings assessed by pc-sampling, as opposed to exact instruction counts, are reported in the output. This is analogous to using `hiprof` in its non-default, CPU cycles counting mode (`-cputime` switch). `gprof` can also summarise profile data to "accumulate profile data across several runs."

See Section 7.1.5 for a discussion of the value of `hiprof`.

7.1.2 Next-Gen

This section is about measuring the number of cache misses, which is one of the three performance indicators chosen in Section 5.2.2.

Spork's trace-based CPROF cache simulator Spork [Spo99] has written a cache *simulator* CPROF[16], which I have decided to modify. Do not confuse it with the cache *profiler* by Lebeck and Wood [LW94], cf., Section 7.1.3, which is also called CPROF. The simulator takes as input a plain text file *reference trace* obeying a certain syntax, and supports customisation of cache metrics and replacement policy on the command line, where it can also be specified, whether the simulation result should be output in plain text or LaTeX format. The simulator categorises a reference as a hit or a *cold start*, *capacity*, or *conflict* miss for reads and writes separately. Cf., Spork [Spo99] for a definition of those terms and example output from the simulator.

CPROF comes with a supplementary tool `gpconvert`, which constructs from an address trace (should be relatively short) an illustration of the reference pattern in GNUPLOT[17] input format.

The CPROF framework also includes facilities for *constructing* the reference trace of `C++` programs, namely template functions `READ` and `WRITE`

[16]Source code is available under the GNU Public License and can be downloaded from `http://www.diku.dk/students/halgrim/`.

[17]In spite of the name, the GNUPLOT utility is not related to the Free Software Foundation. It is covered by its own copyright statement, but freely distributed.

contained in a header file. These functions return their first argument after logging a reference to it in the trace file. They are supposed to be inserted judiciously in ones source code. In the context of a declaration `int table[32];` for instance, the assignment `table[0] = table[31];` should be replaced by `WRITE(table[0],READ(table[31]));`. In a loop `for (int i = 0; i < n; i++)` on the other hand, there is no need to decorate the use of `i`, as this variable will probably be kept in a register.

On-the-fly cache simulation As the use of `READ` and `WRITE` is both cumbersome and error prone, I have decided to integrate `cache` from ATOM with Spork's CPROF simulator. As supplied with ATOM, `cache` "simulates execution of the application in 8KB direct-mapped cache." This simulation is performed in an *on-the-fly* fashion; the state of the simulated cache is maintained in main memory by the instrumented program during its execution, and the simulation result is output before termination. Source code for `cache` is listed in Section 9.2.7.3 of the DIGITAL UNIX Programmer's Guide. By interfacing the trace analysis part of CPROF, which provides the possibility of simulating a cache with arbitrarily chosen metrics, with the facilities of `trace` for intercepting the memory accesses of a program, I have obtained a new tool, NEXT-GEN, which combines the advantages of the two:

- There is no need for manually deciding, à la `READ` and `WRITE`, which memory references to include in the simulation. This is done automatically by means of the instrumentation procedures, inherited from `cache`, defining the new tool. ATOM provides the `IsInstType` routine to determine, if an instruction is a certain kind of instruction. The types relevant in this context are `InstTypeMem`, any instruction that accesses main memory, and `InstTypeLoad` and `InstTypeStore`, integer or floating-point load and store instruction. NEXT-GEN includes the addresses accessed by any of the two last-mentioned instruction types in the cache simulation.

- Simulation is performed on-the-fly. Hence, no intermediate trace file is needed. This is preferable, because literally in the order of billions of memory references can be made in a couple of seconds by today's computers, so the mere size of the file would be a bother, and it would take an inordinate amount of time for I/O to generate and subsequently analyse it.

- The tool can be customised to simulate any cache metrics. Choice of replacement policy is also supported.

- It is not dependent on the programming language in which the source code is written.

The only drawback is, that the instrumentation and analysis procedures defining NEXT-GEN must be written in standard `C`, as ATOM does not support other programming languages for this purpose. Thus, the reuse of Spork's CPROF source code is not mere cut-and-paste, as it is authored in

C++. The name NEXT-GEN ("next generation") might seem pretentious when comparing the tool with existing ones, cf., Section 7.1.3, for investigating the memory behaviour of programs, as NEXT-GEN *is* merely a cache simulator. However, the name is just intended to signal the fact, that NEXT-GEN builds on `cache` and Spork's CPROF. The NEXT-GEN source code is listed and briefly introduced in Section B.2 of Appendix B.

Optimising Next-Gen I am interested in simulating a specific cache, namely the board-level, direct-mapped cache of the WORKSTATION 500AU, in the context of sorting millions of integers. For reasons of efficiency of simulation I therefore chose to prune away CPROF's possibility for simulating different replacement policies, as the victim block is deterministically dependent on the referenced address in a direct-mapped cache. This choice has several advantages:

- It obliterates the need for a random number generator (RNG) for potentially simulating random replacement. A good RNG does not come easy, as discussed below in Section 7.2. In the context of this thesis nothing is gained, because a RNG is needed anyway to generate test instances. In general, independence of a RNG is a potential advantage though.
- The choice reduces the overhead of simulation in several ways. First, fewer instructions are needed to effect the simulation of each memory reference, since 1) it is unnecessary to check which policy is being simulated using for instance a `switch` statement, 2) it requires no effort to choose, if needed, the block to replace, and 3) it is easier to categorise a miss, because only two types, *compulsory* and *replacement*, exist, as the concepts of conflict and capacity misses have no meaning for direct-mapped caches. To grasp the importance of optimising the method for simulating each memory access remember, that the method will be used billions of times. Second, the delimiting of the simulator's scope to direct-mapped caches reduces its memory footprint, as information, which facilitates simulating cache associativity, does not need to be maintained for each simulated cache block.

I made yet an optimisation, a trivial but powerful one. For each reference the simulator needs to extract from the address its (*offset*,) *index* and *tag*, cf., the function `reference` in Section B.2.2. Constraining the cache capacity and block size, and the word size of the machine w to be powers of two means, that the extraction can be done by shifting and masking bits instead of using integer division and remainder calculations.

When simulating the WORKSTATION 500AU cache, the optimised version for direct-mapped caches beats by a factor 5.5 (approximately) its predecessor version, which used integer arithmetic and, pointlessly for the WORKSTATION 500AU cache, simulated a replacement policy, i.e., random replacement.

Discussion and remarks Alternatively one could use `trace` from ATOM. `trace` "generates an address trace, logs the effective address of every load and store operation, and logs the address of the start of every basic block as it is executed." Use of `trace` would imply the need for writing a utility to convert `trace` output format to CPROF input format. In light of the discussion above, this is obviously not favourable. Note however, that `trace` provides the opportunity of simulating, to some extent, instruction as well as data caching. It would require some effort to incorporate simulation of instruction caching into NEXT-GEN, as the `cache` source code does not handle providing original program counter values, cf., Section 7.1.

On-the-fly, trace-driven simulation of a memory hierarchy requires that either:

1. A call of a simulator analysis procedure is made for each reference in order to process it. This induces a sizeable overhead of saving and restoring *caller-save* registers.

2. Or, references are buffered, and a processing procedure is called only when the buffer fills up. This reduces the calling overhead at the price of buffer management and deterioration of memory system performance (of the simulation) due to cache pollution.

ATOM adopts the first alternative, but recognises the switch `-A1`, which instructs ATOM to "optimise the calls to analysis routines by reducing the number of registers that need to be restored. Specifying this switch causes Atom to generate slightly larger, but faster code."

7.1.3 Alternative memory tools

I have also looked into other tools for investigating the memory behaviour of programs. I emphasise that "profiling" means explaining, in some way, the cache performance encountered. Compare this with a simulator, which merely reports the performance. Traditionally, profilers have pointed out sections of the source code, which induce many cache misses.

MemSpy The MEMSPY cache profiler by Martonosi *et al.* [MGA95, MGA92] has the nice property of offering data-structure oriented information as well; it also associates specific variables in the source code with bad cache performance. MEMSPY categorises cache misses (as *first reference*, *replacement*, or *invalidation*; the latter type is only meaningful in the setting of multitasking). *Hit bypassing* is used to alleviate the burden of calls to profiler analysis procedures; the cache-hit check is embedded into the saving on the stack of registers, which enables short-circuiting the saving, if the reference is a hit. The rationale is, that the trivial case of a hit by far accounts for the majority of lookups, hit ratios of 90% or more are common. Furthermore, *trace sampling* is employed to increase performance (at the cost of accuracy). However, MEMSPY is build on TANGOLITE, cf., Goldsmith [Gol93], which according to Srivastava and Eustace [SE94] "change the application program's heap addresses." Also, according to Martonosi, MEMSPY is not being maintained or developed. Thus, I decided to not try and port it to the WORKSTATION 500AU. Note, that TANGOLITE is based on instrumenting assembly code. So, like ATOM, it is tied to a particular CPU instruction set, but is not conceptually tied to any particular high-level programming language.

The CPROF cache profiler Lebeck and Wood [LW94] are responsible for a cache profiler, which is also both data-structure and source-code oriented. It was developed independently from MEMSPY, and, like the simulator of Spork [Spo99], it is called CPROF. Analogously with its namesake, the profiler categorises cache misses as *compulsory*, *capacity*, or *conflict*. It is finer grained than MEMSPY, in that it annotates source code at the line, as opposed to procedure, level. The CPROF profiler is built on and distributed with FAST-CACHE[18], which is an implementation of the *active memory* abstraction for memory system simulation by Lebeck and Wood [LW97]. FAST-CACHE in turn belongs to the Wisconsin Architectural Research Tool Set (WARTS[19]). As it is based on editing executables — by means of EEL by Larus and Schnarr [LS95], also part of WARTS — the FAST-CACHE tool, and hence the CPROF profiler, like ATOM and MEMSPY, is independent of the high-level language, that programs to be investigated are written in. However, FAST-CACHE only runs on SPARC processors under SOLARIS 2.5.1 or older, so I cannot use it.

For the sake of completeness I mention that active memory [LW97] is an effort to reduce the profiling overhead; contrary to ATOM, the call of an analysis procedure in the no-action, cache-hit case is avoided altogether with FAST-CACHE. And, in comparison to MEMSPY, this bypassing is designed in a more clean fashion. Lebeck and Wood state, that FAST-CACHE is "two to three times faster than published numbers for highly optimised trace-driven simulators," referring to Srivastava and Eustace's [SE94] ATOM.

7.1.4 Development tools

The source code has been compiled with EGCS[20] (pronounced "eggs") version 2.91.66 (release 1.1.2 03-14-1999), which is an experimental step in the development of GCC, the GNU compiler. One immediately notable difference between the two is, that EGCS supports `C++` namespaces, whereas GCC does not. There are more profound advantages to EGCS over GCC however. Among them, is better internal compiler architecture. Like the GCC sources all EGCS source code is assigned to the FREE SOFTWARE FOUNDATION. The implementation of the `C++` STL (version 3.2) accompanying the EGCS release I have used, is © 1994 HEWLETT-PACKARD COMPANY and © 1996, 1997 SILICON GRAPHICS COMPUTER SYSTEMS, INC. The EGCS release was build, or rather bootstrapped, with GCC version 2.8.1, and was in turn itself used to build version 4.18 of GDB, the GNU debugger, which has been used for debugging. I also tried out DDD, the DATA DISPLAY DEBUGGER

[18] `http://www.cs.duke.edu/~alvy/#software`
[19] `http://www.cs.wisc.edu/~larus/warts.html`
[20] `http://egcs.cygnus.com/`

(version 3.1.4), which is a graphical user interface to GDB. However, it ran too slowly, so lack of patience prevented me from using it.

7.1.5 Discussion and lessons learned

The NEXT-GEN *simulator* is not as powerful a tool as either the MEMSPY or the CPROF/FAST-CACHE *profiler* for memory-tuning purposes. As stated above, the reason is, that NEXT-GEN only reports the cache performance, and does not give an indication of the cause of it. At least procedure-level granularity is needed of tools, if memory tuning is to be performed based on them. And, as observed by Martonosi *et al.* [MGA95], and Lebeck and Wood [LW94], data-oriented profiles are valuable as well. Within the context of this thesis, I cannot undertake to write a better tool than NEXT-GEN for performance engineering of programs. I have therefore relied on using procedure-level execution times (`hiprof -cputime` or `gcc -pg` with `gprof`) and procedure-level instruction counts (`hiprof`, no switch) in conjunction; a memory bottleneck can be identified as a section of code, which should execute fast in a perfect memory model, but runs slowly. This approach is *very* time consuming, if it is to be carried all the way through, which, accordingly, I have not been able to. `pixie`, another ATOM tool, provides statistics on the basic-block level, but is too confusing.

The EGCS compiler has been an interesting acquaintance to say the least. On several occasions it terminated with an "internal compiler error" message of little use to me. The first time it happened, I wasted valuable time building a newer version of EGCS, which was not an official release. This version did not have the bug, but unfortunately it did not support debugging (`-g` switch). I finally reverted to release 1.1.2, and spent some time isolating the, perfectly standard, `C++` language construct, which caused the internal compiler error. I then settled for a workaround rewrite of the relevant source code section. The subsequent times, I got an internal compiler error, I used this remedy immediately.

There is hope, however. EGCS *is* being actively maintained. In fact, in April 1999, the EGCS steering committee was appointed official GNU maintainer for GCC, and GCC was renamed to the GNU COMPILER COLLECTION. The research community must, in my opinion, support the ongoing development by conscientiously submitting bug reports, if not financially with donations.

ATOM has been easy to use and very helpful in the development of NEXT-GEN.

7.2 Random number generator

The `C` STANDARD LIBRARY (`libc.a`, cf., `man random`) provides a random number generator (RNG) `int random(void)`, which generates values in the range $\{0, \ldots, 2^{31} - 1\}$. The default period of `random` is "approximately $16(2^{31}-1)$", but by explicitly requesting it to use the maximum 256 bytes of *state information*, a period of 2^{69} can be guaranteed. A length 64 bit pattern can, of course, be generated by three invocations of `random`, as together they produce a total of $3 \cdot 31 = 93$ random bits. Is this satisfactory? The `man` page for `random` answers this question in the affirmative: "All bits generated by `random` are usable. For example, `random()&01` produces a random binary value." However, the RNG community strongly advices against using "the built-in generator of your system." Since I was not immediately in a position to reliably assess the validity of this statement, I decided to play it safe and look for another RNG than `random`.

This lead me to Jenkins's [Jen96] ISAAC, which is an abbreviation of "Indirection, Shift, Accumulate, Add, and Count." Alternatives exist, cf., for instance, the Internet[21]. I chose ISAAC, because a `C++` implementation by Jackson[22] invites confidence in itself, and is recommended by Jenkins himself. For convenience I list Jackson's source code (`isaac.hpp`) in Section B.1.12.

ISAAC requires 19 instructions of the types used to form its name to produce a 64-bit result. ISAAC's period depends on an integer `ALPHA`. With `ALPHA` = 8, when generating 64-bit values, all cycles are at least 2^{72} long, and the average cycle length is 2^{16583}. Here, the minimum and average is taken over all "seeds". A seed in turn is actually an array of $2^{\texttt{ALPHA}}$ values of `ALPHA` bits each. The results are uniformly distributed, unbiased, and unpredictable unless you know the initial state. Jackson reports, that his implementation passed George Marsaglia's DIEHARD test.

ISAAC certainly suffices for my purposes. In fact, since it is estimated, that the known (astronomical, as in "Andromeda") universe contains 2^{420} protons, ISAAC should be good enough in any context, where random numbers are needed.

In hindsight, I should not have heeded the warning of the RNG community against using the "built-in generator". Having looked a little into the literature on the subject, I would accept `random`, since it "uses a nonlinear additive feedback random number generator employing a [...] state array." Hence, it takes the same approach as ISAAC, i.e., it uses state information which, as the generation proceeds, is modified in a nonlinear fashion, and without using multiplication. However, I would not have been at ease with accepting `random` *without* looking beyond the "no built-in" warning, and reading about the state-of-the-art in generation of random numbers. Accordingly, having went to the trouble of acquiring the knowledge of ISAAC, I decided to use it. ISAAC does not interfere with `random`, so the two can be used in parallel.

7.3 Test suite

Instance sizes I have tested my programs on problem instances of size 2^p, where $p \in \{10, \ldots, 23\}$. Sorting few, i.e., less than 2^{20} numbers, is done satisfactorily by existing programs, e.g., STL `sort`. Therefore, the behaviour of `KR_sort` and `contender` is not interesting for $n < 2^{20}$. In fact, striving for a decisively faster, general-purpose internal sorting algorithm for $n < 2^{20}$, or for even larger instances, is in my opinion futile. I doubt, if

[21] `http://www.taygeta.com/random.html`
[22] `http://www.qtj.net/ quinn/computer_science/PatternsOfRandomness.html`

improvements can be obtained, which justify the effort necessary to obtain them; I elaborate on this view in Chapter 8. The question for me therefore was, how little values to bother to use for p. I decided on 10 as lower bound, because `sort` is faster than `contender` for $p \leq 10$, but slower for (at least) $10 < p \leq 23$, on ISAAC input. Admittedly a somewhat arbitrary argument. The upper bound of $p = 23$ was forced upon me though. At DIKU a process running on the WorkStation 500au can maximally allocate 128Mb of data memory and 2Mb of stack space. A key/word is 8 bytes, hence there is room for at most 2^{24} keys. This leaves no memory to use for auxiliary data structures, however. I therefore settled with $n = 2^{23}$ as the highest power of two to test with.

Which classes of instances to use? The use of "random", meaning uniformly distributed, data alone to test sorting programs is not uncommon, cf., [RR98, RR99, LL99]. This obviously does not provide a detailed picture of the robustness, or lack thereof, of a sorting program. What kinds of input should one additionally test with then? There is no easy answer to the question: "which kinds of (integer) data generally need to be sorted?" To the best of my knowledge, a standard benchmark for internal sorting does not exist. What I am advertising for here, is something similar to the MinuteSort benchmark for disk-to-disk sorting introduced by Nyberg *et al.* [NBC+94, AlphaSort], and since used by Agarwal [Aga96] and Arpaci-Dusseau *et al.* [ADADC+98, NOW-Sort]. The MinuteSort benchmark calls for sorting as many 100 byte records (satellite data) with 10 byte "random" keys as possible within one minute. It has replaced the Datamation benchmark, cf., Anonymous [Ano85], which calls for sorting 1 million such records as fast as possible, which has already been solved to satisfaction.

Since a standard benchmark does not exists, I have decided to follow e.g., Bentley and McIlroy [BM93], Musser [Mus97], and Andersson and Nilsson [AN], and test with data intended to provoke worst-case behaviour in each of the algorithms/programs experimented with, as well as with "random" data.

Worst-case input The worst-case input for Contender is fragmented data, i.e., data with lots of repetitions of the same key values, because all bits of n' equal-valued keys have to be inspected, if $n' \geq n_0$, where n_0 is the Insertion Sort breakpoint. I have used *skewed* length n sequences adhering to the following template:

$$0, 1, 2, 2, 3, 3, 3, 3, \ldots, \underbrace{\log n, \ldots, \log n}_{n/2} \tag{8}$$

To bring out worst-case performance in `sort`, I have used the *median-of-three-killer* (Mo3K) sequences reported in [Mus97]. Recall, that `sort` implements Introsort, cf., Section 5.2.1, which uses a Quicksort-like partitioning step around the median of three key values. Specifically, those with indices 1, $k+1$, and $2k$ in a length $2k$ sequence. A Mo3K sequence of length

$2k$ is therefore defined as:

$$1, k+1, 3, k+3, 5, \ldots, 2k-3, k-1, 2k-1, 2, 4, 6, \ldots, 2k-2, 2k \qquad (9)$$

The partitioning value for this sequence is 2, and the partitioning results in the sequence $\langle 1, 2 \rangle$ and a sequence of length $2k-2$ with a Mo3K-structure. Musser [Mus97] shows a $\Theta(n \log n)$ running time for Introsort on Mo3K input of length n.

As it happens, the worst-case for KR Sort, or rather the `KR_sort` program using hashing, is uniformly distributed data. The exponential range reduction from, e.g., w to $w/2$, groups keys on the most significant $w/2$ bits. A hash-table entry corresponds to each such group. When the keys are uniformly distributed, the expected number of groups is maximised, and the memory demand is most severe. $2^{64/2}$ is approximately 4 billions. Uniformly distributed 64-bit keys will therefore probably fall in singleton groups when bucketed on their high-order bits, when $n \leq 2^{23}$.

Expected behaviour Note, that Contender can be expected to do bad on Mo3K sequences as well, since even for the largest instance size $n = 2^{23}$, the $64 - \log n = 41$ most significant bits of each key will be zero. Contender behaves best for random data.

STL `sort` ought to do well on skewed data, since the median-of-three partitioning of a skewed sequence of length n results in a skewed sequence of length $n/2$ and a sequence of length $n/2$ of keys with the value $\log n$. `sort` can be expected to do well on random data too, as the median-of-three is a strong partitioning heuristic in this case; it can be expected to split the input sequence in halves.

`KR_sort` will approach its best-case behaviour on skewed data, as the number of hash-table entries will be low. Analogous remarks apply to Mo3K sequences.

Thus, note the pleasant bonus, that the test suite brings out best-case behaviour in each of the programs as well. Furthermore, appreciate that I have explicitly described, cf., Section 7.2, the random number generator, namely ISAAC, which I use to generate "random" keys with. People sometimes forget this, cf., [BM93, Mus97, RR98, RR99, LL99].

7.4 Test bed

All three contestants are fast enough, that a single run on an instance of size 2^{10} of any of the three sequence types is measured by `clock` from the `<ctime>` header to take no time at all. Therefore, I have timed the average running time of repeated runs for small values of p. To be exact, the test bed (`driver.cpp`, cf., Section B.1.11) allocates a `vector` named `keys` of 2^{23} 64-bit machine words. It (conceptually) divides this `vector` into 2^{23-p} *pieces* of size 2^p each. Every piece is then initialised as a sequence of length 2^p of the requested type, i.e., ISAAC random, Mo3K, or skewed. For the two latter types, all pieces are identical afterwards. This is not true for random input, because ISAAC is only seeded once (same seed every time).

The average *sorting* time on the pieces is reported; it includes neither the generation of the input, nor the verification (STL `is_sorted`) of the output.

Because I can only use the WorkStation 500au in a non-dedicated fashion, the sorting time average actually fluctuates a little between different invocations, using the same choice for p and the instance class (a configuration), of the driver. According to Kernighan and Ritchie [KR88] "`clock` returns the processor time used by the program since the beginning of execution, or -1 if unavailable." Hence, the fluctuation is not directly caused by CPU contention; the time a program spends waiting to use the CPU is not included in the time reported by `clock`. However, the multi-tasking pollutes the cache, and is thus indirectly responsible for the fluctuations in running time. My solution is to run each configuration three times (at night) and report the average of the three averages thus obtained. The variation of the three averages was small enough to convince me of the sufficiency of this solution. I have no statistical argument to support this solution.

Note, that Next-Gen simulates a dedicated environment, i.e., no multi-tasking. Hence, the result of simulations does not vary from time to time. The Next-Gen-instrumented version of the test bed accordingly needs to be run only once for each configuration. The number of cache misses reported corresponds to the *complete* execution of the test, not just the sorting. For each configuration I therefore simulated a dummy test, which generates the requested type of input, but calls a dummy sorting routine on each piece. The resulting dummy number of cache misses was subtracted from the number of cache misses reported for each of `contender`, `sort`, and `KR_sort` for the same configuration. Analogous remarks apply to the dynamic instruction counts obtained with the `dyninst`-instrumented version of the test bed.

The `hiprof` tool, which comes prepackaged with ATOM, supports a `-sigdump <sig>` switch, which "causes the process running the instrumented application to catch the signal indicated by `<sig>` [...]. When it receives that signal, the process writes the current profiling data to the output file, reinitialises the profile by setting the execution time to zero, and resumes execution." Something similar could have been done with Next-Gen to avoid the dummy tests, but I did not have time to look into it. Alternatively, the complete *running*, as opposed to sorting, time could have been measured by `clock`, and the overhead of producing the input and verifying the sorting could have been included in all the three performance indicators I measure.

7.5 Results and discussion

Different research teams more often than not carry out experiments on different machines. Whereas this fact does not completely invalidate cross-study comparisons, it certainly requires such comparisons to be conscious of the exact differences between the machines involved. Below I compare my findings with:

- LaMarca and Ladner [LL99] who use a 266MHz DEC AlphaStation 250 (Alpha 21064A chip) with a 2Mb second-level, on-board, direct-mapped cache with 32-byte blocks[23].

- Rahman and Raman [RR99] who use a SUN ULTRASPARC-II with a 512kb second-level, on-board, direct-mapped, write-back cache with 64-byte blocks[24]. Note that this study is different from the one [RR98] by the same authors related in Section 5.1.

Note, that these studies concentrate on the *second-level*, off-chip cache of the machine used. For the WORKSTATION 500AU, which I have used, the 8Mb off-chip cache is third level.

Rahman and Raman [RR99] do not report a clock-frequency for the ULTRASPARC-II; I think it is 400MHz. Never mind that, I do not compare absolute values of real running times anyway, as the 500MHz WORKSTATION 500AU I have used has a different clock-frequency than the ALPHASTATION 250, and more importantly, simply because the machines are not the same.

The three machines are all of the $w = 64$ *reduced instruction set computer* (RISC) variety, however. I argue, that this allows me to compare instruction counts, even though the exact CPU designs in terms of, e.g., pipelining differ.

I compare numbers of cache misses as well which, I am aware, sounds unreasonable on the face of it. My justification is as follows; even though the board-level caches of the three machines have different sizes, it is still interesting to see what happens to the number of cache misses per key, when the instance size increases beyond the cache size. I use the term *cache limit* for the value of n when this happens. The cache limit is smaller for the ALPHASTATION 250, than for the ULTRASPARC-II, for which the cache limit in turn is smaller still than for the WORKSTATION 500AU. Still, the behaviour on each machine at the cache limit can be cross compared.

The validity of cross-study comparisons is also determined by, among other things, differences in compiler, operating system, and machine architecture in general. The amount of work put into different implementations (by different researchers) of an algorithm experimented with is also significant.

NEXT-GEN provides counts of dirty and clean cache misses. This is interesting, since a dirty miss takes longer to service than a clean miss. I do not report these counts though, as none of the two studies mentioned above do.

Be aware, that I do not make comparisons with the study of Rahman and Raman [RR98], which is summarised in Section 5.1. They do not report numbers of cache misses, and their "operation counts" are not comparable with instruction counts; one "operation" is performed by several instructions, and they do not give enough information for me to determine, how many "several" is. Furthermore, their "KR sort" uses LSD Radixsort to sort the reduced keys, whereas my KR Sort uses Contender, which is MSD oriented.

[23] `http://www.compaq.com/products/workstations/digital/retired/a250series.html`
[24] `http://www.sun.com/microelectronics/UltraSPARC-II/`

7.5.1 Plots

Appendix A contains the plots I have made of the test results, and a short introduction to them. The plots are discussed below. Recall my expectations for the behaviour of the algorithms/programs on each type of input, cf., Section 7.3. I will not reiterate those expectations here, nor emphasise when they are confirmed by the results. Rather, I focus on surprising behaviour and comparison of the programs with each other.

The sorting time plots, i.e., Figures 7–12, demonstrate, that all three programs run in nearly linear time on all three input classes. This is somewhat surprising. I expected the worst cases to be worse.

`contender` is on average approximately 25% faster than `sort` on uniformly distributed data. On skewed and Mo3K input the two programs compare as expected. `KR_sort` is basically hopeless. It does beat `sort` for $p = 19, 20, 21$ on Mo3K data, but this is no accomplishment, especially since `KR_sort` runs out of memory for $p = 22, 23$ on Mo3K data, and is consistently by `contender` (on all instance classes). Note, that the best case for `sort` turns out to be skewed data.

The cache misses plots, i.e., Figures 13–18, give, among other things, an indication of why `KR_sort` fares so badly. `contender` and `sort` both give rise to a constant, approximately 0.25, number of cache misses per key for $p \leq 20$. These are compulsory misses. Only for $p > 20$ do replacement misses begin to show. This is expected; the board-level cache of the WorkStation 500au has room for 2^{20} keys, and `contender` and `sort` are both in-place programs. For `KR_sort` however, the measured value for the cache limit is $n = 2^{15}$, due to the space needed for the hash tables and an auxiliary `vector` of size n, cf., Section 6.2.

Furthermore, the number of cache misses per key does not level out for `KR_sort` for some $p > 15$ and on, but rises fast. For `contender` on the other hand, the number of replacement misses seems to settle; on a different number for each class of input, for isaac data only a small change occur, a rise to 0.50. For `sort`, the number of replacement misses keeps increasing, slowly, for $p > 20$ on skewed and isaac data. This is *not* true for `sort` on Mo3K sequences, for which the increase is quite drastic, which has impact on the running time. The drastic increase is attributable to Heapsort, which Introsort switches to; cf., LaMarca and Ladner [LL99], who illustrate how Heapsort thrashes the cache when the cache limit is reached.

The instruction count plots, i.e., Figures 19–24, further support, that `KR_sort` is no good. It uses an enormous number of instructions on all three types of input compared with the other two programs. This is most vividly illustrated by the input-class-oriented plots. Analogously with the sorting-time results, `KR_sort` uses fewer instructions than `sort` for $p = 19, 20, 21$. `contender` more or less uses a constant number of instructions per key on Mo3K *and* skewed input. On isaac data the number fluctuates within a band of width 100. For `sort` the number of instructions increases, fast for Mo3K sequences, and slowly otherwise. This is analogous with `sort`s cache behaviour, and explains why the program runs as slow as it does on skewed input.

Do I think Figures 16 and 22 provide evidence, that `contender` will keep beating `sort` for $p > 23$ on ISAAC, or more generally "random", input? The results do not enable me to answer this question definitively, as I next elaborate. I do think the programs will continue to show a linear running time, since their performance beyond the crucial cache limit does that. The small gap of 25% does not, however, preclude that `sort` will at some point start to consistently beat `contender`.

7.5.2 Cross-study comparisons

LaMarca and Ladner LaMarca and Ladner experiment with many programs on "an array (4,000 to 4,096,000 keys) of 64-bit integers chosen uniformly at random." This is comparable to ISAAC data. See [LL99] for details. The two programs, that are interesting for my purposes, are 1) a LSD Radixsort implementation, cf., Section 5.2.2, and 2) a "memory-tuned" Quicksort program, which switches to Insertion Sort on small subproblems. The Quicksort implementation behaves very much like I have found `sort` (Introsort) to do, to the point of the number of cache misses and instructions being comparable on absolute values. Remember, that LaMarca and Ladner use ATOM and an ALPHA-family based machine as well.

Therefore, I take the liberty of comparing their LSD Radixsort implementation with `contender` on absolute values of the performance indicators. The number of replacement misses for the two programs settles at approximately 3.25 and 0.50 per key respectively. The number of instructions per key is a little larger for `contender` for large n. The overall conclusion of [LL99] is, that the number of cache misses is more important than a small difference in instruction count. Thus, I think that my choice to use MSD- and not LSD-oriented Radixsort to sort short keys in KR Sort is supported empirically, and that Rahman and Raman's [RR98] expectation, cf., Section 5.1, for a hybrid KR Sort to beat LSD Radixsort for $w = 256$ is uninteresting. Exponential range redex does not, like for LSD Radixsort, cf., Section 5.2.1, hold an obvious advantage over MSD Radixsort. In hindsight, this was already evident in the fact, that 1 was the best choice for `RR_steps`, cf., Section 6.2.

LaMarca and Ladner also experiment with "multi-quicksort", which uses one initial multi-way partitioning to divide the input into several groups, each of which fits into the cache with high probability. The number of replacement misses naturally settles for the "multi-quicksort" implementation. This is not the case for neither the "memory-tuned" Quicksort program, nor, according to my findings, for `sort`. It would be interesting to investigate the performance of `contender` if preceded by such a multi-way partitioning step.

Rahman and Raman In [RR99] Rahman and Raman test programs on "n uniformly distributed floating-point numbers, for $n = 2^i$, $i = 10, \ldots, 26$." I compare their results with the ones I have obtained on ISAAC input, even though this is not perfectly reasonable, cf., Section 3.1.

Rahman and Raman experiment with a MSD-oriented Radixsort (RR Radixsort) analogous to Contender. They choose the radix as $r = \min(\lceil \log n \rceil - 2, 16)$, i.e., their `max_radix` is 16. This supports, that the first levels of the cache hierarchy determine `max_radix`, cf., Section 6.1.

In the context of this thesis it is interesting, that Rahman and Raman compare their implementation of RR Radixsort with "a highly tuned implementation of recursive Quicksort from [LL99]." The gap in terms of running time between the two implementations is approximately 30%, cf., [RR99, Figure 2]. Compare this with the 25% gap between `contender` and STL `sort`. Interestingly, the number of replacement misses does not, like for `contender`, settle for Rahman and Raman's RR Radixsort program. Otherwise, their findings on the development on the number of cache misses match mine. Instruction counts are not presented in [RR99].

8 Conclusion

I set out to assess the practical value of some recent, "modern" algorithms for internal integer sorting on the computers currently mass produced, i.e., $w = 64$. These algorithms must all necessarily be analysed within the framework of the word RAM (WRAM) model of computation, which was presented along with a rationale for it in Chapter 1. In Chapter 2 I discussed some preliminary matters, the most important one of which is the notion of "packing" several short keys in one machine word.

These introductory chapters lead to a survey in Chapter 3 of internal integer sorting algorithms for the WRAM, that are not comparison based. That includes classical radix-sorting algorithms, and a recent variation on those; all these radix-oriented algorithms are called "first generation". The first-generation algorithms do not theoretically beat the $\Theta(n \log n)$ worst-case complexity of classical comparison-based sorting algorithms, as the running time of a first-generation algorithm depends not only on n, but also on either the length in bits of the integer keys, or the number of distinguishing (virtual) characters in them.

"Second-generation", or "modern", sorting algorithms do theoretically achieve $o(n \log n)$ running times by means of "packed" sorting of short keys. The short keys are the result of sufficiently reducing the range of the original b-bit input keys, where b is arbitrary, but at most w. The result of the packed sorting is used in a subsequent range expansion step to generate the sorted sequence of the original keys.

The packed sorting employs word-level parallelism (WLP). In Chapter 4 a critique of the research surveyed made me judge WLP, and hence all the "modern" algorithms, practically insignificant on a purely non-empirical basis. I thus chose to experimentally evaluate KR Sort, a first-generation algorithm. KR Sort uses exponential range redex and sorts short keys with Bucketsort. For greater flexibility I chose to instead use Adaptive Radixsort recursively in KR Sort.

In Chapter 5 I summarised existing experimental work with the modern algorithms. I decided that comparing KR Sort (`KR_sort`) with Adaptive Radixsort (`contender`) would be most appropriate in order to make a contribution to the pool of existing research. As a benchmark I decided to include Introsort (STL `sort`) in the comparison. As performance indicators I chose real running time, number of (simulated) cache misses, and number of (counted) instructions, as the literature demonstrates, that using a subset of these is not sufficient.

My implementations, written in `typewriter face` above, of the algorithms was described in Chapter 6. I used a performance-engineering and cache-conscious approach.

Chapter 7 dealt with the execution of the test runs. I presented a customisable cache simulator, NEXT-GEN, which I was forced to synthesise from existing simulators, because I was precluded from using powerful existing memory usage profilers. NEXT-GEN was constructed with the ATOM instrumentation toolkit, which I highly recommend. I included both worst-case and best-case data in the suite of test input classes. The experiments showed, that `KR_sort` is not competitive. Thus, for 64-bit computers range redex is not efficient in the context of internal integer sorting. Radix sorting remains the method of choice.

The main reason for the discrepancy between theory and practice is, that the WRAM model of computation does not reflect the memory hierarchy of real computers. `contender` and `sort` perform well, because they incur few cache misses. The opposite is true for `KR_sort`, even though the sparse tables, which range redex requires, have been implemented using hashing. In terms of number of instructions executed, `KR_sort` is the loser as well.

There is a need for a standard benchmark for internal (integer) sorting, both in terms of instance types and index algorithms, i.e., library routines like STL `sort`. Challenger programs should be implemented in the same spirit as the library routines. The reason why an instance benchmark does not exists is perhaps, that the majority of researchers agree, that internal sorting is a problem already solved to satisfaction, and accordingly concentrate on the MinuteSort benchmark; external, disk-to-disk sorting of as many records as possible within on minute.

Internal sorting remains an interesting case study of synthesis and analysis of algorithms, however.

A Plots

On the following pages the results are plotted. Along the first axis the base two logarithm p of the problem size n is shown. The dependent variable, be it sorting time, number of cache misses per key, or number of instructions per key, is plotted along the second axis. Note that for graphs that illustrate the same type of measurement, the range of the dependent variable is the same. This eases comparison. Furthermore, be aware that plots of sorting time use a base 10 logarithmic scale for the second axis. The plots are organised in three groups, one for each of the three performance indicators; sorting time, cache misses (per key), and instructions (per key) (in that order). Each group contains six plots, one for each of `contender`, STL `sort`, `KR_sort`, ISAAC input, median-of-three-killer input, and skewed input (in that order).

Note that `KR_sort` ran out of memory for $p \in \{20, \ldots, 23\}$ on ISAAC input, and for $p \in \{22, 23\}$ on skewed and Mo3K data.

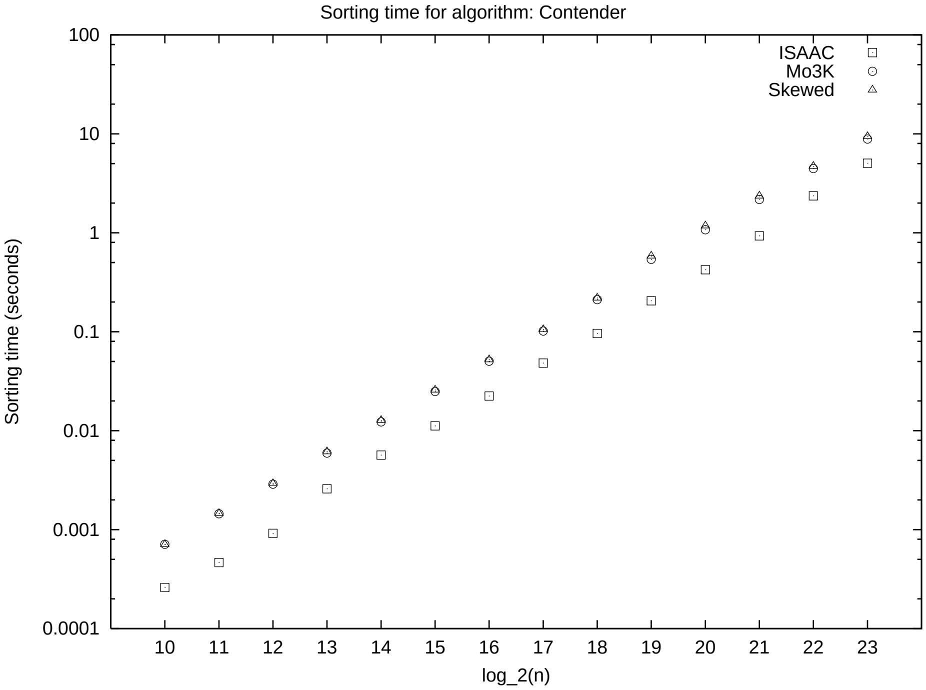

Figure 7: Sorting time for program: **contender**

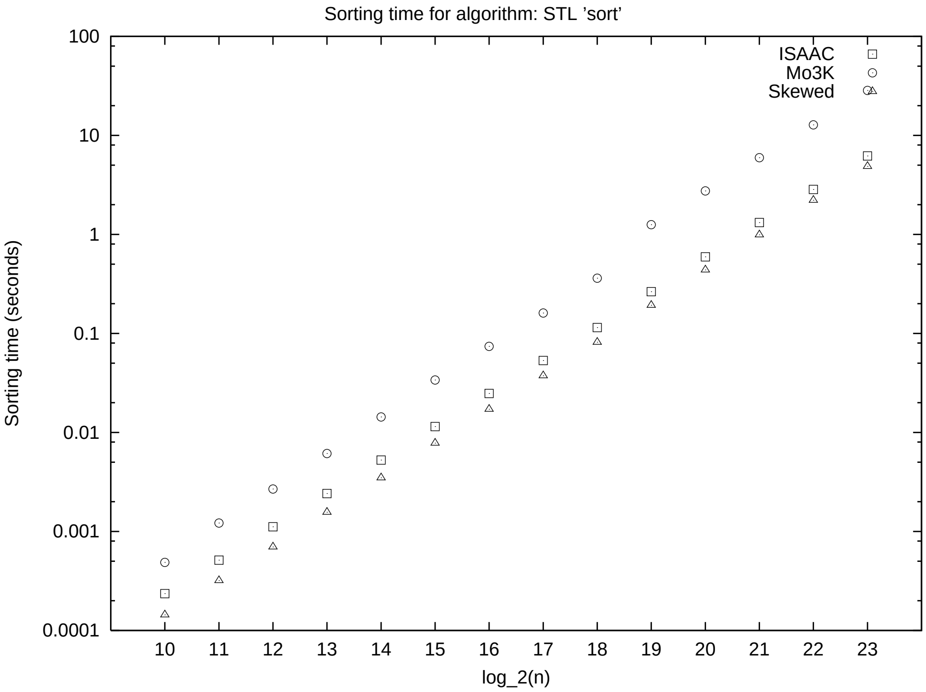

Figure 8: Sorting time for program: STL **sort**

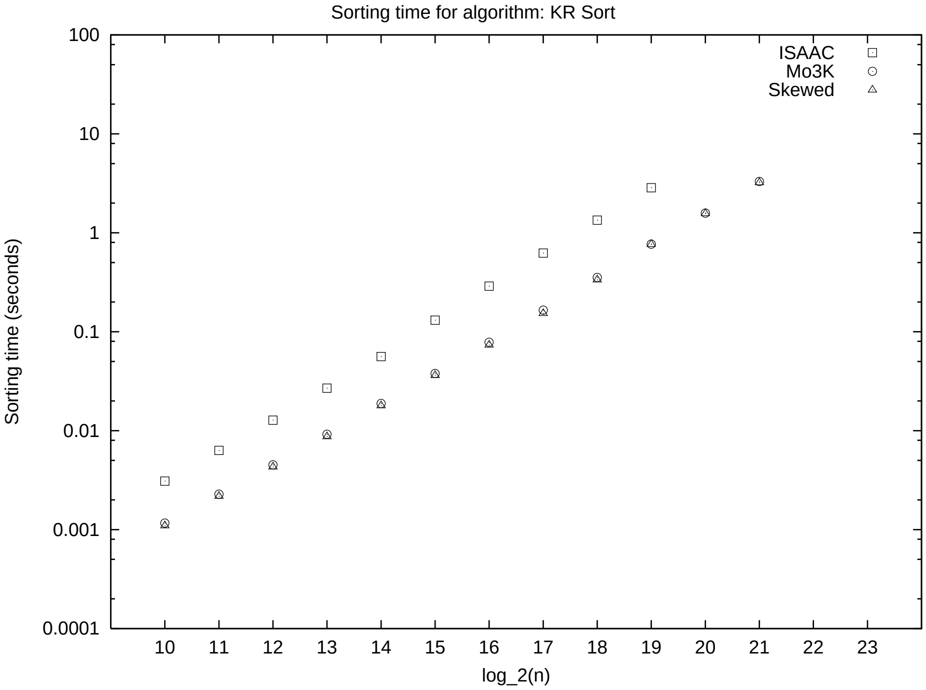

Figure 9: Sorting time for program: `KR_sort`

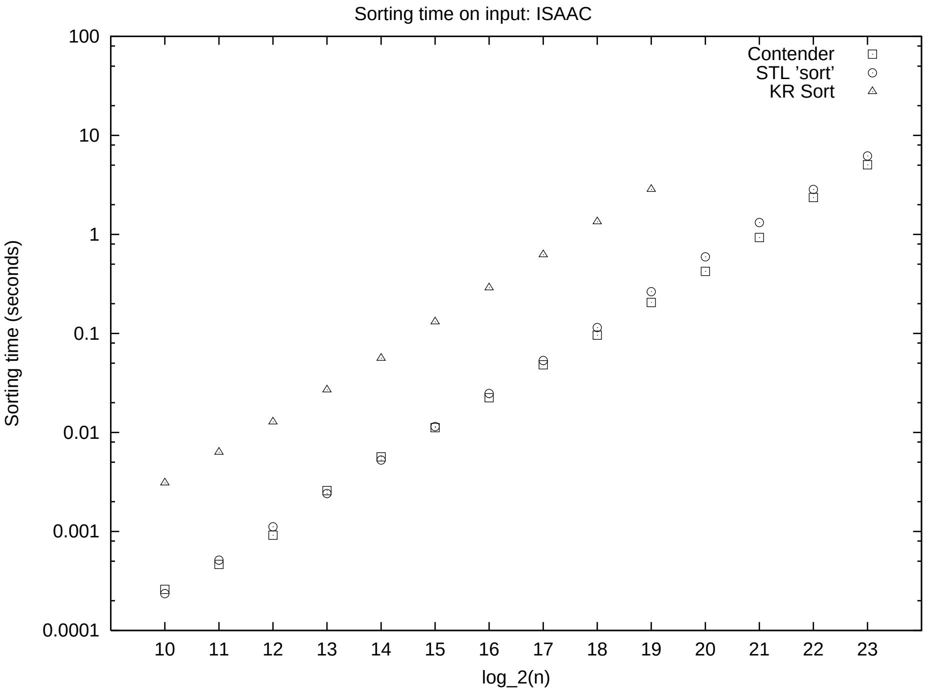

Figure 10: Sorting time on input: ISAAC

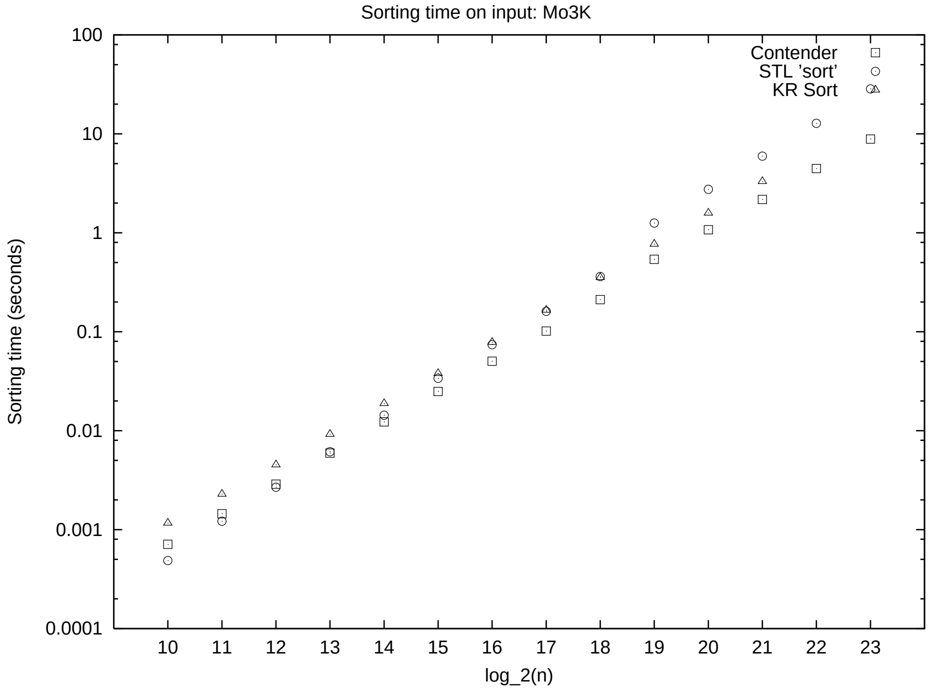

Figure 11: Sorting time on input: Median-of-three-killer

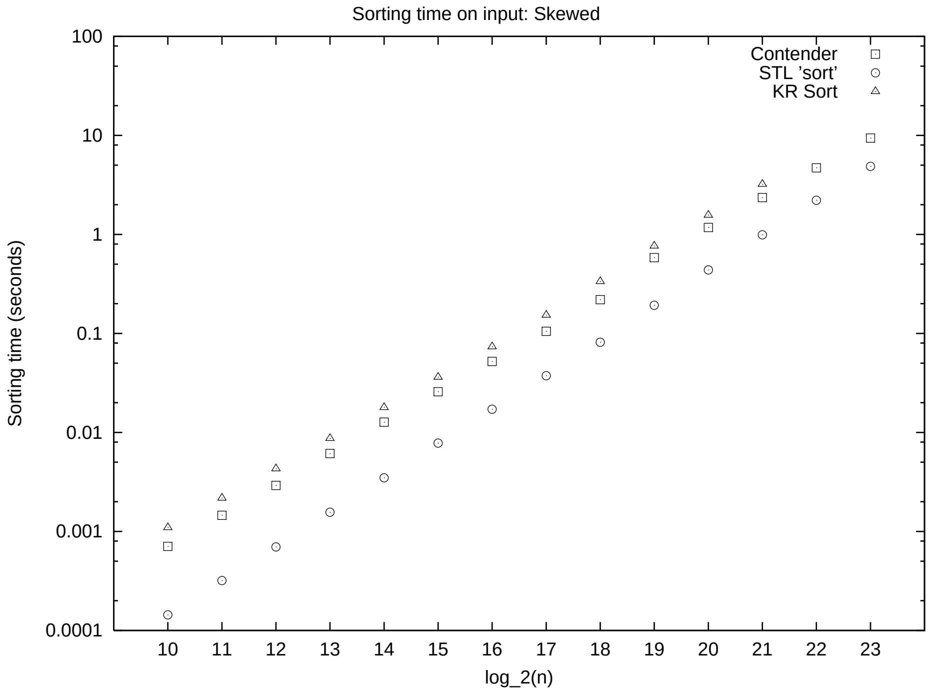

Figure 12: Sorting time on input: Skewed

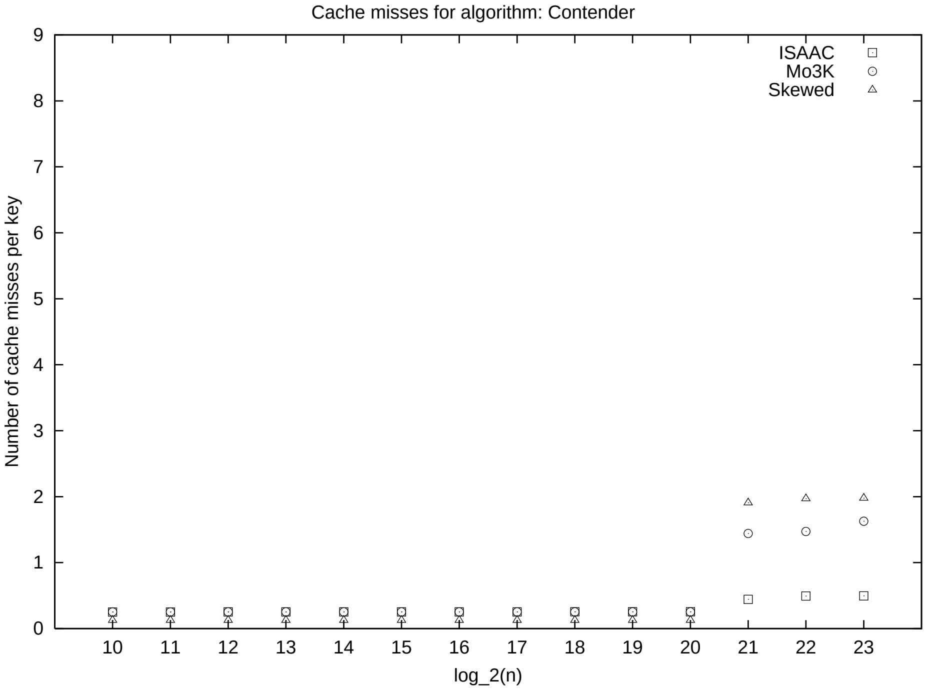

Figure 13: Cache misses for program: `contender`

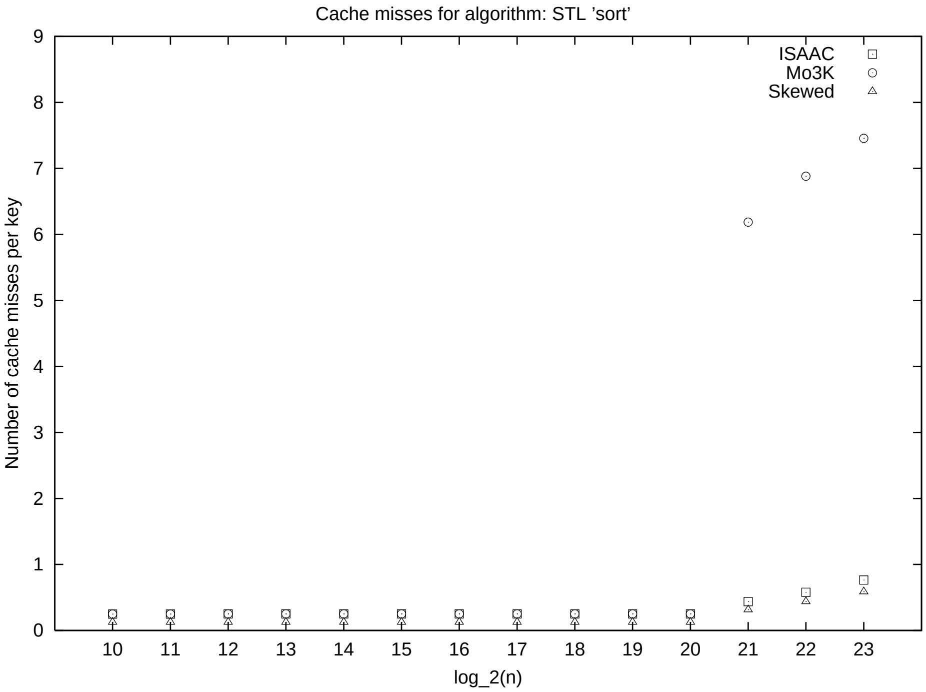

Figure 14: Cahce misses for program: STL `sort`

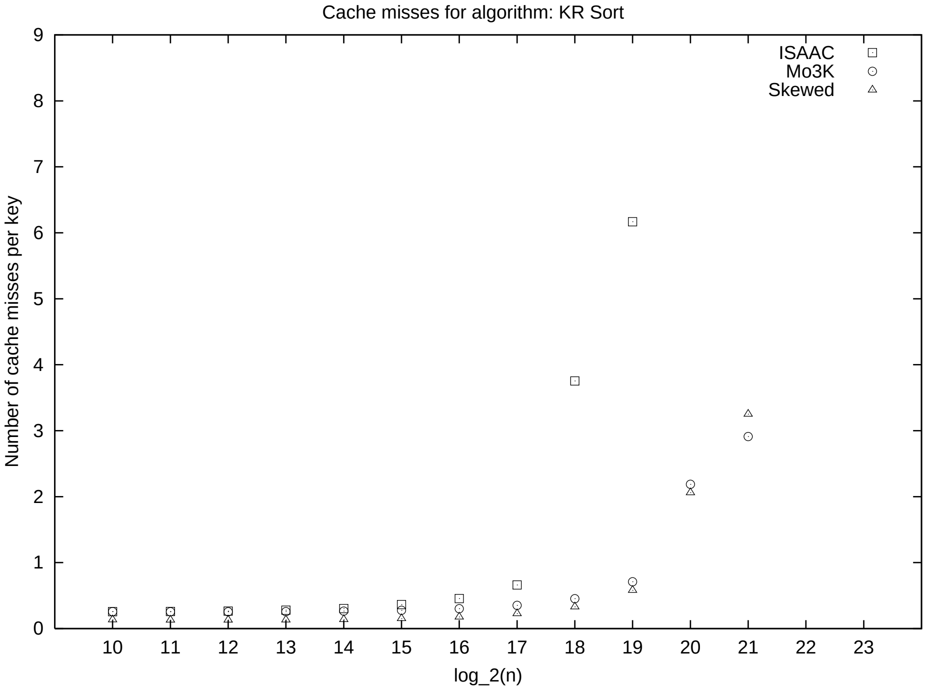

Figure 15: Cahce misses for program: `KR_sort`

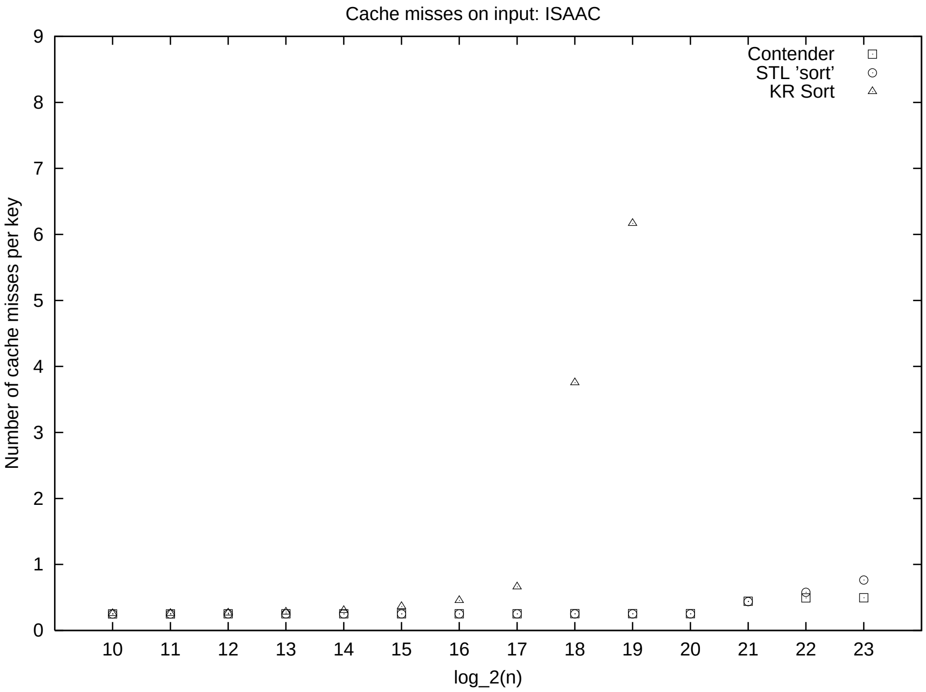

Figure 16: Cache misses on input: ISAAC

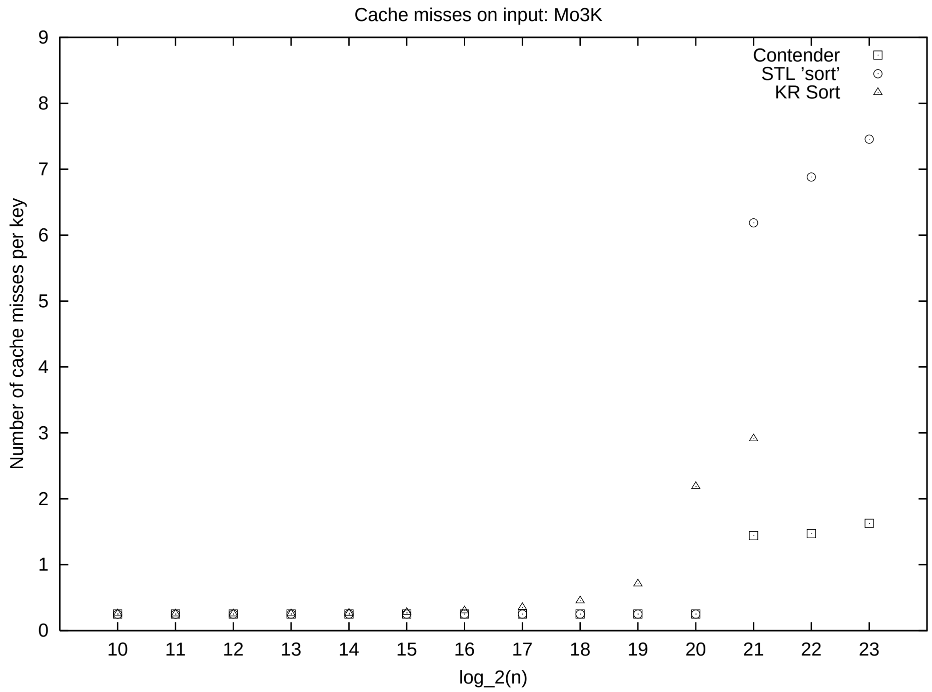

Figure 17: Cache misses on input: Median-of-three-killer

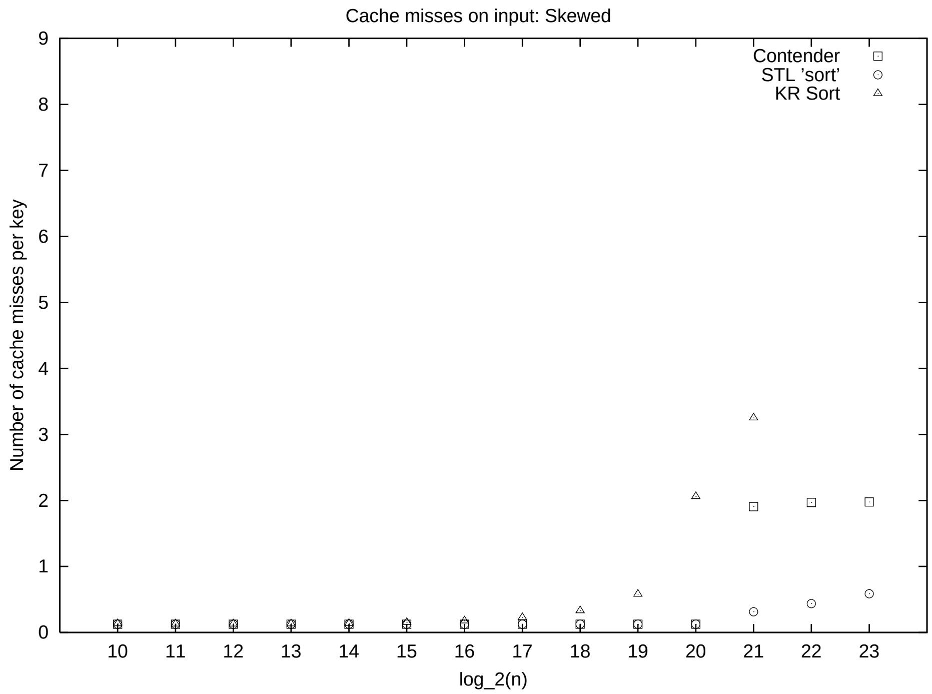

Figure 18: Cache misses on input: Skewed

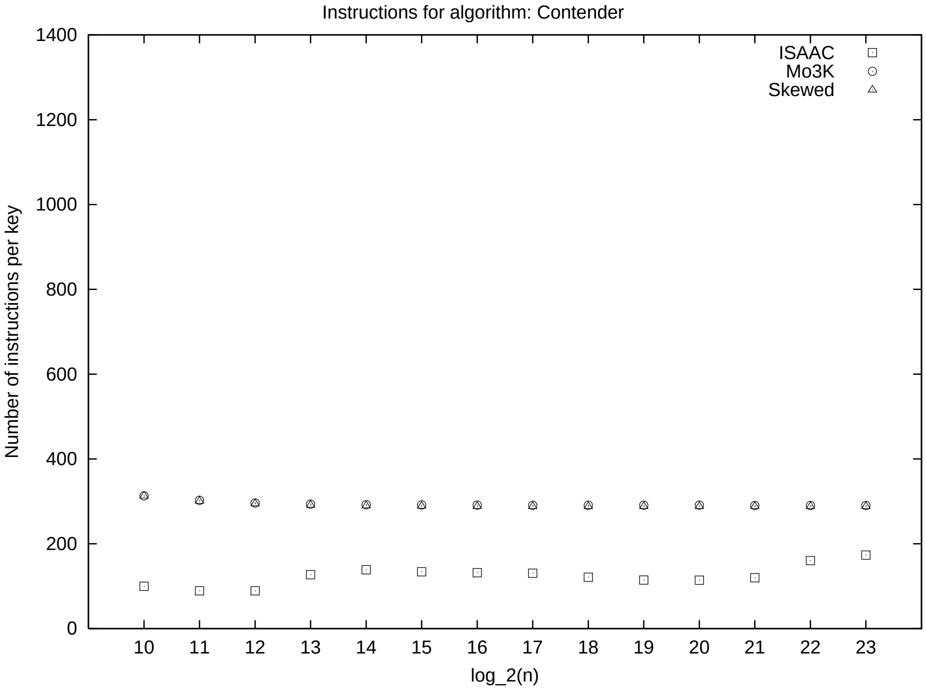

Figure 19: Instructions for program: `contender`

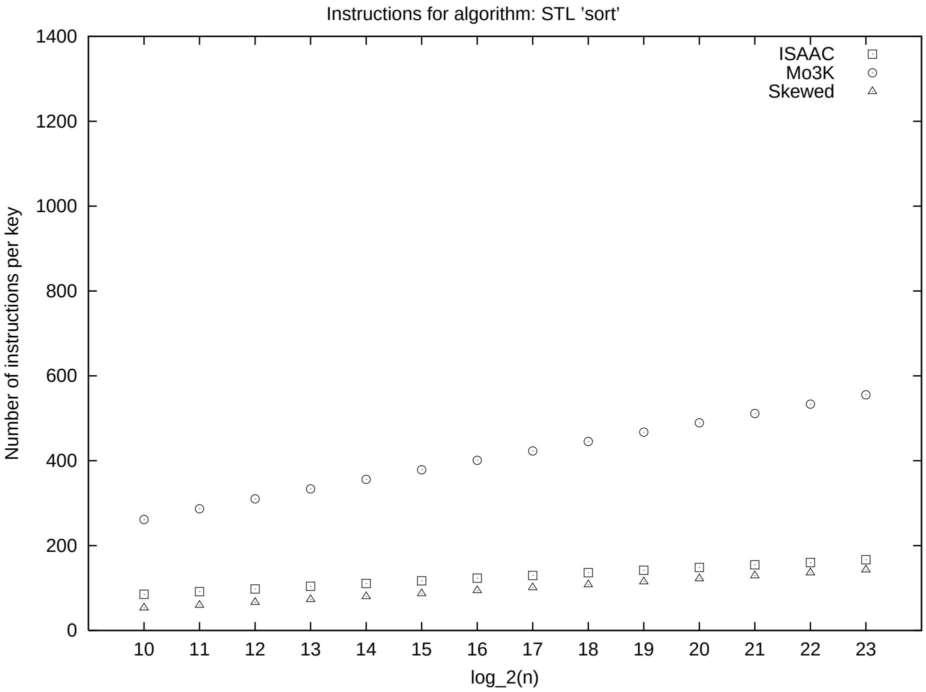

Figure 20: Instructions for program: STL `sort`

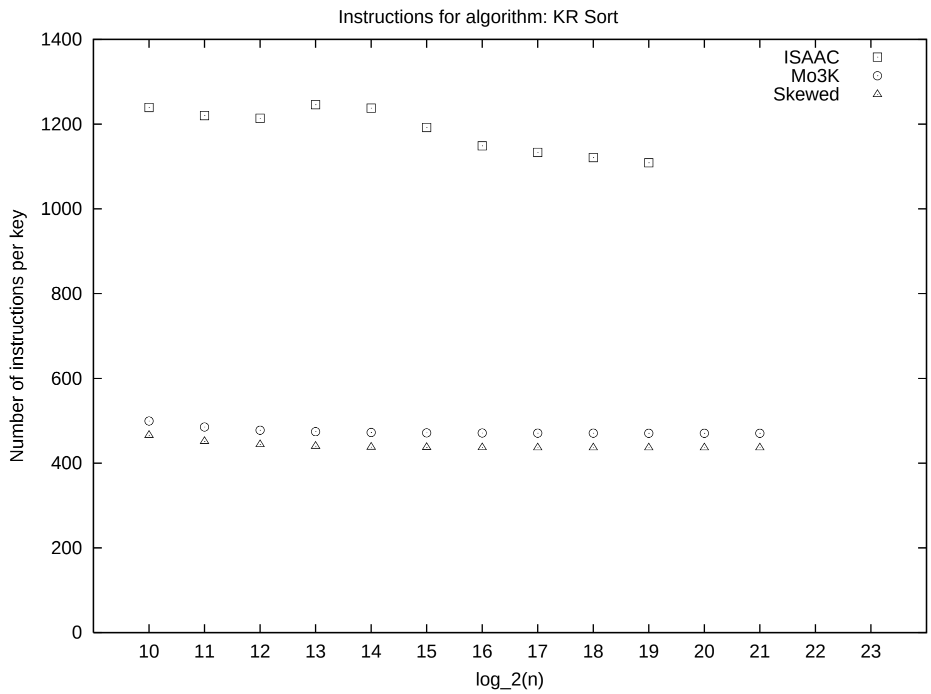

Figure 21: Instructions for program: `KR_sort`

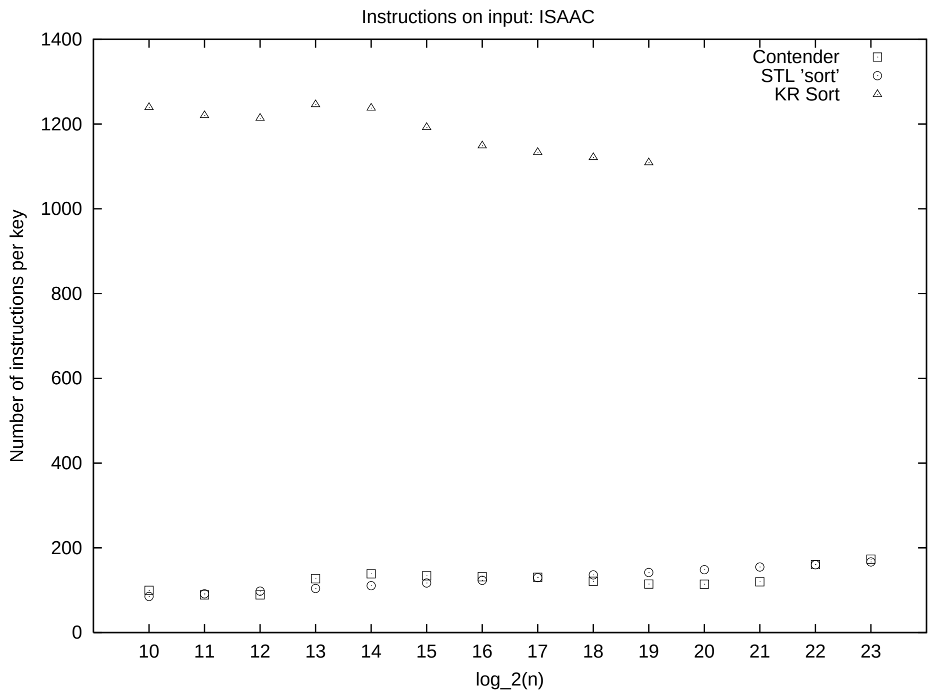

Figure 22: Instructions on input: ISAAC

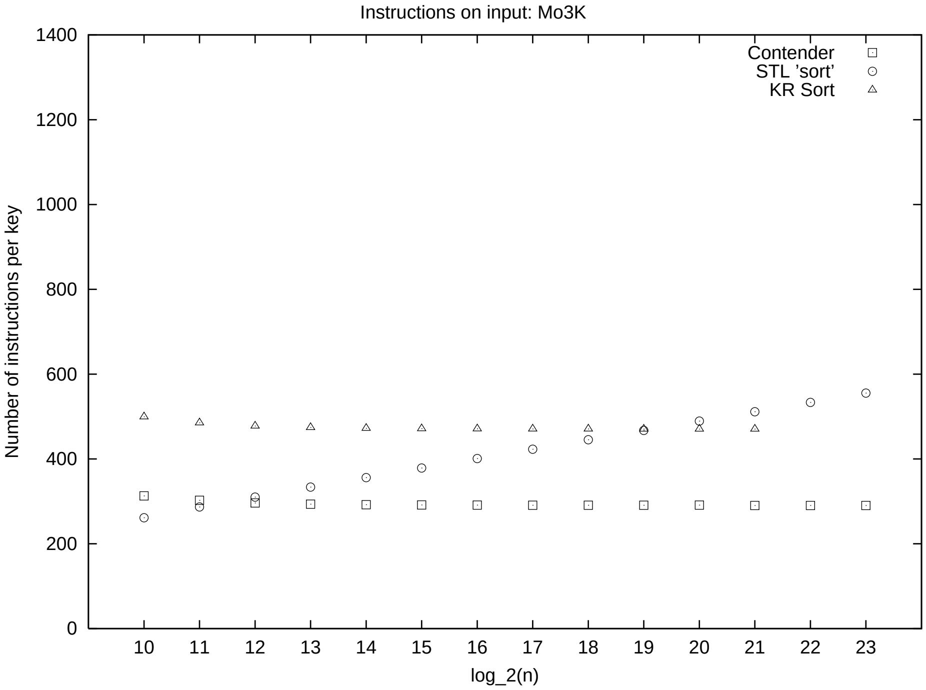

Figure 23: Instructions on input: Median-of-three-killer

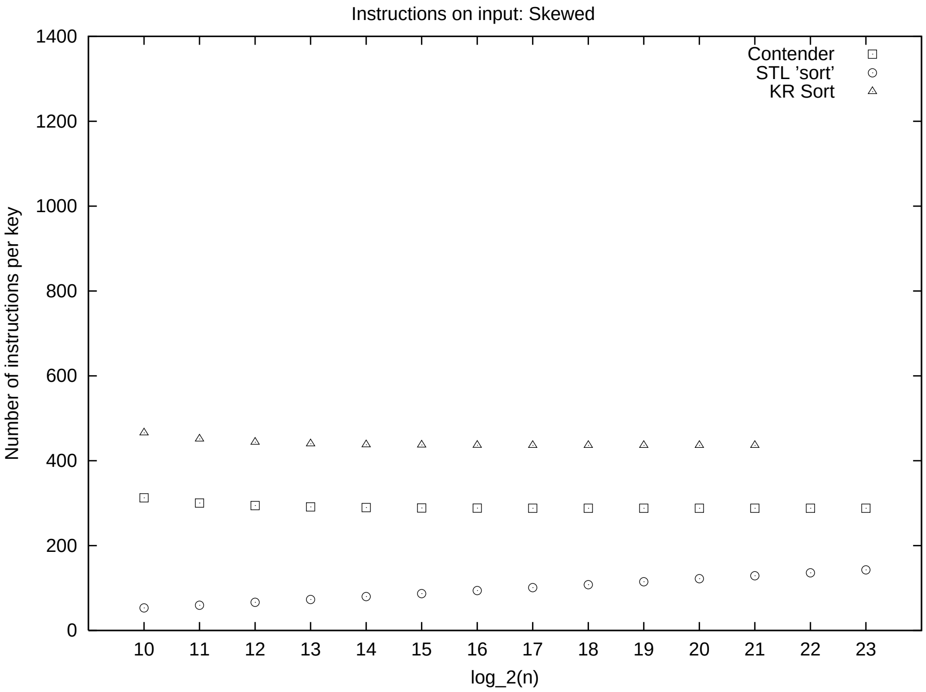

Figure 24: Instructions on input: Skewed

B Source code

This appendix contains the source code I have developed. It contains comments and references to further reading, which are supposed to make it comprehensible in isolation from further explanations of its design. The source code is available for download from the World Wide Web[25] under the GNU Public Licence.

B.1 Internal integer sorting

This section lists the source code I have written for solving the conservative value integer-sorting problem. See Chapter 6 for an overview. The files are listed in a bottom-up fashion; if a file `a` is `#include`d by another file `b`, then `a` is listed before `b`. For the sake of completeness, the file `isaac.hpp`, developed by Jackson, cf., Section 7.2, is listed as well.

B.1.1 NMSPC.h

```
// Filename: NMSPC.h
// Author: Morten Nicolaj Pedersen

#ifndef __NMSPC_H
#define __NMSPC_H

// #define __NAMESPACES
// #undef __NAMESPACES
// Do it with the '-D' and '-U' compiler switches.

#ifdef __NAMESPACES
#define _NMSPC_BEGIN(id) namespace id {
#define _NMSPC_END(id) }
#define _USING_NMSPC(id) using namespace id
#define _QUAL(id) id::
#else
#define _NMSPC_BEGIN(id)
#define _NMSPC_END(id)
#define _USING_NMSPC(id)
#define _QUAL(id)
#endif
// The 'id' is unnecessary in '_NMSPC_END', as it is ignored. Syntax
// highlighting works better, if the macro has an argument
// however. Also, a comment giving the 'id' would be needed anyway,
// each time '_NMSPC_END' were to be used, if 'id' was not included in
// its definition.

#endif // __NMSPC_H
```

B.1.2 Debug.h

```
// Filename: Debug.h
// Author: Morten Nicolaj Pedersen

#ifndef __DEBUG_H
#define __DEBUG_H
```

[25] `http://www.diku.dk/students/forge/`

```
// #pragma interface

#include "NMSPC.h"
#include <string>
#include <iostream>

#define VERIFY(cond,s) _QUAL(Debug) _Assert((cond),s,__LINE__,__FILE__)

_NMSPC_BEGIN(Debug)

/******************************************************************\
  Flags that (de)activate different types of assertions.
\******************************************************************/

/* A compilation unit (*.cpp file) should contain an include guard
   (optional), '#define' or '#undef' of 'NDEBUG', '#include' of
   'Debug.h', and '#include' of other headers in that order. This
   ensures that debugging is turned on/off on compilation unit
   basis. */

#ifdef NDEBUG
    const bool arg_check = false;
    const bool size_check = false;
    const bool table_check = false;
    const bool redex_check = false;
    const bool cvisp_check = false;
#else
    const bool arg_check = true;
    const bool size_check = true;
    const bool table_check = true;
    const bool redex_check = true;
    const bool cvisp_check = true;
#endif

/******************************************************************\
  Failure (class)
\******************************************************************/

/* Note that in an application, where it is not acceptable to simply
   terminate upon encountering a bad assertion, it would be useful, if
   the type of the exception to throw did not need to be hardwired
   into '_Assert'. Then information about the error/unmaintained
   invariant could conveniently be communicated through the !type! of
   the exception thrown.

   One would then use a hierachy of user-defined exceptions with
   'Failure' as parent. The virtual destructor ensures proper cleanup
   of data members, which are added in connection with derivation,
   cf., Stroustrup p. 318.

   This is not done in this context of word-level-parallelism sorting,
   since i) termination upon failure is acceptable, and ii) hardwiring
   is mandatory with EGCS. */

class Failure {

public:
```

```
    Failure(const char* s) : error_msg(s) {}
    virtual ~Failure() {}

    virtual string cause() { return "Failure: " + error_msg + '\n'; }

private:

    string error_msg;

};

/******************************************************************\
  Functions
\******************************************************************/

/*----------------------------------------------------------------*\
  _Assert (see also 'VERIFY' above)
\*----------------------------------------------------------------*/

/* Usage: '_Assert( <bool_const> || condition, <string literal>);' or
   better: 'VERIFY(<bool_const> || condition, <string literal>);'. The
   advantage of using a constant enabling/disabling expression like
   <bool_const> is that the whole assertion can be compiled away* when
   not enabled. A non-constant expression, for instance a variable,
   can also be used to turn assertion on/off at run-time, but then
   code will invariably be generated.

   * To ensure this, the args of the constructor for the exception to
     throw (Failure) must be hardwired. See Stroustrup p. 753.

   Capitalized to distinguish it from 'assert' from '<cassert>'. */

inline void _Assert(bool condition, const char* s,
                    int lno = 0, char* fname="noname")
{
    if (!condition) {
        cerr << lno << '@' << fname << ": ";
        throw Failure(s);
    }
}

_NMSPC_END(Debug)

#endif // __DEBUG_H
```

B.1.3 Global.h

```
// Filename: Global.h
// Author: Morten Nicolaj Pedersen
/* Remarks: 'main' or some other entity should check that 'idx_t' can
   hold the number of elements to be sorted. */

#ifndef __GLOBAL_H
#define __GLOBAL_H

#pragma interface

#include "NMSPC.h"
```

```
#include <limits.h>    // UINT_MAX
#include <vector>
#include <functional>  // binary_function

////////////////////////////////////////////////////////////////////
// DECLARATIONS
////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(Global)

/******************************************************************\
  typedefs and constants
\******************************************************************/

typedef unsigned long int word;
/* 64 bit on egcs-2.91.66 19990314 (egcs-1.1.2 release) installed on
   OSF1 regnar.diku.dk V4.0 878 alpha. */
const word word_min = 0U;
// const word word_max = ULONG_LONG_MAX;
const word word_max = ~word_min;
/* 'word' is the type of keys. To make an integer constant of type
   `unsigned long int', add the suffix `UL' to the integer. Use
   function prototypes when using function arguments of the type
   'word'. */

// typedef unsigned char uchar;

typedef unsigned short width_t;
const width_t w = 8*sizeof(word);
/* The machine 'word' length 'w' in bits. */

typedef unsigned int idx_t;
/* The type of indices into tables ('vector's and arrays) of keys. It
   follows that 'idx_t' can hold the number of keys (minus
   one). Mainly introduced because 'sizeof(size_t) == 8' on the Alpha,
   so the gain of associating an 'unsigned int' with each key instead
   of a 'size_t' is considerable, when there are millions of keys. */
const idx_t idx_t_max = UINT_MAX;

typedef vector<word> word_container;
typedef word_container::iterator word_iterator;

/******************************************************************\
  Cmp_bit_pattern (class)
\******************************************************************/

// See also 'make_mask' below.

class Cmp_bit_pattern : public binary_function<word,word,bool> {

public:

    Cmp_bit_pattern(word m) : mask(m) {}

    bool operator()(word x, word y) const { return (x & mask) < (y & mask); }

private:

    const word mask;
```

```
};

/**********************************************************************\
  Functions
\**********************************************************************/

word make_mask(width_t width, width_t base_shift);
/* Returns the word which has zeroes at all bit positions except those
   in '{base_shift,...,base_shift+width-1}' (LSB is no. 0). */

int log2_floor(unsigned int n);
/* Returns the floor of the base two logarithm of 'n'. */

void read_words(word_container& dest, const char* filename);
void write_words(word_container& src, const char* filename);
/* 'read_words' fills the destination 'vector' (verified to be empty)
   with the contents of the named input file. 'write_words' flushes
   the source 'vector' to the named output file, which is overwritten,
   and clears the source 'vector'.

   The file format used is:

       n
       values

   where 'n' is the number of words in the file, and 'values' the
   words themselves in hex notation. Note: only two end-of-lines.

   NOTE: none of these functions are used anywhere. They are an
   artifact of early code development.  */

// These functions exist in STL <algorithm> header:

// template<class Forward>
// bool is_sorted(Forward first, Forward last);
// /* 'Forward' is a forward iterator. Verifies "non-descending" sorting. */

// template<class Forward, class Compare>
// bool is_sorted(Forward first, Forward last, Compare comp);
// /* 'Forward' is a forward iterator. Checks sorting by 'comp'. */

_NMSPC_END(Global)

#endif // __GLOBAL_H

////////////////////////////////////////////////////////////////////////
//
////////////////////////////////////////////////////////////////////////

/**********************************************************************\

\**********************************************************************/

/*------------------------------------------------------------------*\

\*------------------------------------------------------------------*/

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Global.o"
```

```
//  compilation-scroll-output: t
//  End:
```

B.1.4 Global.cpp

```
// Filename: Global.cpp
// Author: Morten Nicolaj Pedersen

#pragma implementation

// #undef NDEBUG
#define NDEBUG

#include "Debug.h"
#include "Global.h"
#include <vector>
#include <fstream>   // ifstream, ofstream
#include <iterator>  // iterator_traits

/////////////////////////////////////////////////////////////////////
// IMPLEMENTATION
/////////////////////////////////////////////////////////////////////

_USING_NMSPC(Global);

/******************************************************************\
  Functions
\******************************************************************/

word _QUAL(Global) make_mask(width_t width, width_t base_shift)
{
    VERIFY(!_QUAL(Debug) arg_check || !(width & 1U),
           "make_mask: 'width' not even");
    width_t half_width = width >> 1;
    word mask = (1U << half_width) - 1U;
    mask |= mask << half_width;
    mask <<= base_shift;
    return mask;
    // Relies on 'width' being even.
} /* make_mask */

/*-----------------------------------------------------------------*\
  log2_floor
\*-----------------------------------------------------------------*/

int _QUAL(Global) log2_floor(unsigned int n)
{
    VERIFY(!_QUAL(Debug) arg_check || n,"log2_floor: applied to zero");
    int i;
    for ( i = 0; n; n >>= 1, i++)
        ;
    return i - 1;
} /* log2_floor */

/*-----------------------------------------------------------------*\
  read_words + write_words
\*-----------------------------------------------------------------*/
```

```
void _QUAL(Global) read_words(word_container& dest, const char* filename)
{
    VERIFY(!dest.size(),"read_words: destination not empty");
    ifstream file(filename);
    word_container::size_type n;
    file >> dec >> n;
    VERIFY(file.good(),"read_words: input of number of words failed");
    dest.reserve(n);
    word val;
    int i ; // Declared here because it must survive loop.
    for ( i = 0, file >> hex; file >> val; i++)
        dest.push_back(val);
    VERIFY(i == n,"read_words: input of word values failed");
} /* read_words */

void _QUAL(Global) write_words(word_container& src, const char* filename)
{
    ofstream file(filename);
    word_container::size_type n = src.size();
    file << dec << n << '\n';
    VERIFY(file.good(),"write_words: output of number of words failed");
    int i; // Declared here because it must survive loop.
    for (i = 0, file << hex; i < n && (file << src[i] << ' '); i++)
        ;
    VERIFY(i == n,"write_words: output of values failed");
    src.clear();
} /* write_words */

/*-------------------------------------------------------------------*\
  is_sorted
\*-------------------------------------------------------------------*/

// These functions exist in STL:

// /* 'Forward' is a forward iterator. Checks "non-descending": */
// template<class Forward>
// bool _QUAL(Global) is_sorted(Forward first, Forward last)
// {
//     if ( first == last ) return true;
//     iterator_traits<Forward>::value_type prev = *first;
//     for ( Forward keys_it = first + 1; keys_it != keys.end(); ++keys_it ) {
//         if (*keys_it < prev)
//             return false;
//      prev = *keys_it;
//     }

//     return true;
// } /* is_sorted */

// /* 'Forward' is a forward iterator. Verifies sorting according to 'comp': */
// template<class Forward, class Compare>
// bool _QUAL(Global) is_sorted(Forward first, Forward last, Compare comp)
// {
//     if ( first == last ) return true;
//     iterator_traits<Forward>::value_type prev = *first;
//     for ( Forward keys_it = first + 1; keys_it != keys.end(); ++keys_it ) {
//         if (comp(*keys_it,prev))
//             return false;
//      prev = *keys_it;
```

```
//     }

//     return true;
// } /* is_sorted */

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Global.o"
//  compilation-scroll-output: t
//  End:
```

B.1.5 Sparse_table.h

```
// Filename: Sparse_table.h
// Author: Morten Nicolaj Pedersen

#ifndef __SPARSE_TABLE_H
#define __SPARSE_TABLE_H

#pragma interface

#include "Debug.h"
#include "Global.h"
#include <vector>
#include <hash_map>    // STL_table
#include <functional>  // unary_function, equal_to
#include <utility>     // pair
#include <cstddef>     // size_t

///////////////////////////////////////////////////////////////////////
// DECLARATIONS
///////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(Sparse_table)

/*******************************************************************\
  Table_interface (abstract class)
\*******************************************************************/

/* Unique entries. Abstract class. I will not bother with introducing
   member iterator types and more with the sole purpose of imitating
   STL containers. Since !copies! of data values of the type 'T' are
   inserted, 'T' must support the essential operators: copy
   constructor, copy assignment and destructor. Cf. Stroustrup section
   11.7. */

template<class Key, class T>
class Table_interface {

public:

    typedef _QUAL(Global) idx_t size_type;
    typedef Key key_type;
    typedef T data_type;

    virtual ~Table_interface() {}

    virtual T* find(const Key& key) = 0;
    /* Returns a pointer to the data part of the unique entry with the
```

```
      given key value. NULL (0) if not found. */

   virtual T* insert(const Key& key, const T& data) = 0;
   /* Returns a pointer to the data part of the new entry. NULL (0)
      if an entry with the given key value already exists. */

   virtual bool erase(const Key& key) = 0;
   /* Erases the entry with the given key value. Returns 'true' if it
      pre-existed, 'false' otherwise. Perhaps nicer to return the
      data associated with the entry, which is deleted. Erase is not
      used in namespace 'Range_redex', so this effort I do not
      make. */

   virtual inline size_type size() const = 0;
   virtual inline size_type max_size() const = 0;
   virtual inline bool empty() const = 0;

}; /* Table_interface */

/******************************************************************\
  Hash_func (concrete type)
\******************************************************************/

/* Policy. 'mapped_w' is the allowed maximum width in bits of the
   values that variables of type 'Key' will be mapped to by
   'operator()'. See also the 'mapped_sz' constant.

   Language rules dictate that only one specialization per type is
   allowed. A third template argument 'string method' could remedy
   this, i.e. 'Hash_func<unsigned int,unsigned
   short,"Dietzfelbinger">'. Or 'operator()' could be made virtual so
   alternatives could be provided by derivation for dynamic
   polymorphism. */

template<class Key, typename m_type>
class Hash_func : unary_function<Key,m_type> {

public:
   typedef m_type mapped_type;

   Hash_func(_QUAL(Global) width_t K_w = 8*sizeof(Key),
             _QUAL(Global) width_t m_w = 16)
      /* 'K_w' can be used to pass additional knowledge about the
         domain to 'Hash_func'. I.e. 'Key == unsigned int', but 'key
         <= UINT_MAX / 2' as well. */
      : Key_w(K_w), mapped_w(m_w), mapped_sz(1U << m_w)
      /* Ensured that 'mapped_sz' does not overflow, if the
         assertion in the body succeeds. */
   {
      VERIFY(Key_w > 0 && mapped_w > 0,
             "Hash_func: width must be positive");
      VERIFY(Key_w > mapped_w, "Hash func: misuse");
      VERIFY(8*sizeof(mapped_type) >= mapped_w,
             "Hash_func: 'mapped_type' too narrow");
      VERIFY(8*sizeof(Key) >= Key_w,
             "Hash_func: value 'Key_w' too big");
      VERIFY(8*sizeof(unsigned int) > mapped_w,
             "Hash_func: type of 'mapped_sz' too narrow");
   }
```

```
    inline mapped_type operator() (const Key& key) const;
    /* Should this function be static? No! Then for instance 'a' of
       the specialization for 'word' would only be chosen once. Also,
       it needs to access the non-static '_w' data members.*/

    const unsigned int mapped_sz;
    /* operator() maps 'Key's to '{0,...,mapped_sz-1}'. Hence the size
       of its mapped range is 'mapped_sz'. */

private:
    _QUAL(Global) width_t mapped_w;
    _QUAL(Global) width_t Key_w;

}; /* Hash_func */

/*******************************************************************\
  Hash_table (concrete type)
\*******************************************************************/

/* Unique entries. Dynamic dictionary in the sense, that it supports
   'insert' and 'delete'. Its maximum size is static though. Written
   in the spirit of the STL containers, although complete conformance
   is not offered. 'T' is the type of information (data) associated
   with keys. See 'Table_interface' for comments on 'T'. 'HF' and 'EQ'
   are policies offering compile-time polymorphism.

   The size of a 'Hash_table' does not adapt dynamically. For speed it
   is therefore important that one chooses the 'Hash_func'/mapped
   range, so that the maximum number of keys that are mapped to the
   same value is small - even when the table is almost full. Do not
   confuse the cardinality of the mapped range with the number of keys
   in the hash table.

   The 'Hash_table' implementation of 'Table_interface' is to be used
   in (exponential) range reduction. Thus 'erase' is not necessary -
   range reduction can be implemented by means of 'insert' and 'find'
   only. Therefore a simple "next-alloc" scheme is adopted. 'entries'
   grows with insertions, and erasing introduces "holes". If necessary
   these holes are found by traversing 'entries' in a circular
   fashion.

   The duplicate checking of 'insert', among other things, is turned
   on and off with 'Debug::table_check'. */

template<class Key, class T, class HF=Hash_func<Key,unsigned int>,
    class EQ=equal_to<Key> >
class Hash_table : public Table_interface<Key,T> {

public:

    typedef HF hasher;
    typedef EQ key_equal;

    Hash_table(size_type n, const HF& hf=HF(), const EQ& eq=EQ())
        /* 'n' is considered to be a !tight! upper bound on the number
           of elements present in the hash table at any time. If this
           capacity restriction is not obeyed, 'insert' exits with an
           overflow error message. */
```

```
        : capacity(n), erased_count(0), size_count(0),
          hash_func(hf), equal_op(eq), cache(0)
    {
        /* Allocate and reserve. */
        buckets = vector<Entry*>(hf.mapped_sz); /* Inits. with 0. */
        entries.reserve(n);
        pos = entries.begin();
    }

    ~Hash_table()
    {
        buckets.clear();
        entries.clear();
        // Uses destructor of 'T'.
    }

    /* Next six functions inherited from 'Table_interface'. For
       efficiency reasons duplicate checking, and other belt and
       suspenders verifications, can be turned off with
       'Debug::table_check'. */
    T* find(const Key& key);
    T* insert(const Key& key, const T& data);
    bool erase(const Key& key);

    inline size_type size() const { return size_count; }
    inline size_type max_size() const { return capacity; }
    inline bool empty() const { return size_count == 0; }

private:

    HF hash_func;
    EQ equal_op;
    size_type erased_count, size_count;
    const size_type capacity;

    struct Entry {

        Entry(Key k, T d) : key(k), data(d) {}
        // Uses copy constructor of 'T'.

        Key key;
        T data;
        Entry* next;
        Entry** prev_ptr;
        /* Iff 'Entry' is well defined, 'prev_ptr !=
           0'. Because of 'prev_ptr' an 'Entry' can be linked
           out of its bucket upon erasure. The elements of
           'entries' that can be accessed through 'buckets'
           are therefore all well defined. */
    };

    vector<Entry*> buckets;
    vector<Entry> entries;
    vector<Entry>::iterator pos;
    /* A one-level hashing scheme. The hashed values are used as
       indices into 'buckets'. The actual 'T' data values are kept in
       'entries', and are organized in chains. */

    Entry* cache;
```

```
    Entry* _find(const Key& key);
    /* Helper function. Returns '0 == NULL' if not found. Not a
       'const' function since it updates 'cache'. */

}; /* Hash_table */

/*******************************************************************\
  One_one (concrete type)
\*******************************************************************/

/* Policy. A 1-1 function from values of type 'Key' to
   {0,...,'range_sz-1'}.

   See 'Hash_func' for a comment on specialization that applies here
   as well. Generally 'One_one' has a lot in commen with
   'Hash_func'. The 'Key_w' here corresponds to the 'mapped_w'
   there.

   Introduced for use as 'BIJ' in 'Simple_table' below. 'BIJ' might
   seem silly. It should be regarded as a policy for checking the
   validity of the table indexes, that are passed to 'Simple_table'
   primitives. 'One_one' implements a check for "out of range". */

template<class Key, typename r_type>
class One_one : unary_function<Key,r_type> {

public:
    typedef r_type range_type;

    One_one(_QUAL(Global) width_t K_w=8*sizeof(Key))
        /* See 'Hash_func' constructor for a comment on 'K_w'. Note in
           that connection: for 'One_one' the "mapped" width is the
           same as 'K_w'. */
        : range_sz(1U << K_w), Key_w(K_w)
        /* Ensured that 'range_sz' does not overflow by verifications
           in body.. */
    {
        VERIFY(Key_w > 0,"One_one: width must be positive");
        VERIFY(8*sizeof(range_type) >= Key_w,
               "One_one: 'range_type' too narrow");
        VERIFY(8*sizeof(Key) >= Key_w,
               "One_one: value 'Key_w' too big");
        VERIFY(8*sizeof(unsigned int) > Key_w,
               "One_one: type of 'range_sz' too narrow");
    }

    inline range_type operator() (const Key& key) const;

    const unsigned int range_sz;
    /* operator() maps 'Key's to '{0,...,range_sz-1}'. Hence the size
       of its range is 'range_sz'. */

private:
    _QUAL(Global) width_t Key_w;

}; /* One_one */

/*******************************************************************\
```

```
  Simple_table (concrete type)
\*****************************************************************/

/* BIJ is a policy for converting values of type 'Key' to table
   ('vector') indexes, and checking their validity. Perhaps 'unsigned
   short' in the template specification should simply be replaced by
   'Table_interface<Key,T>::size_type'.

   For better uniformity an 'EQ' template argument should be
   added. Compare with 'Hash_table'. */

template<class Key, class T, class BIJ=One_one<Key, unsigned short> >
class Simple_table : public Table_interface<Key,T> {

public:
    typedef BIJ bijector;

    Simple_table(const BIJ& b = BIJ())
        : entries(b.range_sz), bij(b), size_count(0) {}
    /* Initializes with default values, which are flagged as
       unpresent. */

    ~Simple_table() { entries.clear(); }
    // Uses destructor of 'T'.

    /* Next six functions inherited from 'Table_interface'. */
    T* find(const Key& key);
    T* insert(const Key& key, const T& data);
    bool erase(const Key& key);

    inline size_type size() const { return size_count; }
    inline size_type max_size() const { return bij.range_sz; }
    inline bool empty() const { return size_count == 0; }

private:

    struct Entry {
        bool present;
        T data;

        Entry(bool p = false, const T& d = T())
            : present(p), data(d) {}
        /* Uses the default value of 'T' if second argument is
           omitted. */
    };

    vector<Entry> entries;

    BIJ bij;
    size_type size_count;

}; /* Simple_table */

/*****************************************************************\
  STL_table (concrete type)
\*****************************************************************/

/* Reroutes to 'hash_map'. */
```

```
// For 'hash_map' with 'Hash_func' hash function:

// template<class Key, class T, class HF=Hash_func<Key,size_t>,
//     class EQ=equal_to<Key> >
// class STL_table : public Table_interface<Key,T>,
//                 private hash_map<Key,T,HF,EQ>
// {
// public:

//     typedef HF hasher;
//     typedef EQ key_equal;
//     typedef Table_interface<Key,T>::size_type size_type;

//     STL_table(size_type n, const HF& hf=HF(), const EQ& eq=EQ())
//      : hash_map<Key,T,HF,EQ>(hf.mapped_sz,hf,eq) {}
//     // Note that 'n' is not used. The STL Hashed Associative
//     // Containers does not provide member methods for specifying
//     // approximately how many elements will be inserted. Can possibly
//     // be done by not using the default 'Allocator'. Passing
//     // 'hf.mapped_sz' ensures at least that number of buckets in the
//     // hash table.
//     ~STL_table() {}

//     inline T* find(const Key& key) {
//      hash_map<Key,T,HF,EQ>::iterator hm_it =
//          hash_map<Key,T,HF,EQ>::find(key);
//      if ( hm_it == end() ) return static_cast<T*>(0); // NULL
//      else return static_cast<T*>(&(hm_it->second));
//     }

//     inline T* insert(const Key& key, const T& data) {
//      pair<hash_map<Key,T,HF,EQ>::iterator,bool> P =
//          hash_map<Key,T,HF,EQ>::insert(pair<Key,T>(key,data));
//      if ( P.second )
//          return static_cast<T*>(&((P.first)->second));
//      else
//          return static_cast<T*>(0); // NULL
//     }

//     inline bool erase(const Key& key) {
//      return 1 == hash_map<Key,T,HF,EQ>::erase(key);
//      // 'insert' ensures that 'hash_map<Key,T,HF,EQ>::erase(key)'
//      // is either 0 or 1.
//     }

//     inline size_type size() const { return hash_map<Key,T,HF,EQ>::size(); }
//     inline size_type max_size() const { return hash_map<Key,T,HF,EQ>::max_size(); }
//     inline bool empty() const { return hash_map<Key,T,HF,EQ>::empty(); }
//     // Should care be taken to ensure that 'size_type' does not
//     // overflow? Only a potential problem with 'max_size', since
//     // 'idx_t' can easily hold the number of element to be sorted, and
//     // thus the number of groups. Cf., 'Table_interface' and 'Global.h'.

// }; /* STL_table */

// For 'hash_map' with 'hash' hash function:

template<class Key, class T>
class STL_table : public Table_interface<Key,T>,
```

```
                  private hash_map<Key,T>
{
public:

    typedef hash_map<Key,T>::hasher hasher;
    typedef hash_map<Key,T>::key_equal key_equal;
    typedef Table_interface<Key,T>::size_type size_type;

    STL_table(size_type no_of_buckets) : hash_map<Key,T>(no_of_buckets) {}
    ~STL_table() {}

    inline T* find(const Key& key) {
        hash_map<Key,T>::iterator hm_it = hash_map<Key,T>::find(key);
        if ( hm_it == end() ) return static_cast<T*>(0); // NULL
        else return static_cast<T*>(&(hm_it->second));
    }

    inline T* insert(const Key& key, const T& data) {
        pair<hash_map<Key,T>::iterator,bool> P =
            hash_map<Key,T>::insert(pair<Key,T>(key,data));
        if ( P.second )
            return static_cast<T*>(&((P.first)->second));
        else
            return static_cast<T*>(0); // NULL
    }

    inline bool erase(const Key& key) {
        return 1 == hash_map<Key,T>::erase(key);
        // 'insert' ensures that 'hash_map<Key,T,HF,EQ>::erase(key)'
        // is either 0 or 1.
    }

    inline size_type size() const { return hash_map<Key,T>::size(); }
    inline size_type max_size() const { return hash_map<Key,T>::max_size(); }
    inline bool empty() const { return hash_map<Key,T>::empty(); }
    // Should care be taken to ensure that 'size_type' does not
    // overflow? Only a potential problem with 'max_size', since
    // 'idx_t' can easily hold the number of element to be sorted, and
    // thus the number of groups. Cf., 'Table_interface' and 'Global.h'.

}; /* STL_table */

/*******************************************************************\
  Creator_interface (abstract class)
\*******************************************************************/

/* Abstract class. Policy determining which implementation of
   'Table_interface' to use based on the constructor
   arguments. Cf. "Factory Method" of Gamma et al.: "Design Patterns",
   Addison-Wesley, ISBN 0-201-63361-2.

   Small tables can be realized without hashing, whereas this is
   sorely needed, if the index bit width is larger than, say, 16
   bits. The two implementations should not differ in
   interfaces. 'Creator_interface' achieves this. */

template<class Key, class T>
class Creator_interface {
```

```
public:
    typedef Table_interface<Key,T>::size_type size_type;
    typedef Table_interface<Key,T> table_t;

    Creator_interface(_QUAL(Global) width_t s_w,
                      _QUAL(Global) width_t mh_w,
                      _QUAL(Global) idx_t abs)
        : sep_width(s_w), max_hashed_width(mh_w), avg_bckt_sz(abs) {}
    ~Creator_interface () {}

    virtual table_t* factory_method(_QUAL(Global) width_t idx_width,
                                    size_type n) const = 0;
    // 'idx_width' is the width in bits of 'Key's. 'n' is an upper
    // bound on the number of elements that will be stored in the
    // table.

protected:

    const _QUAL(Global) width_t sep_width;
    const _QUAL(Global) width_t max_hashed_width;
    const _QUAL(Global) idx_t avg_bckt_sz; // Average bucket size.
    // If the 'idx_width' is <= 'sep_width', 'factory_method' returns a
    // simple 1-1 table. Otherwise a hash table is returned. For a hash
    // table which will potentially contain 'n' entries, the mapped
    // width of the hash function could be chosen as
    // 'min(max_hashed_width,log2_floor(n/avg_bckt_sz))'. But see the
    // derivates of 'Creator_interface' below.

}; /* Creator_interface */

/*******************************************************************\
  Own_impl_creator (concrete type)
\*******************************************************************/

// For 'Hash_table' with 'Hash_func':

template<class Key, class T>
class Own_impl_creator : public Creator_interface<Key,T> {

public:

    Own_impl_creator(_QUAL(Global) width_t s_w = 16,
                     _QUAL(Global) width_t mh_w = 19,
                     _QUAL(Global) idx_t abs = 2)
        : Creator_interface<Key,T>(s_w,mh_w,abs) {}
    ~Own_impl_creator() {}

    table_t* factory_method(_QUAL(Global) width_t idx_width,
                            size_type n) const
    {
        _USING_NMSPC(Global);

        if (idx_width <= sep_width) {
            Simple_table<Key,T>::bijector bijj(idx_width);
            return new Simple_table<Key,T>(bijj);
        }
        else {
            // Consider how to choose 'mapped_w'(the second arg to
            // 'Hash_func' constructor) and do it! The default 16 bits
```

```
            // might not suffice, when the order of magnitude of 'n'
            // is millions. The value of 'mapped_w' also influences
            // the allowed second template args to 'Hash_func'.
            width_t hashed_width = log2_floor(n/avg_bckt_sz);
            if ( hashed_width > max_hashed_width )
                hashed_width = max_hashed_width;
            Hash_table<Key,T>::hasher hash_funcc(idx_width,hashed_width);
            return new Hash_table<Key,T>(n,hash_funcc);
        }
        // Since "the semantics of argument passing are identical to
        // the semantics of initialisation", cf. Stroustrup p. 145, a
        // constructed table gets a !copy! of the
        // 'bijector'/'hasher'. The copy is destructed when the table
        // is destructed. The copying itself takes neglible time.
    }

}; /* Own_impl_creator */

/******************************************************************\
  STL_impl_creator (concrete type)
\******************************************************************/

// For 'hash_map' with 'Hash_func' hash function:

// template<class Key, class T>
// class STL_impl_creator : public Creator_interface<Key,T> {

// public:

//     STL_impl_creator(_QUAL(Global) width_t s_w = 16,
//                      _QUAL(Global) width_t mh_w = 19,
//                      _QUAL(Global) idx_t abs = 2)
//      : Creator_interface<Key,T>(s_w,mh_w,abs) {}
//     ~STL_impl_creator() {}

//     table_t* factory_method(_QUAL(Global) width_t idx_width,
//                             size_type n) const
//     {
//      _USING_NMSPC(Global);

//      if (idx_width <= sep_width) {
//          Simple_table<Key,T>::bijector bijj(idx_width);
//          return new Simple_table<Key,T>(bijj);
//      }
//      else {
//          // Consider how to choose 'mapped_w'(the second arg to
//          // 'Hash_func' constructor) and do it! The default 16 bits
//          // might not suffice, when the order of magnitude of 'n'
//          // is millions. The value of 'mapped_w' also influences
//          // the allowed second template args to 'Hash_func'.
//          width_t hashed_width = log2_floor(n/avg_bckt_sz);
//          if ( hashed_width > max_hashed_width )
//              hashed_width = max_hashed_width;
//          STL_table<Key,T>::hasher hash_funcc(idx_width,hashed_width);
//          return new STL_table<Key,T>(n,hash_funcc);
//      }
//      // Since "the semantics of argument passing are identical to
//      // the semantics of initialisation", cf. Stroustrup p. 145, a
//      // constructed table gets a !copy! of the
```

```
//      // 'bijector'/'hasher'. The copy is destructed when the table
//      // is destructed. The copying itself takes neglible time.
//     }

// }; /* STL_impl_creator */

// For 'hash_map' with 'hash' hash function:

template<class Key, class T>
class STL_impl_creator : public Creator_interface<Key,T> {

public:

    STL_impl_creator(_QUAL(Global) width_t s_w = 16,
                     _QUAL(Global) width_t mh_w = 19,
                     _QUAL(Global) idx_t abs = 2)
        : Creator_interface<Key,T>(s_w,mh_w,abs) {}
    ~STL_impl_creator() {}

    table_t* factory_method(_QUAL(Global) width_t idx_width,
                            size_type n) const
    {
        _USING_NMSPC(Global);

        if (idx_width <= sep_width) {
            Simple_table<Key,T>::bijector bijj(idx_width);
            return new Simple_table<Key,T>(bijj);
        }
        else {
            return new STL_table<Key,T>(n/avg_bckt_sz);
        }
        // Since "the semantics of argument passing are identical to
        // the semantics of initialisation", cf. Stroustrup p. 145, a
        // constructed table gets a !copy! of the
        // 'bijector'/'hasher'. The copy is destructed when the table
        // is destructed. The copying itself takes neglible time.
    }

}; /* STL_impl_creator */

_NMSPC_END(Sparse_table)

#endif // __SPARSE_TABLE_H

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Sparse_table.o"
//  compilation-scroll-output: t
//  End:
```

B.1.6 Sparse_table.cpp

```
// Filename: Sparse_table.cpp
// Author: Morten Nicolaj Pedersen

#pragma implementation

// #undef NDEBUG
#define NDEBUG
```

```
#include "Debug.h"
#include "Global.h"
#include "Sparse_table.h"
#include <vector>
#include <cfloat>          // DBL_MAX
#include <cstdlib>         // random and srandom
#include <cstddef>         // size_t

///////////////////////////////////////////////////////////////////////
// IMPLEMENTATION
///////////////////////////////////////////////////////////////////////

_USING_NMSPC(Global);
_USING_NMSPC(Debug);
_USING_NMSPC(Sparse_table);

/******************************************************************\
  Hash_func (concrete type)
\******************************************************************/

/*-----------------------------------------------------------------*\
  choose_a (auxiliary function)
\*-----------------------------------------------------------------*/

// Helper function for specializations immediately below. 'ui_t' is an
// unsigned integral type.

template<typename ui_t>
ui_t choose_a(width_t k, width_t l)
{
    VERIFY(k > 0 && l >= 1 && l <= k,
           "choose_a: 'k' or 'l' invalid\n");

    VERIFY(sizeof(unsigned long) > sizeof(ui_t),
           "choose_a: unsigned long too short for 'max + 1U'");
    VERIFY(8*sizeof(ui_t) >= k,
           "choose_a: ui_t too short for 'k'");
    ui_t max = static_cast<ui_t>((1UL << k) - 1U);
    /* The verifications ensure that the left shift and the cast are
       safe. */

    VERIFY(DBL_MAX >= max,"choose_a: potential truncation error in mult.\
                           of 'max' and random double from [0.0;1.0].");
    const int random_max = 2147483647U; // 2^^31 - 1. Cf., 'man random' page.
    ui_t a;
    do
        a=static_cast<ui_t>(max * (random()/static_cast<double>(random_max)));
       /* The expression in the outer parenthesis is a
          double. Immediately above it has been checked, that the
          mult. does not overflow. The cast is safe as well, since
          'max' is <=  the largest value an 'ui_t' can
          hold. */
    while (a == 0 || !(a & 1U));
    /* 'a' must be odd. Furthermore 'a' may not take the value
       'pow(2,k)', but this is automatically ensured, since 'max ==
       pow(2,k) - 1' is multiplied by a double in [0.0,1.0]. */
    return a;
```

```
} /* choose_a */

/*-----------------------------------------------------------------*\
  operator()
\*-----------------------------------------------------------------*/

template<class Key, typename m_type>
inline m_type Hash_func<Key,m_type>::operator() (const Key& key) const
{
    /* DEFINE! */
    VERIFY(false,"Hash_func operator(): generic version not implemented");
    return static_cast<m_type>(0);
} /* operator() */

// I want a partial specialization like:

// template<typename m_type>
// inline m_type Hash_func<unsigned int,m_type>::operator()
//     (const unsigned int& key) const
//     /* Specialization for 'unsigned int'. See Dietzfelbinger et al.:
//        "A Reliable Randomized Algorithm for the Closest-Pair Problem",
//        J. Algorithms 25 (1997), pp. 19-51. */
// {
//     static const unsigned int a = choose_a<unsigned int>(Key_w,mapped_w);
//     /* 'choose_a' also checks validity of 'Key_w' and 'mapped_w'. */
//     static const unsigned int mod_mask =
//      static_cast<unsigned int>((1UL << Key_w) - 1U);
//     static const width_t div_shift = Key_w-mapped_w;

//     VERIFY(!_QUAL(Debug) arg_check || (key <= mod_mask),
//         "Hash_func operator(): key too big");
//     return static_cast<m_type>((a*key & mod_mask) >> div_shift);
//     /* The verification is not necessary as far as the computation of
//        the hash value is concerned. The influence of bits in positions
//        greater than 'Key_w' is canceled by the bit-wise "and" with
//        'mod_mask'. The constructor ensures that the cast is safe. */

// } /* operator() */

// But the compiler and/or language does not support this. I therefore
// manually instantiate the partial specialization with 'unsigned
// short' and 'size_t'.

// unsigned short
inline unsigned short Hash_func<unsigned int,unsigned short>::operator()
    (const unsigned int& key) const
    /* Specialization for 'unsigned int'. See Dietzfelbinger et al.:
       "A Reliable Randomized Algorithm for the Closest-Pair Problem",
       J. Algorithms 25 (1997), pp. 19-51. */
{
    static const unsigned int a = choose_a<unsigned int>(Key_w,mapped_w);
    /* 'choose_a' also checks validity of 'Key_w' and 'mapped_w'. */
    static const unsigned int mod_mask =
        static_cast<unsigned int>((1UL << Key_w) - 1U);
    static const width_t div_shift = Key_w-mapped_w;

    VERIFY(!_QUAL(Debug) arg_check || (key <= mod_mask),
           "Hash_func operator(): key too big");
    return static_cast<unsigned short>((a*key & mod_mask) >> div_shift);
```

```
    /* The verification is not necessary as far as the computation of
       the hash value is concerned. The influence of bits in positions
       greater than 'Key_w' is canceled by the bit-wise "and" with
       'mod_mask'. The constructor ensures that the cast is safe. */

} /* operator() */

// unsigned int
inline unsigned int Hash_func<unsigned int,unsigned int>::operator()
    (const unsigned int& key) const
    /* Specialization for 'unsigned int'. See Dietzfelbinger et al.:
       "A Reliable Randomized Algorithm for the Closest-Pair Problem",
       J. Algorithms 25 (1997), pp. 19-51. */
{
    static const unsigned int a = choose_a<unsigned int>(Key_w,mapped_w);
    /* 'choose_a' also checks validity of 'Key_w' and 'mapped_w'. */
    static const unsigned int mod_mask =
        static_cast<unsigned int>((1UL << Key_w) - 1U);
    static const width_t div_shift = Key_w-mapped_w;

    VERIFY(!_QUAL(Debug) arg_check || (key <= mod_mask),
           "Hash_func operator(): key too big");
    return static_cast<unsigned int>((a*key & mod_mask) >> div_shift);
    /* The verification is not necessary as far as the computation of
       the hash value is concerned. The influence of bits in positions
       greater than 'Key_w' is canceled by the bit-wise "and" with
       'mod_mask'. The constructor ensures that the cast is safe. */

} /* operator() */

// size_t
inline size_t Hash_func<unsigned int,size_t>::operator()
    (const unsigned int& key) const
    /* Specialization for 'unsigned int'. See Dietzfelbinger et al.:
       "A Reliable Randomized Algorithm for the Closest-Pair Problem",
       J. Algorithms 25 (1997), pp. 19-51. */
{
    static const unsigned int a = choose_a<unsigned int>(Key_w,mapped_w);
    /* 'choose_a' also checks validity of 'Key_w' and 'mapped_w'. */
    static const unsigned int mod_mask =
        static_cast<unsigned int>((1UL << Key_w) - 1U);
    static const width_t div_shift = Key_w-mapped_w;

    VERIFY(!_QUAL(Debug) arg_check || (key <= mod_mask),
           "Hash_func operator(): key too big");
    return static_cast<size_t>((a*key & mod_mask) >> div_shift);
    /* The verification is not necessary as far as the computation of
       the hash value is concerned. The influence of bits in positions
       greater than 'Key_w' is canceled by the bit-wise "and" with
       'mod_mask'. The constructor ensures that the cast is safe. */

} /* operator() */

/******************************************************************\
  Hash_table (concrete type)
\******************************************************************/

/*----------------------------------------------------------------*\
  _find
```

```
\*-----------------------------------------------------------------------*/

template<class Key, class T, class HF, class EQ>
inline Hash_table<Key,T,HF,EQ>::Entry*
Hash_table<Key,T,HF,EQ>::_find(const Key& key)
{
    typename HF::mapped_type idx = hash_func(key);
    Entry* e_ptr;
    for (e_ptr = buckets[idx]; e_ptr; e_ptr = e_ptr->next)
        /* Invariant that pointed to 'Entry' is well
           defined. Cf. typedef of 'Entry'. */
        if (equal_op(e_ptr->key,key))
            break;
    return e_ptr;
} /* _find */

/*-----------------------------------------------------------------------*\
  find
\*-----------------------------------------------------------------------*/

template<class Key, class T, class HF, class EQ>
T* Hash_table<Key,T,HF,EQ>::find(const Key& key)
{
    cache = _find(key);
    if (cache) return &cache->data;
    else return 0; // NULL
} /* find */

/*-----------------------------------------------------------------------*\
  insert
\*-----------------------------------------------------------------------*/

template<class Key, class T, class HF, class EQ>
T* Hash_table<Key,T,HF,EQ>::insert(const Key& key, const T& data)
{
    _USING_NMSPC(Debug);

    Entry* e_ptr;
    const typename HF::mapped_type idx = hash_func(key);

    // Check for duplicates.
    if ( table_check ) {
        for ( e_ptr = buckets[idx]; e_ptr; e_ptr = e_ptr->next )
            if ( equal_op(e_ptr->key,key) )
//              VERIFY(false,"Hash table insert: duplicate");
                return 0; // NULL
    }

    // No duplicate found. Insert.
    VERIFY(!table_check || size_count < capacity,
           "Hash_table insert: overflow");
    // Verifies that the table can hold the new element.
    const vector<Entry>::size_type e_size = entries.size();
    if (e_size == capacity) {
        /* 'entries' "full". Find "hole" by next-alloc. */
        VERIFY(!table_check || erased_count,
               "Hash_table insert: hole does not exist");
        vector<Entry>::const_iterator e_end = entries.end();
        bool flag = false;
```

```
        while (pos->prev_ptr != 0) {
            ++pos;
            if (pos == e_end) {
                VERIFY(!table_check || !flag,
                       "Hash_table insert: cannot find hole");
                // Full circle must not happen twice!
                pos = entries.begin();
                flag = true;
            }
        }
        e_ptr = static_cast<Entry*>(pos);
        e_ptr->key = key;
        e_ptr->data = data;
        // Uses copy assignment operator of 'T'.
        erased_count--;
    }
    else {
        /* Important that 'entries.size()' is increased here. */
        entries.push_back(Entry(key,data));
        // Uses copy constructor of 'T'.
        e_ptr = static_cast<Entry*>(entries.end() - 1);
        /* Perhaps optimize here. Rather stupid to insert dummy 'Entry'
           and then define it below. Might as well init. with final
           value. */
    }
    VERIFY(!table_check || e_ptr,"Hash_table insert: e_ptr == NULL");
    // Actually too late to verify here, since 'e_ptr' has already
    // been used above. Oh well!

    if (buckets[idx]) {
        /* Make new element head of existing bucket. */
        buckets[idx]->prev_ptr = &e_ptr->next;
        e_ptr->prev_ptr = &buckets[idx];
        e_ptr->next = buckets[idx];
        buckets[idx] = e_ptr;
    }
    else {
        /* Init. bucket and insert new element into it. */
        e_ptr->next = 0;
        e_ptr->prev_ptr = &buckets[idx];
        buckets[idx] = e_ptr;
    }
    size_count++;

    return &e_ptr->data;

} /* insert */

/*------------------------------------------------------------------*\
  erase
\*------------------------------------------------------------------*/

template<class Key, class T, class HF, class EQ>
bool Hash_table<Key,T,HF,EQ>::erase(const Key& key)
{
    Entry* e_ptr;
    if (cache && equal_op(cache->key,key)) e_ptr = cache;
    else e_ptr = _find(key);
    /* I use 'e_ptr' here because I am not interested in updating cache
```

```
         to refer to an element, which has been deleted. The whole point
         of cache is to make erasure of an element, which was just
         searched for, fast. */

    if (e_ptr) {
        *(e_ptr->prev_ptr) = e_ptr->next;
        /* If element to be deleted has successor in linked chain: */
        if (e_ptr->next) (e_ptr->next)->prev_ptr = e_ptr->prev_ptr;
        e_ptr->prev_ptr = 0; /* Mark as erased. */
        /* Not necessary to destruct data of entry explicitly. The 'T'
           destructor will be called implicitly when the table is
           destructed. */
        erased_count++;
        size_count--;
        return true;
    }
    else return false;
} /* erase */

/*******************************************************************\
  One_one (concrete type)
\*******************************************************************/

/*-----------------------------------------------------------------*\
  operator()
\*-----------------------------------------------------------------*/

template<class Key, typename r_type>
inline r_type One_one<Key, r_type>::operator()
    (const Key& key) const
{
    /* DEFINE! */
    VERIFY(false,"One_one operator(): generic version not implemented\n");
    return static_cast<r_type>(0);
} /* operator() */

inline unsigned short One_one<unsigned int,unsigned short>::operator()
    (const unsigned int& key) const
{
    VERIFY(!_QUAL(Debug) arg_check || (key < range_sz),
           "One_one operator(): key too big");
    return static_cast<unsigned short>(key);
} /* operator() */

/*******************************************************************\
  Simple_table (concrete type)
\*******************************************************************/

/*-----------------------------------------------------------------*\
  find
\*-----------------------------------------------------------------*/

template<class Key, class T, class BIJ>
T* Simple_table<Key,T,BIJ>::find(const Key& key)
{
    typename BIJ::range_type idx = bij(key);
    if (entries[idx].present) return &entries[idx].data;
    else return 0; // NULL
} /* find */
```

```
/*----------------------------------------------------------------------*\
  insert
\*----------------------------------------------------------------------*/

template<class Key, class T, class BIJ>
T* Simple_table<Key,T,BIJ>::insert(const Key& key, const T& data)
{
    typename BIJ::range_type idx = bij(key);
    if (entries[idx].present) return 0;
    else {
        entries[idx].present = true;
        entries[idx].data = data;
        // Uses copy assignment operator of 'T'.
        size_count++;
        return &entries[idx].data;
    }
} /* insert */

/*----------------------------------------------------------------------*\
  erase
\*----------------------------------------------------------------------*/

template<class Key, class T, class BIJ>
bool Simple_table<Key,T,BIJ>::erase(const Key& key)
{
    typename BIJ::range_type idx = bij(key);
    if (entries[idx].present) {
        entries[idx].present = false;
        /* Not necessary to destruct data of entry explicitly. The 'T'
           destructor will be called implicitly when the table is
           destructed. */
        size_count--;
        return true;
    }
    else return false;
} /* erase */

////////////////////////////////////////////////////////////////////////
// EXPLICIT INSTANTIATION
////////////////////////////////////////////////////////////////////////

#include "Range_redex.h" // Grp_info

_NMSPC_BEGIN(Sparse_table)

template class Hash_func<unsigned int,unsigned short>;
template class Hash_table<unsigned int,
    _QUAL(Range_redex) _QUAL(KR_impl) Grp_info>;
// Needed by 'Own_impl_creator<unsigned int,Grp_info>' which is used
// in the explicit instantiation 'KR<word_iterator,
// Own_impl_creator<unsigned int,Grp_info> >' in 'Range_redex.cpp'.

template class Hash_func<unsigned int,size_t>;
template class STL_table<unsigned int,
    _QUAL(Range_redex) _QUAL(KR_impl) Grp_info>;
// Needed by 'STL_impl_creator<unsigned int,Grp_info>' which is used
// in the explicit instantiation 'KR<word_iterator,
// STL_impl_creator<unsigned int,Grp_info> >' in 'Range_redex.cpp'.
```

```
template class One_one<unsigned int, unsigned short>;
template class Simple_table<unsigned int,
    _QUAL(Range_redex) _QUAL(KR_impl) Grp_info>;
// Needed by both the explicit instantiations in 'Range_redex.cpp'
// mentioned above.

_NMSPC_END(Sparse_table)

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Sparse_table.o"
//  compilation-scroll-output: t
//  End:
```

B.1.7 Range_redex.h

```
// Filename: Range_redex.h
// Author: Morten Nicolaj Pedersen
/* Remarks: In the comments in this source file 'keys[i]' is sometimes
   used to denote '*(first + i)'. */

#ifndef __RANGE_REDEX_H
#define __RANGE_REDEX_H

#pragma interface

#include "Debug.h"
#include "Global.h"
#include "Sparse_table.h"
#include <utility>        // pair
#include <vector>

///////////////////////////////////////////////////////////////////////
// DECLARATIONS
///////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(Range_redex)

/*******************************************************************\
  RR_interface (abstract class)
\*******************************************************************/

/* The generic range changer. Implements its public interfac, i.e.,
   'sort', by means of the protected functions 'reduce', 'expand',
   'short_keys_sort', 'init', and 'cleanup', which derivates must
   define. 'sort' permutes 'keys' into ascending (better:
   non-descending) order. 'Ran' is a random access iterator, which is
   verified by the constructor. */

template<class Ran>
class RR_interface {

public:

    RR_interface() {
        VERIFY(typeid(iterator_traits<Ran>::iterator_category) ==
               typeid(random_access_iterator_tag),
               "RR_interface: sort: 'Ran' not random access iterator");
```

```
         VERIFY(typeid(iterator_traits<Ran>::value_type) ==
                typeid(word),
                "RR_interface: sort: '*Ran' not 'word'");

    }
    virtual ~RR_interface() {} /* Destructor. */

    void sort(Ran first, Ran last, uchar s);
    /* Not virtual! This is where the invocations of the 'protected
       virtual' functions are hardwired into 'RR_interface'. Cf. the
       implementation in 'Range_redex.cpp'. In summary 'sort' performs
       's' reductions steps (by calling 'reduce' once), then sorts the
       short keys thus obtained by calling 'short_keys_sort', and
       finally uses the sorted short keys to produce in 'keys' the
       original keys in sorted order (by calling 'expand' once).

       The 'protected virtual' functions 'init' and 'cleanup'
       administrate the environment, except for 'steps' and 'size',
       necessary for derivates to implement the three functions
       mentioned immediately above.

       'Ran' is a random access iterator of 'word's. This is
       verified. */

protected:

    uchar steps; /* Holds the 'steps' argument of 'sort'. */
    _QUAL(Global) idx_t size;  /* 'last - first'. */

    virtual void reduce(Ran first, Ran last) = 0;
    virtual void short_keys_sort(Ran first, Ran last) = 0;
    virtual void expand(Ran first, Ran last) = 0;
    /* 'reduce' narrows the key width of the orginal sorting problem
       represented by 'first' and 'last', and saves the resulting
       problem consisting of shorter keys, and their widths,
       internally. They are sorted by invoking
       'short_keys_sort'. Thereafter 'expand' can by relied on to
       return the original keys in sorted order.

       Note that the user cannot call any of these three functions
       directly. They are applied by 'sort' in the manner, which is
       hardwired into the implementation of 'sort'.

       The interfaces of the functions are thin. All auxiliary
       variables, except for 'steps' and 'size', are handled by 'init'
       and 'cleanup'. This makes for maximal freedom when deriving
       from 'RR_interface'. */

    virtual void init(_QUAL(Global) idx_t sz) = 0;
    virtual void cleanup() = 0;
    /* These two functions take care of the environment that derivates
       need in order to implement 'reduce', 'short_keys_sort', and
       'expand'. This administration cannot happen exclusively in the
       constructor/destructor of the derived class, unless the number
       of steps and the cardinality of the range represented by
       'first' and 'last' is hardwired into the object. I choose not
       to do so. */

}; /* RR_interface */
```

```
/******************************************************************\
  KR (concrete type)
\******************************************************************/

_NMSPC_BEGIN(KR_impl);

struct Grp_info {
    _QUAL(Global) idx_t idx; // Index of "extreme" key/element of group.
    _QUAL(Global) idx_t sz;  // Size of group.
    // 'idx' is undef. iff 'sz == 0'.

    Grp_info(_QUAL(Global) idx_t i = 0, _QUAL(Global) idx_t s = 0)
        : idx(i), sz(s) {}

    bool operator==(const Grp_info& rhs) const {
        return idx == rhs.idx && sz == rhs.sz;
    }
};
/* The type to use for 'T', i.e., data, in the sparse tables in
   'KR'. Placed outside of class since it is needed in definition of
   the default value of the second template argument. Also used in
   explicit instantiations in 'Sparse_table.cpp'. Instead of 'idx_t
   idx' one could use 'Ran pos'. 'Ran' is an 8 byte pointer on the
   Alpha though, whereas 'idx_t' is only 4 bytes.

   The implicit, default essential operators, cf.,
   'Table_interface' in 'Sparse_table', which are generated by the
   compiler are sufficient, since the data members are
   built-ins. */

_NMSPC_END(KR_impl);

/* Cf. Rahman and Raman: "An experimental study of word-level
   parallelism in some sorting algorithms", Proceedings WAE'98,
   pp. 193-203, for a somewhat detailed description of the
   implementation of this derivate of 'RR_interface'. The general idea
   of exponential range reduction originates from Kirkpatrick and
   Reisch: "Upper bounds for sorting integers on random access
   machines", Theo. Comput. Sci. 28 (1984), pp. 263-276.

   'Creator' is a derivate of
   'Sparse_table::Creator_interface<unsigned int,Grp_info>'. I
   introduced it as template argument because I got an internal
   compiler error when trying to achieve the same with run-time
   polymorphism, i.e., an additional constructor argument of type
   'Sparse_table::Creator_interface<unsigned int,Grp_info>*'. A
   'Creator' has a 'factory_method' member function, which returns a
   derivate of 'Sparse_table::Table_interface'. The exact type of the
   derivate depends on the arguments passed to 'factory_method'. */

template<class Ran, class Creator>
class KR : public RR_interface<Ran> {

public:

    typedef void (*indirect_sorter)(_QUAL(Global) word_iterator first,
                                    _QUAL(Global) word_iterator last,
                                    _QUAL(Global) width_t width,
```

```
                                    _QUAL(Global) width_t base_shift);
    /* The pointer to function type of the first constructor
       argument. The post-condition is that the full 'w'-bit keys in
       the range 'first' to 'last' are sorted on the value of the bit
       pattern obtained by right shifting 'base_shift' places and
       masking out the 'width' least significant bits. */

    KR(indirect_sorter idss) : ids(idss)
    {
        /* Ought to check that 'Creator' is a derivate of
           'Sparse_table::Creator_interface<unsigned
           int,Grp_info>'. */

        _USING_NMSPC(Global);

        VERIFY(_QUAL(Global) log2_floor(w) <= 8*sizeof(uchar),
               "KR: type of 'phs' potentially too narrow");
        VERIFY(w <= ((1UL << (8*sizeof(width_t))) - 1U),
               "KR: type of 'shift_dists' and 'width' potentially too narrow");
        VERIFY(8*sizeof(unsigned int)*2 == w,
               "KR: unsigned int not suitable for 'Key'");
        // Since 'KR' implements exponential range reduction, the
        // maximum bit width of a table index is 'w/2'.
    }

    ~KR() {} /* 'init' and 'cleanup' take care of dynamic
                (de)allocation. Nothing to do in the destructor. */

protected:

    // These five functions are inherited from 'RR_interface'.
    void reduce(Ran first, Ran last);
    void short_keys_sort(Ran first, Ran last);
    void expand(Ran first, Ran last);

    void init(_QUAL(Global) idx_t sz);
    void cleanup();

private:

    indirect_sorter ids;   // Used in 'short_keys_sort'.

    vector<uchar> phs; // Promotion histories.
    vector<uchar>::iterator phs_pos, phs_first;
    static const _QUAL(Global) word zero = 0U, one = 1U;
    /* 'phs[i]' records in a bit-flag manner whether 'keys[i]'
       contributed its "range" (high-order bits), flag cleared, or
       "value" (low-order bits), flag set, in each reduction step. The
       invariant maintained is that !after! !reduction!  step 'n', the
       'n' LSBs of 'phs[i]' have meaning and its other bits are
       cleared per default. The LSB corresponds to step 'n' and so
       forth. The interpretation is that iff 'phs[i] < phs[j]' then
       'keys[i]' comes before 'keys[j]' in the non-descending sorted
       output. Furthermore a cleared bit signifies, that 'keys[i]' was
       "promoted" during the corresponding step; it was an "extreme"
       element of its "group" (the set of keys with the same promotion
       history as 'keys[i]' upon entering the step).

       When !expanding! 'phs' is not used. */
```

```

    typedef pair<_QUAL(Global) idx_t,_QUAL(Global) idx_t> sz_pair;
    vector<sz_pair> grp_sizes;
    vector<sz_pair>::iterator grp_end,grp_pos;
    _QUAL(Global) idx_t no_of_pairs;
    /* A 'sz_pair' is interpreted as (#promoted,#demoted), and each
       (one) 'sz_pair' corresponds to a single "group" as defined
       above. 'idx_t' is guarenteed to be large enough, cf.,
       'Global.h'. During !reduction! 'no_of_pairs' is used to store
       the number of pairs to reduce from in the current step (0 for
       the first, 1 for the second, 2 for the third, and so
       forth). This corresponds to 1 group (all keys), 2 groups, 4
       groups, and so forth. 'no_of_pairs' is !not! a cache for
       'grp_sizes.size()'. This is because 'grp_sizes' accumulates
       'sz_pair's, i.e., after the third step its size is 1 + 2 +
       4. During !expansion! the interpretations are analogous. */

    Creator table_creator;
    vector<typename Creator::table_t*> tables; // See immediately above.
    vector<typename Creator::table_t*>::iterator finger; // Pos. in 'tables'.
    /* Must use pointers here to allow polymorphism. 'tables' is
       parallel to 'grp_sizes'. The 'i'th table corresponds to the
       'i'th 'sz_pair'/"group". */

    _QUAL(Global) word_container masks;
    vector<_QUAL(Global) width_t> shift_dists;
    /* The expression 'keys[i] & masks[phs[i]]' yields the currently
       interesting bits of 'keys[i]'. That is 'w/pow(2,l)' consecutive
       bits !during! !reduction! step 'l' (steps are numbered from
       0). These are the bits that the key contributed to the current
       step.

       Further shifting these bits to the right by shift_dists[phs[i]]
       places the high-order half of them in the LSBs of the result,
       which can then be used as table index to pair keys on the
       "range" of the contributed bits.

       Both names are updated when starting a new step. They are not
       well-defined/used during step 0.

       During !expansion! they have a sligtly different
       interpretation. */

    _QUAL(Global) word_container buffer;
    _QUAL(Global) word_iterator buffer_pos, buffer_first;
    /* Some auxiliary working space is needed, because a
       unification/expansion cannot be performed in-place. */

    _QUAL(Global) width_t width;
    /* Number of bits each key contributes during the current
       step. 'width' must always be even. Guaranteed when initialized
       by 'w' in 'RR_interface::sort', since 'w ==
       8*sizeof(word)'. With some extra effort 'KR' can be implemented
       so that 'width' is allowed to be odd. */

}; /* KR */

_NMSPC_END(Range_redex)
```

```
#endif // __RANGE_REDEX_H

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Range_redex.o"
//  compilation-scroll-output: t
//  End:
```

B.1.8 Range_redex.cpp

```
// Filename: Range_redex.cpp
// Author: Morten Nicolaj Pedersen
/* Remarks: In the comments in this source file 'keys[i]' is sometimes
   used to denote '*(first + i)'. 'KR::reduce/expand' assumes that 'w'
   is even. */

#pragma implementation

// #undef NDEBUG
#define NDEBUG

#include "Debug.h"
#include "Global.h"
#include "Range_redex.h"
#include "Sparse_table.h"
#include <typeinfo>   // typeid, type_info
#include <iterator>   // iterator_traits
#include <vector>

////////////////////////////////////////////////////////////////////////
// IMPLEMENTATION
////////////////////////////////////////////////////////////////////////

_USING_NMSPC(Global);
_USING_NMSPC(Sparse_table);
_USING_NMSPC(Range_redex);
_USING_NMSPC(KR_impl);

/*******************************************************************\
  RR_interface (abstract class)
\*******************************************************************/

/*-----------------------------------------------------------------*\
  sort
\*-----------------------------------------------------------------*/

template<class Ran>
void RR_interface<Ran>::sort(Ran first, Ran last, uchar s)
{
    VERIFY(s > 0, "RR_interface: sort: 'steps' must be positive");
    steps = s;
    size = static_cast<idx_t>(last - first);
    init(size);
    reduce(first,last);
    short_keys_sort(first,last);
    expand(first,last);
    cleanup();
} /* sort */
```

```
/*********************************************************************\
  KR (concrete type)
\*********************************************************************/

/*-------------------------------------------------------------------*\
  init
\*-------------------------------------------------------------------*/

template<class Ran,class Creator>
void KR<Ran,Creator>::init(idx_t sz)
{
    phs = vector<uchar>(sz);
    phs_first = phs.begin();

    grp_sizes.reserve((1U << steps) - 1U);
    // 'grp_sizes' grows ('push_back') in 'reduce' and shrinks
    // ('pop_back') in 'expand'. The call of 'reserve' ensures that
    // loop (A) is safe.

    tables = vector<typename Creator::table_t*>((1U << steps) - 1U);
    finger = tables.begin();
    // Size of 'tables' is '2^^steps - 1'.

    masks = word_container(1U << (steps-1));
    shift_dists = vector<width_t>(1U << (steps-1));
    // Size of 'masks' and 'shift_dists' is '2^^(steps-1)'.

    buffer = word_container(sz);
    buffer_first = buffer.begin();

    width = w;
}

/*-------------------------------------------------------------------*\
  cleanup
\*-------------------------------------------------------------------*/

template<class Ran,class Creator>
void KR<Ran,Creator>::cleanup()
{
    // The 'erase' (and thus 'clear') member function invokes the
    // destructor for the specified elements.
    phs.clear();
    VERIFY(grp_sizes.empty(),
           "RR_interface: cleanup: 'grp_sizes' not empty");
    tables.clear();
    masks.clear();
    shift_dists.clear();
    buffer.clear();
}

/*-------------------------------------------------------------------*\
  reduce
\*-------------------------------------------------------------------*/

// Relies on 'init' having just been called, so that various
// 'vector's are filled with zeroes.

template<class Ran,class Creator>
```

```
void KR<Ran,Creator>::reduce(Ran first, Ran last)
{
    int i;
    Ran pos;
    Grp_info* gi_ptr;
    width_t dist;
    word mask;

    /* First halving. Special case: */
    *finger = table_creator.factory_method(width/2, size); // Inherited 'size'
    dist = width / 2;
    // The 'private' variable 'width'. Set to 'w' in 'init'. Thus
    // 'width/2' is the number of bits to reduce !to!.
    for (pos = first, phs_pos = phs_first, i = 0;
         pos != last;
         ++pos, ++phs_pos, ++i) {

        typename Creator::table_t::key_type high = *pos >> dist;
        // No cast needed. Cf., constructor.
        if (gi_ptr = (*finger)->find(high)) {
            /* Group already represented. */
            if (*pos < *(first + gi_ptr->idx)) {
                /* 'keys[i]' is new "extreme" element of
                   group. Promote it! Note that it is the full 'word's
                   that are compared. OK since it is the most
                   significant half that is interesting. */
                *(phs_first + gi_ptr->idx) = one; /* Demote old "extreme". */
//              *phs_pos = zero; /* Promote new "extreme". */
                gi_ptr->idx = i;
            }
            else {
                /* 'keys[i]' does not get promoted. */
                *phs_pos = one;
            }
            gi_ptr->sz++;
        }
        else {
            /* Group does not pre-exist. Create it! The promotion
               history was cleared by 'init', so the default promotion
               is already registered. */
            gi_ptr = (*finger)->insert(high,Grp_info(i,1));
        }
    }
    VERIFY(grp_sizes.empty(), "KR reduce: 'grp_sizes' not empty initially");
    no_of_pairs = 1;
    grp_sizes.push_back(sz_pair((*finger)->size(),
                                size - (*finger)->size()));
    ++finger; /* Move 'finger' to unused location. */

    /* Remaining halvings. General case: */
    for (int s = 1; s < steps; s++) {

        width /= 2; // Now holds the "source" width to reduce !from!.

        /* 'no_of_pairs' holds the number of pairs to split. Updated
           at end of loop. The number of groups is twice the number of
           pairs. */

        /* Prepare. */
```

```
dist = width / 2;
mask = (1U << width) - 1U; /* Low-order 'width' bits. */
for (int j = 2*no_of_pairs - 1; j >= 0; j--) {
    shift_dists[j] = dist;
    dist += width;
    masks[j] = mask;
    mask <<= width;
    // Cf., comments on 'shift_dists' and 'masks' in
    // 'Range_redex.h'. They pick out the high-order bits of
    // the "source" bits.
}

grp_end = grp_sizes.end();
for (grp_pos = grp_end - no_of_pairs; grp_pos != grp_end; ++grp_pos) {

    *finger++ = table_creator.factory_method(width/2,grp_pos->first);
    *finger++ = table_creator.factory_method(width/2,grp_pos->second);
    /* The tables to use for splitting the promoted and
       demoted elements respectively of the group
       corresponding to the 'sz_pair' pointed to by
       'grp_pos'. */
}
finger -= 2*no_of_pairs;
/* Position 'finger' such that indexing it with a promotion
   history picks the corresponding table pointer. */

/* Actually perform duplicate identification of high order
   bits. */
for (pos = first, phs_pos = phs_first, i = 0;
     pos != last;
     ++pos, ++phs_pos, ++i) {

    word bits = *pos & masks[*phs_pos];
    typename Creator::table_t::key_type high =
        bits >> shift_dists[*phs_pos];
    if (gi_ptr = finger[*phs_pos]->find(high)) {
        /* Entry exists. */
        if (bits <  (*(first + gi_ptr->idx) & masks[*phs_pos])) {
            /* New "extreme". Promote it! */
            *(phs_first + gi_ptr->idx) |= one; /* Demote. */
            *phs_pos <<= 1; /* Shifts in 0. */
            gi_ptr->idx = i;
        }
        else {
            /* No promotion. */
            *phs_pos = (*phs_pos << 1) | one;
        }
        gi_ptr->sz++;
    }
    else {
        /* Entry must be inserted. */
        gi_ptr = finger[*phs_pos]->insert(high,Grp_info(i,1));
        *phs_pos <<= 1; /* Promote as default. Shifts in 0. */
    }
}

/* Push new 'sz_pair's corresponding to the tables used
   immediatly above onto 'grp_sizes'. Advance 'finger' in
   preparation of the next iteration. */
```

```
        grp_end = grp_sizes.end();
        for (grp_pos = grp_end - no_of_pairs; grp_pos != grp_end; ++grp_pos) {

            /* Label (loop): (A). */

            grp_sizes.push_back(sz_pair((*finger)->size(),
                                        grp_pos->first - (*finger)->size()));
            ++finger;

            grp_sizes.push_back(sz_pair((*finger)->size(),
                                        grp_pos->second - (*finger)->size()));
            ++finger;
            // Note that 'sz_pair's corresponding to groups with
            // larger promotion histories are placed nearer the end of
            // 'grp_sizes'.
        }
        /* 'finger' now points to unused position in 'tables'. */

        no_of_pairs <<= 1; /* Double. */
    }

    width /= 2;
    /* Update 'width' to reflect the last iteration. */

    /* On exit, 'keys' (actually 'first' and 'last'), 'width', and
       'phs' describe the recursive short keys sorting problem, cf.,
       'short_keys_sort'. Given the solution of the recursive problem
       the information in 'no_of_pairs', 'tables', and 'grp_sizes'
       facilitates solving the original problem, cf., 'expand'.

       Note that the number of groups is '2*no_of_pairs', as a pair
       induces two groups, one for promoted keys and one for demoted
       keys. */

} /* reduce */

/*-------------------------------------------------------------------*\
  short_keys_sort
\*-------------------------------------------------------------------*/

template<class Ran,class Creator>
void KR<Ran,Creator>::short_keys_sort(Ran first, Ran last)
{
    /* MSD radix sort of the reduced keys on '(phs[i], "contrib"[i])',
       where "contrib"[i] is the bit pattern that 'keys[i]' contribute
       to the recursive sorting problem. */

    Ran pos;

    // First sort on 'phs[i]':

    /* Compute group_iterators. */
    vector<word_iterator> group_iterators;
    buffer_pos = buffer_first;
    grp_end = grp_sizes.end();
    for (grp_pos = grp_end - no_of_pairs; grp_pos != grp_end; ++grp_pos) {
        /* Compute start index of each group. */
        group_iterators.push_back(buffer_pos);
        buffer_pos += grp_pos->first;
```

```
        group_iterators.push_back(buffer_pos);
        buffer_pos += grp_pos->second;
        // Cf., (A).
    }
    VERIFY(buffer_pos == buffer.end(),
           "KR short_keys_sort: the sum of group sizes is wrong ");

    /* Order on 'phs[i]'. */
    const idx_t no_of_groups = 2*no_of_pairs;
    VERIFY(no_of_groups * width == w,
           "KR short_keys_sort: 'no_of_groups * width == w' not true"); // (B)
    for (pos = first, phs_pos = phs_first;
         pos != last;
         ++pos, ++phs_pos) {


        VERIFY(!_QUAL(Debug) redex_check || *phs_pos < no_of_groups,
               "KR short_keys_sort: promotion history too large");
        // 'group_iterators.size() == 2*no_of_pairs == no_of_groups' .
        *group_iterators[*phs_pos]++ = *pos;
    }
    /* The iterators in 'group_iterators' now point one entry beyond the end
       of the corresponding "group". */

    // Second sort each group with 'ids'.
    buffer_pos = buffer_first;
    width_t base_shift = w;
    /* The first 'group_iterators' entry corresponds to the group of keys,
       which were promoted at each level of reduction. They contribute
       their 'width' most significant bit positions to the recursive
       sorting problem. The next 'group_iterators' entry/group contribute
       the next 'width' bit positions and so on. Cf., (A). */
    for (vector<word_iterator>::iterator g_it = group_iterators.begin();
         g_it != group_iterators.end(); ++g_it) {

        base_shift -= width; // Cf., (B).
        ids(buffer_pos,*g_it,width,base_shift);
//      VERIFY( !_QUAL(Debug) redex_check ||
//              is_sorted(buffer_pos,*g_it,
//                        Cmp_bit_pattern(make_mask(width,base_shift))),
//              "KR short_keys_sort: group not sorted" );
        buffer_pos = *g_it;
    }

    group_iterators.clear();

    // The result of the MSD sort now resides in 'buffer'. 'phs' has
    // not been reordered to reflect the new order of keys in
    // 'buffer'.
}

/*-------------------------------------------------------------------*\
  expand
\*-------------------------------------------------------------------*/

template<class Ran,class Creator>
void KR<Ran,Creator>::expand(Ran first, Ran last)
{
    /* -----> 'phs' is not used in 'expand' <----- */
```

```

    /* Now unite neighboring (in 'buffer') groups as dictated by
       'grp_sizes' and 'tables' until only one remains. */

    VERIFY(typeid(Ran) == typeid(word_container::iterator),
           "KR expand: 'Ran' and 'word_container::iterator' not same type");

    Ran src_first = buffer_first, dest_first = first;
    Grp_info* gi_ptr;

    /* All expansions in one loop: */
    for (int s = steps; s > 0; s--) {

        /* On entry 'width' is the "source" width. */

        /* Prepare step. Init. 'masks' and 'shift_dists'. Even though
           "contrib[i]", cf. 'short_keys_sort', is the "value", the
           "range" must still be used as table index. The
           init. reflects this. */
        width_t dist = width;
        word mask = (1U << width) - 1U;
        mask |= mask << width;
        width <<= 1; /* Now the "destination" width! */
        for (int j = no_of_pairs - 1; j >= 0; j--) {
            masks[j] = mask;
            mask <<= width;
            shift_dists[j] = dist;
            dist += width;
        }

        /* Perform step. */

        /* The sorting in 'short_keys_sort' above ensures that the
           promotion histories of the keys in the source container are
           non-decreasing, since the keys are grouped. Furthermore the
           'first' and 'second' coordinates of the 'sz_pair's in
           'grp_sizes' hold the group sizes. These invariants are
           maintained in the loop below. */
        Ran src_pos = src_first, dest_pos = dest_first;
        finger -= no_of_pairs;
        uchar dest_grp_idx = 0;
        idx_t base = 0;
        grp_end = grp_sizes.end();
        for (grp_pos = grp_end - no_of_pairs; grp_pos != grp_end; ++grp_pos) {

            // If 'grp_pos->second == 0' these next two loops are not
            // necessary. A straightforward copy of 'grp_pos->first'
            // keys into the destination suffices, since all
            // destination group sizes are one.

            /* A lot of similarity in the next two loops. Exploit it
               to reduce source lines? */
            for (int k = 0; k < grp_pos->first; k++) {
                /* Place "extreme" elements in destination. */
                word bits = *src_pos & masks[dest_grp_idx];
                typename Creator::table_t::key_type high =
                    bits >> shift_dists[dest_grp_idx];
                VERIFY(gi_ptr = (*finger)->find(high),"KR expand: not found");
                gi_ptr->idx = base;
```

```
                /* The 'idx' field of 'Grp_info' is now used to index
                   destination container. */
                base += gi_ptr->sz;
                dest_first[gi_ptr->idx++] = *src_pos++;
            }

            for (int k = 0; k < grp_pos->second; k++) {
                /* Place corresponding non-"extreme" elements. */
                word bits = *src_pos & masks[dest_grp_idx];
                typename Creator::table_t::key_type high =
                       bits >> shift_dists[dest_grp_idx];
                VERIFY(gi_ptr = (*finger)->find(high),"KR expand: not found");
                dest_first[gi_ptr->idx++] = *src_pos++;
            }
            delete *finger++;

            dest_grp_idx++;
        }

        /* Possibly check expansion step: */
//      if (_QUAL(Debug) redex_check) {
//          width_t base_shift = w;
//          Ran check_pos = dest_first;
//          grp_end = grp_sizes.end();
//          for (grp_pos = grp_end - no_of_pairs;
//               grp_pos != grp_end; ++grp_pos) {

//              base_shift -= width; /* 'width' is "destination" width. */
//              idx_t grp_size_cache = grp_pos->first + grp_pos->second;
//              VERIFY(is_sorted(check_pos,check_pos+grp_size_cache,
//                                Cmp_bit_pattern(make_mask(width,
//                                                          base_shift))),
//                     "KR expand: step failed");
//              check_pos += grp_size_cache;
//          }
//      }

        /* Prepare for next step. */
        finger -= no_of_pairs;

        for (int j = 0; j < no_of_pairs; j++) grp_sizes.pop_back();
        no_of_pairs >>= 1; /* Halve. */

        swap(src_first,dest_first);
    }

    VERIFY(grp_sizes.empty(), "KR reduce: 'grp_sizes' not emptied");

    if (!(steps & one)) {
        /* If 'steps' is even, the final destination was not 'keys',
           but 'buffer'. Remedy this: */
        Ran pos;
        for (pos = first, buffer_pos = buffer_first; pos != last; )
            *pos++ = *buffer_pos++;
    }

} /* expand */

//////////////////////////////////////////////////////////////////////
```

```
// EXPLICIT INSTANTIATION
//////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(Range_redex)

template class RR_interface<word_iterator>;

template class KR<word_iterator, STL_impl_creator<unsigned int,Grp_info> >;
// Needed by 'CVISP::KR_sort<word_iterator>'.

template class KR<word_iterator, Own_impl_creator<unsigned int,Grp_info> >;
// Needed by 'CVISP::KR_sort<word_iterator>'.

_NMSPC_END(Range_redex)

//  Local Variables:
//  compile-command: "gmake /usr/tmp/Range_redex.o"
//  compilation-scroll-output: t
//  End:
```

B.1.9 CVISP.h

```
// Filename: CVISP.h
// Author: Morten Nicolaj Pedersen
/* Remarks: */

#ifndef __CVISP_H
#define __CVISP_H

#pragma interface

#include "Global.h"
#include <algorithm> // __stl_threshold

//////////////////////////////////////////////////////////////////////
// DECLARATIONS
//////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(CVISP)

/********************************************************************\
  typedefs and constants
\********************************************************************/

const _QUAL(Global) width_t w_copy = _QUAL(Global) w;
// For the default value of the 'width' argument of 'contender'. The
// compiler cannot handle 'width = _QUAL(Global) w'.

/********************************************************************\
  Functions
\********************************************************************/

template<class Ran>
void contender(Ran first, Ran last,
               _QUAL(Global) width_t width = w_copy,
               _QUAL(Global) width_t base_shift = 0,
               _QUAL(Global) width_t max_radix = 9,
               _QUAL(Global) idx_t threshold = __stl_threshold);
```

```
// Implements Adative Radixsort with an upper bound 'max_radix' on the
// radix. Switches to Insertion Sort if the number of keys is less
// than 'threshold'. 'Ran' is a random access iterator of
// 'word's. This is verified. Sorts the keys in the range 'first' to
// 'last' on the value of the !bit pattern! masked out by '((1U <<
// width) - 1U) << base_shift', cf., 'Global::make_mask'. The default
// is to sort on the complete key value.

template<class Ran>
inline void contender_wrapper(Ran first, Ran last) { contender(first,last); }
// Introduced because the 'width' and 'base_shift' arguments of
// 'contender' are not properly set to their default values if the
// 'sf' function pointer in 'invocate' in 'driver.cpp' is set to
// '&contender'.

template<class Ran>
void KR_sort(Ran first, Ran last, uchar RR_steps = 1);
// Exponential range reduction ('RR_steps' steps) complemented by
// 'contender'.

template<class Ran>
inline void KR_sort_wrapper(Ran first, Ran last) { KR_sort(first,last); }
// Cf., 'contender_wrapper' above.

inline void KR_sort_helper(_QUAL(Global) word_iterator first,
                           _QUAL(Global) word_iterator last,
                           _QUAL(Global) width_t width,
                           _QUAL(Global) width_t base_shift)
{
    contender(first,last,width,base_shift,9,__stl_threshold);
}
// Introduced to help the compiler with default arguments. Cf.,
// 'Range_redex::KR::indirect_sorter' and 'KR_sort'. Compare default
// values '9, __stl_threshold' with 'contender' prototype above.

_NMSPC_END(CVISP)

#endif // __CVISP_H

//  Local Variables:
//  compile-command: "gmake /usr/tmp/CVISP.o"
//  compilation-scroll-output: t
//  End:
```

B.1.10 CVISP.cpp

```
// Filename: CVISP.cpp
// Author: Morten Nicolaj Pedersen
/* Remarks: The code below uses internals ('__insertion_sort') of the
   Copyright (c) 1994 Hewlett-Packard Company and Copyright (c)
   1996,1997 Silicon Graphics Computer Systems, Inc. implementation of
   the STL which accompanies EGCS version 2.91.66 (release 1.1.2
   03-14-1999).

   Hence, the code relies on properties of the HP/SGI implementation
   of the STL, which are not guaranteed by the C++ Standard. These
   properties are explicitly described at the relevant places.
```

```
   Should '__insertion_sort' be (manually) inlined? */

#pragma implementation

// #undef NDEBUG
#define NDEBUG

#include "Debug.h"
#include "Global.h"
#include "CVISP.h"
#include "Range_redex.h"  // KR
#include "Sparse_table.h" // STL_impl_creator and Own_impl_creator
#include <algorithm>      // STL sorting functions
#include <typeinfo>       // typeid, type_info
#include <iterator>       // iterator_traits
#include <stack>
#include <vector>

///////////////////////////////////////////////////////////////////////
// IMPLEMENTATION
///////////////////////////////////////////////////////////////////////

_USING_NMSPC(CVISP);
_USING_NMSPC(Global);

/*-----------------------------------------------------------------*\
  __insertion_sort
\*-----------------------------------------------------------------*/

// Call '__insertion_sort' from STL although this is explicitly
// discouraged in the 'stl_algo.h' header. Cannot in general use
// '__unguarded_insertion_sort' because it does not check whether an
// element "falls off" the beginning of the segment to be sorted
// during insertion. '__unguarded_insertion_sort' is used in
// '__final_insertion_sort', which in turn is used in 'sort'. In this
// context '__unguarded_insertion_sort' is safe, because a subsegment
// longer than '__stl_threshold' has been partitioned by
// '__unguarded_partition'. Cf., 'stl_algo.h'.
//
// '__insertion_sort' is defined like this:
// template <class RandomAccessIterator>
// void __insertion_sort(RandomAccessIterator first, RandomAccessIterator last)

/*-----------------------------------------------------------------*\
  sort
\*-----------------------------------------------------------------*/

// Call STL 'sort', which is an implementation of Introsort, cf.,
// D. R. Musser: Introspective Sorting and Selection Algorithms,
// Software - Practice and Experience, vol. 27(8), pp. 983-993, August
// 1997.
//
// 'sort' is defined as
// template <class RandomAccessIterator>
// inline void sort(RandomAccessIterator first, RandomAccessIterator last)
//
// Is guaranteed to run in O(n*log n) time worst-case! The constant
// factors are those of a good Quicksort implementation. Not
// guaranteed to be stable.
```

```
/*-----------------------------------------------------------------*\
  contender
\*-----------------------------------------------------------------*/

// Adaptive Radixsort with bound on the allowed radix size, switch to
// Insertion Sort for small subproblems, and in-place permutation of
// keys. Sorts into ascending (better: non-descending) order. For the
// reverse order, change direction of traversals of 'radix_table'.

_NMSPC_BEGIN(CVISP)

// Auxiliary data structures. Alternatively gather with 'contender' in
// a class. Use 'pair' from '<utility>' instead?

template<class Ran>
struct Range {

    Range(Ran s, Ran f): start(s), finish(f) {}

    Ran start, finish;
};

template<class Ran>
struct Group {
    idx_t count;
    Ran pos;
};

_NMSPC_END(CVISP)

template<class Ran>
void _QUAL(CVISP) contender(Ran first, Ran last,
                            width_t width = w_copy, width_t base_shift = 0,
                            width_t max_radix = 9,
                            idx_t threshold = __stl_threshold)
{
    VERIFY(typeid(iterator_traits<Ran>::iterator_category) ==
           typeid(random_access_iterator_tag),
           "contender: 'Ran' not random access iterator");
    VERIFY(typeid(iterator_traits<Ran>::value_type) ==
           typeid(word),
           "contender: '*Ran' not 'word'");
    VERIFY(width + base_shift <= w,
           "contender: 'width + base_shift > w'");
    VERIFY(threshold >= 3, "contender: 'threshold' too small");
    // Recursion with an STL sorting function must happen on a problem
    // size of at least two.

    if (first == last) return;

    Cmp_bit_pattern cbp(make_mask(width,base_shift));

    if (last - first < threshold) {
        if (width == w_copy) __insertion_sort(first,last); // Uses '<'.
        else __insertion_sort(first,last,cbp); // 'width < w'.
        // If 'width == w_copy' the comparison sort can safely compare
        // full words regardless of the value of 'base_shift', since
        // the MSBs are more important. If keys are sorted on 'w' bits,
```

```
            // they are also sorted on the 'w-base_shift' most significant
            // ones.
            return;
        }

        stack<Range<Ran> > range_stack; // Explicit recursion stack. Range part
        stack<width_t> bpos_stack;      // and bit position part. MSB is no. 'w'.

        typedef vector<Group<Ran> > rdx_tbl_t;
        rdx_tbl_t radix_table(1U << max_radix);
        rdx_tbl_t::iterator radix_table_first = radix_table.begin();
        // Initialised with 'count' reset.

        // Main loop. The lines marked with (I) ensure that the sorting
        // inspects at most the bits specified by 'width' and
        // 'base_shift'. Maintains the invariant that a subproblem popped
        // from the stack is larger than 'threshold'.
        range_stack.push(Range<Ran>(first,last));
        bpos_stack.push(width); // (I)
        while ( !range_stack.empty() ) { // && !bpos_stack.empty();
            // Recurse with Adaptive Radixsort.
            Ran r_iter, pos;
            rdx_tbl_t::iterator t_iter;
            rdx_tbl_t::size_type idx;
            iterator_traits<Ran>::difference_type range_sz,recursive_range_sz;

            Range<Ran>& range = range_stack.top();
            range_sz = range.finish - range.start;
            VERIFY(range_sz > 0, "contender: empty subproblem");

            width_t bpos = bpos_stack.top(); // Bit position within pattern.
            VERIFY(bpos > 0, "contender: inspecting too many bits"); // (I)
            int l;
            l = log2_floor(range_sz)+1;
            width_t radix = max_radix < l ? max_radix : l;
            if ( bpos < radix ) radix = bpos; // (I)
            VERIFY(radix, "contender: 'radix' is zero");
            rdx_tbl_t::size_type table_sz = (1U << radix);
            word mask = table_sz - 1U;
            width_t recursive_bpos = bpos - radix;
            width_t shift_dist = recursive_bpos + base_shift; // (I)

            // Tally group sizes. Record span min.
            rdx_tbl_t::size_type span_low_idx = table_sz;
            for ( r_iter = range.start; r_iter != range.finish; ++r_iter ) {
                idx = *r_iter >> shift_dist & mask;
                if ( idx < span_low_idx ) span_low_idx = idx;
                ++radix_table[idx].count;
            }

            // Set 'pos' in each relevant 'Group<Ran>' to point to the
            // start of it. Traverse only span, and record the end of
            // it.
            t_iter = radix_table_first + span_low_idx;
            for ( pos = range.start; pos != range.finish; ++t_iter ) {
                t_iter->pos = pos;
                pos += t_iter->count;
            }
            rdx_tbl_t::iterator span_high_it = t_iter; // 'const_iterator'?
```

```

        // Permute in-place. For convenience the outer loop traverses
        // the radix table span. This is OK, because the loop resets
        // the counts as well, which requires a traversal in any
        // event.
        //
        // The recursive subproblems are either solved directly with
        // an STL sorting function or pushed onto the explicit stack(s).
        pos = range.start;
        range_stack.pop();
        bpos_stack.pop();
        for ( t_iter = radix_table_first + span_low_idx;
              t_iter != span_high_it; ++t_iter ) {

            if ( t_iter->count > 0 ) {
                Ran sentinel = t_iter->pos + t_iter->count;
                word displaced = *(sentinel - 1);
                rdx_tbl_t::iterator entry;
                do {
                    idx = displaced >> shift_dist & mask;
                    entry = radix_table_first + idx;
                    swap<word>(*entry->pos,displaced);
                    ++entry->pos;
                    --entry->count;
                }
                while ( entry->pos != sentinel );
                VERIFY(!_QUAL(Debug) cvisp_check || t_iter->count == 0,
                       "contender: logical error during in-place permutation");
            }
            // 't_iter->pos' now marks the end of the corresponding group.

            if (recursive_bpos && t_iter->pos != pos ) {
                // Counts are now reset, so necessary to use
                // 't_iter->pos != pos' to test if group is not empty.
                range_sz = t_iter->pos - pos;
                if ( range_sz < threshold ) {
                    // Switch to STL sorting function.
                    if (width == w_copy) __insertion_sort(pos,t_iter->pos);
                    else __insertion_sort(pos,t_iter->pos,cbp);
                }
                else {
                    range_stack.push(Range<Ran>(pos,t_iter->pos));
                    bpos_stack.push(recursive_bpos); // (I)
                }
                pos = t_iter->pos;
            }
        }
        // The 'pos' of a 'Group<Ran>' corresponding to a non-empty
        // group now marks the end of the group, i.e., it points
        // one element beyond the last element of the group.
    }

} /* contender */

/*-----------------------------------------------------------------------*\
  sort_bit_pattern
\*-----------------------------------------------------------------------*/

// Not used anywhere. Was used for debugging.
```

```
template<class Ran>
void sort_bit_pattern(Ran first, Ran last,
                      width_t width = w_copy, width_t base_shift = 0)
{
    if (width == w_copy) sort(first,last); // Uses '<'.
    else sort(first,last,Cmp_bit_pattern(make_mask(width,base_shift)));
}

/*-----------------------------------------------------------------*\
  KR_sort
\*-----------------------------------------------------------------*/

// Uses exponential range reduction.

template<class Ran>
void _QUAL(CVISP) KR_sort(Ran first, Ran last, uchar RR_steps = 1)
{
    _USING_NMSPC(Range_redex);
    _USING_NMSPC(Sparse_table);

    if (first==last) return;

//     // Note that 'Own_impl_creator' means 'Hash_table' with
//     // 'Hash_func' (both from 'Sparse_table').
//     KR<Ran,Own_impl_creator<unsigned int, _QUAL(KR_impl) Grp_info> >
//      exp_rr(&KR_sort_helper);
//     exp_rr.sort(first,last,RR_steps);
//     /* Uses 'Own_impl_creator<unsigned int,Grp_info>' for the second
//        template argument of class 'KR'. Cf., the definition in
//        'Range_redex.h'. */

    // Note that 'STL_impl_creator' means 'hash_map' with 'hash' (both
    // from STL). To get 'hash_map' with 'Sparse_table::Hash_func'
    // alter commenting in definition of 'STL_table' and
    // 'STL_impl_creator' in 'Sparse_table.h'.
    KR<Ran,STL_impl_creator<unsigned int, _QUAL(KR_impl) Grp_info> >
        exp_rr(&KR_sort_helper);
    exp_rr.sort(first,last,RR_steps);
    /* Uses 'STL_impl_creator<unsigned int,Grp_info>' for the second
       template argument of class 'KR'. Cf., the definition in
       'Range_redex.h'. */
}

///////////////////////////////////////////////////////////////////////
// EXPLICIT INSTANTIATION
///////////////////////////////////////////////////////////////////////

_NMSPC_BEGIN(CVISP)

template void contender(word_iterator, word_iterator,
                        width_t, width_t,
                        width_t, idx_t);
// Needed by 'driver.cpp'.

template void KR_sort(word_iterator, word_iterator, uchar);
// Needed by 'driver.cpp'.

_NMSPC_END(CVISP)
```

```
// Local Variables:
// compile-command: "gmake /usr/tmp/CVISP.o"
// compilation-scroll-output: t
// End:
```

B.1.11 driver.cpp

```
// Filename: driver.cpp
// Author: Jyrki Katajainen and Morten Nicolaj Pedersen

// #undef NDEBUG
#define NDEBUG

#define __ISAAC64
#include "isaac.hpp"  // ISAAC PRNG

#include "Debug.h"
#include "Global.h"
#include "CVISP.h"    // contender
#include <cstddef>    // size_t
#include <cstdlib>    // random and srandom
#include <ctime>      // clock_t and clock
#include <iostream>   // standards streams
#include <functional> // less
#include <algorithm>  // swap and sort
#include <iterator>   // iterator_traits

///////////////////////////////////////////////////////////////////////
// DECLARATIONS
///////////////////////////////////////////////////////////////////////

// The type of a pointer to a sorting function. Cannot directly
// templatize a 'typedef'.
template<class random_access_iterator>
struct SF {
    typedef void (*sorting_function)(random_access_iterator first,
                                     random_access_iterator last);
};

// Pseudo Random Number Generator.
QTIsaac<8> prng;

///////////////////////////////////////////////////////////////////////
// IMPLEMENTATION
///////////////////////////////////////////////////////////////////////

_USING_NMSPC(Global);

/*-----------------------------------------------------------------*\
  dummy
\*-----------------------------------------------------------------*/

template<class random_access_iterator>
void dummy(random_access_iterator first, random_access_iterator last) {}

/*-----------------------------------------------------------------*\
  show
```

```
\*-----------------------------------------------------------------*/

// Show range on 'cout'.
template <class random_access_iterator>
void show(random_access_iterator first, random_access_iterator last) {

    random_access_iterator p;
    cout << "{";
    for (p = first; p < last - 1; ++p)
        cout <<  *p << ", ";
    // 'p' now points to 'last - 1'.
    cout << *p << "}" << endl;

} // show

/*-----------------------------------------------------------------*\
  permute
\*-----------------------------------------------------------------*/

// Permute. Does it generate each permutation with probability 1/fac(n)?
template <class random_access_iterator>
void permute(random_access_iterator first, random_access_iterator last) {
    // Does not call 'srandom'! If it did, the same permutation of
    // {1,...,n} would be constructed each time.

    random_access_iterator p,q;
    for (q = last - 1; q > first; --q) {
        p = first + (random() % (q - first + 1));
        swap(*p, *q);
    }

} // permute

/*-----------------------------------------------------------------*\
  generate
\*-----------------------------------------------------------------*/

// Generate sequence.
template <class random_access_iterator>
void generate(random_access_iterator first, random_access_iterator last,
              char sequence_type) {

    typedef iterator_traits<random_access_iterator>::value_type value_type;
    iterator_traits<random_access_iterator>::difference_type diff,half,length;
    random_access_iterator q;
    unsigned int i,val;

    switch (sequence_type) {
    case 'r':  // an ISAAC random sequence (default)
        for (q = first; q != last; ++q) {
            *q = value_type(prng.rand());
        }
        break;
    case 'b':  // a random binary sequence
        // 0 1 1 0 1 1 1 1 0 0 1 0 0, ...
        for (q = first; q != last; ++q)
            *q = value_type(random() & 1U);
        break;
    case 'd':  // a decreasing sequence
```

```
        // n-1, n-2, ... , 0
        for (q = first; q != last; ++q)
            *q = value_type(last - q - 1);
        break;
    case 'i':  // an increasing sequence
        // 0, 1, ... , n-1
        for (q = first; q != last; ++q)
            *q = value_type(q - first);
        break;
    case 'p':  // a random permutation of 0, 1, ... , n-1
        for (q = first; q != last; ++q)
            *q = value_type(q - first);
        permute(first, last);
        break;
    case 'z':  // a sequence of zeros
        // 0, 0, ... , 0
        for (q = first; q != last; ++q) {
            *q = value_type(0);
        }
        break;
    case 's': // skewed
        // 0
        // 1
        // 2 2
        // 3 3 3 3
        // 4 4 4 4 4 4 4 4
        // ...
        // log_2(n) (n/2 times)
        diff = last - first;
        if ( diff & 1U ) {
            cerr << "generate: length not even, cannot generate skewed"
                 << endl;
            exit(5);
        }
        *first = value_type(0);
        q = first + 1;
        length = 1;
        val = 1;
        while ( length < diff ) {
            for ( i = 0; i < length; ++i, ++q )
                *q = value_type(val);
            length <<= 1;
            val++;
        }
        break;
    case 'm': // median-of-three-killer
        diff = last - first;
        if ( diff & 1U ) {
            cerr << "generate: length not even, cannot generate " << endl
                 << "          median-of-three-killer" << endl;
            exit(5);
        }
        half = diff >> 1;
        for ( q = first, i = 0; i < half; ++q, ++i )
            if ( i & 1U ) *q = value_type(half - 1 + i);
            else *q = value_type(i);
        // Now: 'i == half' and 'q == first + half'.
        for ( val = 1; i < diff; ++q, ++i, val += 2 )
            *q = value_type(val);
```

```
        break;
    }

} // generate

/*----------------------------------------------------------------------*\
  usage
\*----------------------------------------------------------------------*/

// Print usage message on 'cout'.
void usage(int argc, char **argv) {

    cerr << "Usage: " << argv[0]
         << " [<log_2(size)>] [<sequence type>] [<seed>]" << endl
         << "    where <sequence type> is one of the following:" << endl
         << "        'r' an ISAAC random sequence (default)" << endl
         << "        'b' a random binary sequence" << endl
         << "        'd' a decreasing sequence" << endl
         << "        'i' an increasing sequence" << endl
         << "        'p' a random permutation" << endl
         << "        'z' a sequence of zeros" << endl
         << "        's' skewed" << endl
         << "        'm' median-of-three-killer" << endl
         << "    and <seed> is the seed for 'random' from the" << endl
         << "    C Standard Library." << endl;

} // usage

/*----------------------------------------------------------------------*\
  invocate
\*----------------------------------------------------------------------*/

// Invocate the sorting function specified by the 'sf' function pointer.
void invocate(const char* name,
              SF<word_iterator>::sorting_function sf,
              word_container& keys,
              size_t n, size_t repetitions,
              char sequence_type, unsigned int seed) {

    clock_t c_start, c_stop, c_time;
    word_iterator first,last;

    // Reset PRNGs.
    srandom(seed);
    prng.srand();

    // Generate instance(s).
    first = keys.begin();
    last = first + n;
    for (size_t i = 0; i < repetitions; i++) {
        generate(first, last, sequence_type);
//      show(first,last);
        first = last;
        last += n;
    }

    // Cache now in "reset" state. Cf., (B) below.

    // Sort.
```

```
    first = keys.begin();
    last = first + n;
    c_start = clock();
    for (size_t i = 0; i < repetitions; i++) {
        sf(first, last);
        first = last;
        last += n;
    }
    c_stop = clock();

    // Print result.
    c_time = c_stop - c_start;
    cout << name << ": Average sorting time (s): "
         << double(c_time) / (double(CLOCKS_PER_SEC) * double(repetitions))
         << endl;

    // Verify sorting.
    first = keys.begin();
    last = first + n;
    for (int i = 0; i < repetitions; i++) {
        if (!is_sorted(first, last) && sf != &dummy<word_iterator>)
            cerr << "*** sorting error ***" << endl;
        first = last;
        last += n;
    }

    cout << endl;
} // invocate

/*-----------------------------------------------------------------*\
  main
\*-----------------------------------------------------------------*/

// Command line arguments: 0 - name of object file, 1 - positive
// integer specifing the base two logarithm of the number of keys to
// be sorted, 2 - character determining the sequence type, cf.,
// 'usage', 3 - seed for 'srandom'.

// Perform experiments.
int main(int argc, char **argv) {

    const size_t max_n = 1U << 23;  // 8M. Cf. (A) and (C) below.
    size_t n = 1U << 20;            // 1M, or 1048576. Default.
    size_t repetitions = max_n / n; // Default.
    unsigned int seed = 1;          // For `random' from C Std. Lib. Default.
    char sequence_type = 'r';       // Default.
    word_container keys(max_n);

    // Label (B): 'sizeof(word)*max_n' must be larger than the size
    // of the board-level cache, and 'n' must divide 'max_n' with
    // remainder zero. Under these conditions 'generate' addresses
    // each cache block at least once. This ensures that the
    // generation of instances in 'invocate' before the sorting is
    // performed, leaves the board-level cache in the same state, each
    // time 'invocate' is called.
    if ( sizeof(word)*max_n < 8*(1U << 20) ||
         max_n % n != 0 ) {

        cerr << "main: 'max_n' too small or 'max_n % n != 0'" << endl;
```

```
        exit(5);
        // The Alpha ('ng2h del1-alpha') cache is 8Mb.
    }

    // Parse command line.
    if (argc < 1 || argc > 4) {
        usage(argc, argv);
        exit(1);
    }
    if (argc > 1) {
        n = 1 << atoi(argv[1]);
        if (n < 2 || n > max_n) {
            cerr << "main: size out of range {2,...," << max_n << "}" << endl;
            usage(argc, argv);
            exit(2);
        }
        repetitions = max_n / n;
        // Label (A): As long as 'max_n' is a power of two as well, it
        // is guaranteed that 'max_n % n == 0'.
    }
    if (argc > 2) {
        sequence_type = argv[2][0];
        switch (sequence_type) {
        case 'r': case 'b': case 'd':
        case 'i': case 'p': case 'z':
        case 's': case 'm':
            ;
            break;
        default:
            usage(argc, argv);
            exit(3);
            break;
        }
    }
    if (argc > 3) {
        int buffer = atoi(argv[3]);
        if (buffer <= 0) {
            cerr << "main: seed should be positive" << endl;
            usage(argc, argv);
            exit(4);
        }
        else seed = buffer;
    }

    // Execute experiments/test runs.
    time_t now = time(NULL);
    cout << endl << asctime(localtime(&now));
    cout << "n: " << n << endl;
    cout << "repetitions: " << repetitions << endl;
    cout << "sequence type: ";
    switch (sequence_type) {
    case 'r':
        cout << "an ISAAC random sequence";
        break;
    case 'b':
        cout << "a random binary sequence";
        break;
    case 'd':
        cout << "a decreasing sequence";
```

```
            break;
        case 'i':
            cout << "an increasing sequence";
            break;
        case 'p':
            cout << "a random permutation";
            break;
        case 'z':
            cout << "a sequence of zeros";
            break;
        case 's':
            cout << "skewed";
            break;
        case 'm':
            cout << "median-of-three-killer";
            break;
        }
        cout << endl;
        cout << "seed (for 'srandom'): " << seed << endl;
        cout << endl;

        try {
//          invocate("Dummy",&dummy,
//                   keys,n,repetitions,sequence_type,seed);
//          invocate("Contender",&_QUAL(CVISP) contender_wrapper,
//                   keys,n,repetitions,sequence_type,seed);
//          invocate("KR_sort",&_QUAL(CVISP) KR_sort_wrapper,
//                   keys,n,repetitions,sequence_type,seed);
            invocate("STL Introsort ('std::sort')",&sort,
                     keys,n,repetitions,sequence_type,seed);
        }
        catch (_QUAL(Debug) Failure& f) {
            _QUAL(std) cerr << f.cause();
            exit(-1);
        }

        return 0;

    } // main

    // Label (C): Brugerprocesser kan maksimalt fylde 128 MB data og 2 MB
    // stak. Undtagelse vil kræve omkonfigurering og installation af nye
    // kerner, hvilket vi ikke har resourcer til.

    //  Local Variables:
    //  compile-command: "gmake driver"
    //  compilation-scroll-output: t
    //  End:
```

B.1.12 isaac.hpp

```
#ifndef __ISAAC_HPP
#define __ISAAC_HPP

/*

    C++ TEMPLATE VERSION OF Robert J. Jenkins Jr.'s
    ISAAC Random Number Generator.
```

```
    Ported from vanilla C to to template C++ class
    by Quinn Tyler Jackson on 16-23 July 1998.

        qjackson@wave.home.com

    The function for the expected period of this
    random number generator, according to Jenkins is:

        f(a,b) = 2**((a+b*(3+2^^a)-1)

        (where a is ALPHA and b is bitwidth)

    So, for a bitwidth of 32 and an ALPHA of 8,
    the expected period of ISAAC is:

        2^^(8+32*(3+2^^8)-1) = 2^^8295

    Jackson has been able to run implementations
    with an ALPHA as high as 16, or

        2^^2097263

*/

#include <cstdlib>

#ifndef __ISAAC64
typedef unsigned int UINT32;
const UINT32 GOLDEN_RATIO = UINT32(0x9e3779b9);
typedef UINT32 ISAAC_INT;
#else   // __ISAAC64
// typedef unsigned __int64 UINT64;
// const UINT64 GOLDEN_RATIO = UINT64(0x9e3779b97f4a7c13);
// typedef UINT64 ISAAC_INT;
typedef unsigned long int UINT64;
const UINT64 GOLDEN_RATIO = UINT64(0x9e3779b97f4a7c13);
typedef UINT64 ISAAC_INT;
#endif  // __ISAAC64


template <int ALPHA = (8), class T = ISAAC_INT>
   class QTIsaac
   {
   public:

      typedef unsigned char byte;

      struct randctx
      {
         randctx(void)
         {
            randrsl = static_cast<T*>(std::malloc(N * sizeof(T)));
            randmem = static_cast<T*>(std::malloc(N * sizeof(T)));
         }

         ~randctx(void)
         {
            std::free(static_cast<void*>(randrsl));
```

```
            std::free(static_cast<void*>(randmem));
         }

         T randcnt;
         T* randrsl;
         T* randmem;
         T randa;
         T randb;
         T randc;
      };

      QTIsaac(T a = 0, T b = 0, T c = 0);
      virtual ~QTIsaac(void);

      T rand(void);
      virtual void randinit(randctx* ctx, bool bUseSeed);
      virtual void srand(T a = 0, T b = 0, T c = 0, T* s = NULL);

      enum {N = (1<<ALPHA)};

   protected:

      virtual void isaac(randctx* ctx);

      T ind(T* mm, T x);
      void rngstep(T mix, T& a, T& b, T*& mm, T*& m, T*& m2,T*& r, T& x, T& y);
      virtual void shuffle(T& a, T& b, T& c, T& d, T& e, T& f, T& g, T& h);

   private:

      randctx m_rc;
   };


template<int ALPHA, class T>
   QTIsaac<ALPHA,T>::QTIsaac(T a, T b, T c)
   {
      srand(a, b, c);
   }


template<int ALPHA, class T>
   QTIsaac<ALPHA,T>::~QTIsaac(void)
   {
    // DO NOTHING
   }


template<int ALPHA, class T>
   void QTIsaac<ALPHA,T>::srand(T a, T b, T c, T* s)
   {
      for(int i = 0; i < N; i++)
      {
         m_rc.randrsl[i] = s != NULL ? s[i] : 0;
      }

      m_rc.randa = a;
      m_rc.randb = b;
      m_rc.randc = c;
```

```
      randinit(&m_rc, true);
   }


template<int ALPHA, class T>
   inline T QTIsaac<ALPHA,T>::rand(void)
   {
      return(!m_rc.randcnt-- ? (isaac(&m_rc), m_rc.randcnt=(N-1),
                                m_rc.randrsl[m_rc.randcnt]) :
             m_rc.randrsl[m_rc.randcnt]);
   }


template<int ALPHA, class T>
   void QTIsaac<ALPHA,T>::randinit(randctx* ctx, bool bUseSeed)
   {
      int i = 0;

      T a,b,c,d,e,f,g,h;

      a = b = c = d = e = f = g = h = GOLDEN_RATIO;

      T* m = (ctx->randmem);
      T* r = (ctx->randrsl);

      if(!bUseSeed)
      {
         ctx->randa = 0;
         ctx->randb = 0;
         ctx->randc = 0;
      }

    // scramble it
      for(i=0; i < 4; ++i)
      {
         shuffle(a,b,c,d,e,f,g,h);
      }

      if(bUseSeed)
      {
        // initialize using the contents of r[] as the seed

         for(i=0; i < N; i+=8)
         {
            a+=r[i  ]; b+=r[i+1]; c+=r[i+2]; d+=r[i+3];
            e+=r[i+4]; f+=r[i+5]; g+=r[i+6]; h+=r[i+7];

            shuffle(a,b,c,d,e,f,g,h);

            m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
            m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
         }

        //do a second pass to make all of the seed affect all of m

         for(i=0; i < N; i += 8)
         {
            a+=m[i  ]; b+=m[i+1]; c+=m[i+2]; d+=m[i+3];
```

```
            e+=m[i+4]; f+=m[i+5]; g+=m[i+6]; h+=m[i+7];

            shuffle(a,b,c,d,e,f,g,h);

            m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
            m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
         }
      }
      else
      {
        // fill in mm[] with messy stuff

         shuffle(a,b,c,d,e,f,g,h);

         m[i  ]=a; m[i+1]=b; m[i+2]=c; m[i+3]=d;
         m[i+4]=e; m[i+5]=f; m[i+6]=g; m[i+7]=h;
      }

      isaac(ctx);         // fill in the first set of results
      ctx->randcnt = N;   // prepare to use the first set of results
   }


template<int ALPHA, class T>
   inline T QTIsaac<ALPHA,T>::ind(T* mm, T x)
   {
    #ifndef __ISAAC64
      return (*(T*)((byte*)(mm) + ((x) & ((N-1)<<2))));
    #else   // __ISAAC64
    return (*(T*)((byte*)(mm) + ((x) & ((N-1)<<3))));
    #endif  // __ISAAC64
   }


template<int ALPHA, class T>
   inline void QTIsaac<ALPHA,T>::rngstep(T mix, T& a, T& b,
                                         T*& mm, T*& m, T*& m2,
                                         T*& r, T& x, T& y)
   {
      x = *m;
      a = (a^(mix)) + *(m2++);
      *(m++) = y = ind(mm,x) + a + b;
      *(r++) = b = ind(mm,y>>ALPHA) + x;
   }


template<int ALPHA, class T>
   void QTIsaac<ALPHA,T>::shuffle(T& a, T& b, T& c, T& d,
                                  T& e, T& f, T& g, T& h)
   {
    #ifndef __ISAAC64
      a^=b<<11; d+=a; b+=c;
      b^=c>>2;  e+=b; c+=d;
      c^=d<<8;  f+=c; d+=e;
      d^=e>>16; g+=d; e+=f;
      e^=f<<10; h+=e; f+=g;
      f^=g>>4;  a+=f; g+=h;
      g^=h<<8;  b+=g; h+=a;
      h^=a>>9;  c+=h; a+=b;
```

```
    #else // __ISAAC64
    a-=e; f^=h>>9;  h+=a;
    b-=f; g^=a<<9;  a+=b;
    c-=g; h^=b>>23; b+=c;
    d-=h; a^=c<<15; c+=d;
    e-=a; b^=d>>14; d+=e;
    f-=b; c^=e<<20; e+=f;
    g-=c; d^=f>>17; f+=g;
    h-=d; e^=g<<14; g+=h;
    #endif // __ISAAC64
   }


template<int ALPHA, class T>
   void QTIsaac<ALPHA,T>::isaac(randctx* ctx)
   {
      T x,y;

      T* mm = ctx->randmem;
      T* r  = ctx->randrsl;

      T a = (ctx->randa);
      T b = (ctx->randb + (++ctx->randc));

      T* m    = mm;
      T* m2   = (m+(N/2));
      T* mend = m2;

      for(; m<mend; )
      {
        #ifndef __ISAAC64
         rngstep((a<<13), a, b, mm, m, m2, r, x, y);
         rngstep((a>>6) , a, b, mm, m, m2, r, x, y);
         rngstep((a<<2) , a, b, mm, m, m2, r, x, y);
         rngstep((a>>16), a, b, mm, m, m2, r, x, y);
        #else   // __ISAAC64
        rngstep(~(a^(a<<21)), a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a>>5)  , a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a<<12) , a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a>>33) , a, b, mm, m, m2, r, x, y);
        #endif  // __ISAAC64
      }

      m2 = mm;

      for(; m2<mend; )
      {
        #ifndef __ISAAC64
         rngstep((a<<13), a, b, mm, m, m2, r, x, y);
         rngstep((a>>6) , a, b, mm, m, m2, r, x, y);
         rngstep((a<<2) , a, b, mm, m, m2, r, x, y);
         rngstep((a>>16), a, b, mm, m, m2, r, x, y);
        #else   // __ISAAC64
        rngstep(~(a^(a<<21)), a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a>>5)  , a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a<<12) , a, b, mm, m, m2, r, x, y);
        rngstep(  a^(a>>33) , a, b, mm, m, m2, r, x, y);
        #endif  // __ISAAC64
      }
```

```
        ctx->randb = b;
        ctx->randa = a;
    }

#endif // __ISAAC_HPP
```

B.2 Next-Gen

This section lists the instrumentation functions (Section B.2.1, modification of `cache` instrumentation functions) and the analysis functions (Section B.2.2, modification of Spork's CPROF cache simulator) which defines NEXT-GEN, my customisable, on-the-fly cache simulator, cf., Section 7.1.2. The "DM" suffix used below is an abbreviation of "direct mapped". I also briefly describe how to use NEXT-GEN.

ATOM can instrument programs regardless of whether they are linked with libraries in a non-shared or call-shared fashion. See the manual page. I describe the first alternative. The first step in using NEXT-GEN.DM is to compile your program using your favourite compiler and the desired optimization level, e.g., `g++ -O3 -static -Xlinker -non_shared` *`source_files`* `-o` *`object_file`*. Next change to the object code directory and instrument `object_file` by giving the command: `atom` *`object_file`* `Next-Gen.inst.c Next-Gen.DM.anal.c -all -A1 [-toolargs=''`*`switches`*`''] -o` *`inst_object_file`*. As a side-effect this creates a file called *`object_file`*`.Next-Gen.config`, which is used by `inst_object_file`. Finally, run the instrumented program with *`inst_object_file args`*. This performs the task, which the program was written for, simultaneously with the on-the-fly cache simulation. Per default an 8Mb direct-mapped cache with 64-byte blocks and a capacity of 2^{20} 64-bit words is simulated, and the result is written to a file called `Next-Gen.DM.out`. Use `-h` for *`switches`* and run the instrumented program for a synopsis of how to change the default configuration.

B.2.1 Next-Gen.inst.c

```
/* Filename: Next_Gen.inst.c
   Author: Morten Nicolaj Pedersen
   Remarks: Instrumentation file for use with ATOM, cf.,

   @InProceedings{SrivastavaE1994,
   title={{ATOM}---A System for Building Customized Program Analysis
   Tools},
   author={Amitabh Srivastava and Alan Eustace},
   crossref={pldi94},
   pages={196--205},
   source={http://theory.lcs.mit.edu/~dmjones/hbp/pldi/pldi.bib}
   }

   @Proceedings{PLDI94,
   title={Proceedings of the ACM SIGPLAN'94 Conference on Programming
   Language Design and Implementation (PLDI)},
   booktitle={Proceedings of the ACM SIGPLAN'94 Conference on Programming
   Language Design and Implementation (PLDI)},
```

```
   address={Orlando, Florida},
   month={20--24~} # jun,
   year=1994,
   note={\emph{SIGPLAN Notices} 29(6), June 1994},
   crossrefonly=1,
   source={http://theory.lcs.mit.edu/~dmjones/hbp/pldi/pldi.bib}
   }@string{iandcomp={Information and Computation}}
*/

#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <cmplrs/atom.inst.h>

/*******************************************************************\
  functions
\*******************************************************************/

/*-----------------------------------------------------------------*\
  write_sim_config
\*-----------------------------------------------------------------*/

/* Writes the simulator configuration (set with the '-toolargs' switch
   of ATOM), i.e., the requested cache metrics, to a file
   '<object_file>.Next-Gen.config'. */

int write_sim_config(const char* config_file_name, int argc, char **argv)
{
    int i,len,strs_len;
    FILE* config_file;

    config_file = fopen(config_file_name,"w");
    assert(config_file != NULL);
    fprintf(config_file,"%u\n",argc);
    strs_len = 0;
    for ( i = 0; i < argc; i++ ) {
        len = strlen(argv[i]) + 1; /* 1 <--> '\0'. */
        strs_len += len;
        fprintf(config_file,"%s %u\n",argv[i],len);
    }
    fclose(config_file);
    return strs_len;
}

/*-----------------------------------------------------------------*\
  InstrumentAll
\*-----------------------------------------------------------------*/

/* The instrumentation routine of Next-Gen. See Chapter 9 of the
   Digital Unix Programmer's Guide. */

unsigned InstrumentAll(int argc, char **argv)
{
    int strs_len;
    char* config_file_name;
    Obj *o; Proc *p;  Block *b;  Inst *i;


    config_file_name = (char*) malloc((16+strlen(argv[0])+1)*sizeof(char));
```

```
    sprintf(config_file_name,"%s.Next-Gen.config",argv[0]);
    /* 16 is length of static part of name. 1 corresponds to '\0'. */
    strs_len = write_sim_config(config_file_name,argc,argv);

    AddCallProto("read_sim_config(const char*,int)");
    AddCallProto("report_res()");
    AddCallProto("reference(VALUE,char)");
    for (o = GetFirstObj(); o != NULL; o = GetNextObj(o)) {
        if (BuildObj(o)) return 1;
        for (p=GetFirstProc(); p != NULL; p = GetNextProc(p)) {
            for (b = GetFirstBlock(p); b != NULL;  b = GetNextBlock(b)) {
                for (i = GetFirstInst(b); i != NULL; i = GetNextInst(i)) {
                        if (IsInstType(i,InstTypeLoad))
                            AddCallInst(i,InstBefore,"reference",
                                        EffAddrValue,'r');
                        else if (IsInstType(i,InstTypeStore))
                            AddCallInst(i,InstBefore,"reference",
                                        EffAddrValue,'w');
                }
            }
        }
        WriteObj(o);
    }
    AddCallProgram(ProgramBefore,"read_sim_config",
        config_file_name,strs_len);
    AddCallProgram(ProgramAfter,"report_res");

    return 0;

} /* InstrumentAll */
```

B.2.2 Next-Gen.DM.anal.c

```
/* Filename: Next-Gen.DM.anal.c
   Author: Morten Nicolaj Pedersen
   Remarks: Analysis file for use with ATOM. */

#include <stdio.h>
#include <assert.h>
#include <getopt.h>
#include <stdlib.h>
#include <string.h>

/*********************************************************************\
  typedefs and global variables
\*********************************************************************/

struct Block {
    unsigned long tag;
/*     unsigned valid : 1; */
/*     unsigned dirty : 1; */
    unsigned char valid, dirty;
};
/* Cache-block type. Is it better to use 'unsigned char' instead of
   the bit field capability? */

struct Block* cache;
/* The global cache variable. malloc'ed in 'read_sim_config' and
```

```
   free'd in 'report_res'. */

struct Metrics {
    unsigned int a;              /* associativity, must equal 1 */
    unsigned int M;              /* capacity in words */
    unsigned int B;              /* block/line size in words */
    unsigned int bitsperword;    /* word size in bits */
    unsigned int s;              /* sets = M/(aB) */
    unsigned int bytesperword;   /* word size in bytes */
    unsigned int bytesperline;   /* bytes per block/line */
} metrics = { 1, 1048576, 8, 64 };
/* The global cache metrics variable. The default is an 8Mb
   direct-mapped cache with 64-byte blocks. The wordsize is 64
   bits. Models an Alpha 21164 board-level cache. Since 'a==1', the
   replacement policy has no significance. Random is chosen for
   convenience/speed. */

struct Statistics {
    unsigned long compulsory, replacement; /* miss types */
    unsigned long comread, comwrite;
    unsigned long represd, repwrite;
    unsigned long clean, dirty;            /* clean/dirty misses */
    unsigned long missread, misswrite;     /* miss reads/writes */
    unsigned long hitread, hitwrite;       /* hit reads/writes */
    unsigned long totread, totwrite;       /* total reads/writes */
    unsigned long ser;                     /* serial reference number */
} stats = { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 };
/* Miss statistics. Global variable. */

struct Twiddle {
    unsigned long bytespl_mask, s_mask;
    unsigned char bytespw_dist, bytespl_dist, tag_dist;
} twiddle;
/* Shift distances and masks for bitwise logical and. Used in
   'reference' */

const int error_code = 10;
int verbose = 0;
int LaTeX = 0;
char *comment = "", *LaTeX_comment = "% ";
FILE* res_file = 0; /* NULL */
/* Environment. */

/*******************************************************************\
  functions
\*******************************************************************/

/*-----------------------------------------------------------------*\
  log2_floor
\*-----------------------------------------------------------------*/

int log2_floor(unsigned int n)
{
    int i;
    assert(n);
    for ( i = 0; n; n >>= 1, i++)
        ;
    return i - 1;
} /* log2_floor */
```

```
/*-----------------------------------------------------------------*\
  parse_switch
\*-----------------------------------------------------------------*/

void parse_switch(int opt)
{
    int i;

    switch (opt) {
    case 'l':
        LaTeX = 1;
        comment = LaTeX_comment;
        break;
    case 'v':
        verbose = 1;
        break;
    case 'r':
        res_file = fopen(optarg,"w");
        /* 'res_file' asserted below. */
        break;
    case 'm':
        metrics.M = atoi (optarg);
        break;
    case 'b':
        metrics.B = atoi (optarg);
        break;
    case 'w':
        metrics.bitsperword = atoi (optarg);
        break;
    case 'h':
        fprintf(stdout,"Next-Gen.DM, an on-the-fly simulator");
        fprintf(stdout,"of direct-mapped caches.\n");
        fprintf(stdout,"Usage: atom appl_prog Next-Gen.inst.c");
        fprintf(stdout,"Next-Gen.DM.anal.c\n");
        fprintf(stdout,"       -o inst_prog -all -A1 -shlibdir");
        fprintf(stdout,"shared_lib_dir\n");
        fprintf(stdout,"       [-toolargs=\"switches\"]\n");
        fprintf(stdout,"Allowed switches are:\n");
        fprintf(stdout," -mM: set cache capacity to M words");
        fprintf(stdout," ( default %u )\n", metrics.M);
        fprintf(stdout," -bB: set block size to B words");
        fprintf(stdout," ( default %u )\n", metrics.B);
        fprintf(stdout," -wW: set word size to W bits");
        fprintf(stdout," ( default %u )\n", metrics.bitsperword);
        fprintf(stdout," -rR: output result of simulation to file R");
        fprintf(stdout," ( default is stdout )\n");
/*      fprintf(stdout," -l: produce LaTeX output\n"); */
/*      fprintf(stdout," -v: verbose output (all memory references)\n"); */
        fprintf(stdout," -h: this message\n");
        exit(0);
        break;
    case '?':
        fprintf(stderr,"Next_Gen.DM: error: bad arg\n");
        exit (error_code);
        break;
    }
} /* parse_switch */
```

```
/*-------------------------------------------------------------------*\
  read_sim_config
\*-------------------------------------------------------------------*/

/* Reads the simulator configuration (set with the '-toolargs' switch
   of ATOM), i.e., the requested cache metrics, from a file
   '<object_file>.Next-Gen.config'. */

void read_sim_config(const char* config_file_name, int strs_len)
{
    int i,len,opt,argcc;
    unsigned int nof_blocks;
    struct Block* bp;
    FILE* config_file;
    char* args;
    char** argvv;

    /* Read simulator configuration file. */
    args = (char*) malloc(strs_len*sizeof(char));
    config_file = fopen(config_file_name,"r");
    assert(config_file != NULL);
    fscanf(config_file,"%u",&argcc);
    argvv = (char**) malloc(argcc*sizeof(char*));
    for ( i = 0; i < argcc; i++ ) {
        fscanf(config_file,"%s %u",args,&len);
        argvv[i] = args;
        args += len;
    }
    fclose(config_file);

    /* Parse arguments. */
    do {
        opt = getopt (argcc, argvv, "lvhr:m:b:w:");
        parse_switch(opt);
    } while (opt != -1);
    assert(res_file != NULL);

    /* Free memory. */
    free(args);
    free(argvv);

    /* Clean up option arguments and work out metrics */
    if ( (metrics.M - 1) & metrics.M) {
        fprintf(stderr,"Next_gen.DM: error: 'M' not power of two\n");
        exit (error_code);
    }

    if ( metrics.B > metrics.M || (metrics.M % metrics.B) ||
         ((metrics.B - 1) & metrics.B) ) {
        fprintf(stderr,"Next_gen.DM: error: 'B' does not divide 'M',");
        fprintf(stderr,"or 'B' > 'M', or 'B' not power of two\n");
        exit (error_code);
    }

    if ( (metrics.a * metrics.B > metrics.M) ||
         (metrics.M % (metrics.a * metrics.B)) ) {
        fprintf(stderr,"Next_Gen.DM: error: associativity too high,");
        fprintf(stderr,"or not appropriate\n");
        exit (error_code);
```

```
    }

    if ( (metrics.bitsperword % 8) ||
         ((metrics.bitsperword - 1) & metrics.bitsperword) ) {
        fprintf(stderr,"Next_Gen.DM: error: word length mod 8 != 0,");
        fprintf(stderr,"or not power of two\n");
        exit (error_code);
    }

    if (metrics.a != 1) {
        fprintf(stderr,"Next_gen.DM: error: not direct-mapped\n");
        exit (error_code);
    }

    metrics.s = metrics.M / (metrics.a * metrics.B);
    metrics.bytesperword = metrics.bitsperword / 8;
    metrics.bytesperline = metrics.B * metrics.bytesperword;
    /* All left-hand-sides are powers of two as all right-hand-side
       quantities are! */

    /* Define twiddle. */
    twiddle.bytespl_mask = metrics.bytesperline - 1;
    twiddle.s_mask = metrics.s - 1;
    twiddle.bytespw_dist = log2_floor(metrics.bytesperword);
    twiddle.bytespl_dist = log2_floor(metrics.bytesperline);
    twiddle.tag_dist = log2_floor(metrics.bytesperline * metrics.s);

    /* malloc returns pointer to uninitialised space. Must reset
       manually. */
    nof_blocks = (metrics.M / metrics.B);
    cache = (struct Block*) malloc(nof_blocks*sizeof(struct Block));
    for ( bp = cache, i = 0; i < nof_blocks; i++, bp++) {
        bp->tag = 0;
        bp->valid = 0;
        bp->dirty = 0;
    }

} /* read_sim_config */

/*-------------------------------------------------------------------*\
  reference
\*-------------------------------------------------------------------*/

void reference(unsigned long ref, char w)
{
    unsigned long offset, index, tag;

    index = (ref >> twiddle.bytespl_dist) & twiddle.s_mask;
    tag = ref >> twiddle.tag_dist;
    if (tag == cache[index].tag && cache[index].valid) {
        /* Cache hits */
        if (w == 'w') stats.hitwrite++;
        else stats.hitread++;
    }
    else {
        /* Cache misses */
        /* compulsory miss: first access to a block (valid = false)
           replacment miss: all other accesses to a block
           (valid = true) */
```

```
        if (cache[index].valid) {
            stats.replacement++;
            if (w == 'w') stats.repwrite++;
            else stats.represd++;
        } else {
            stats.compulsory++;
            if (w == 'w') stats.comwrite++;
            else stats.comread++;
        }

        if (cache[index].dirty) stats.dirty++;
        else stats.clean++;
        if (w == 'w') stats.misswrite++;
        else stats.missread++;

        cache[index].dirty = 0;
        cache[index].valid = 1;
        cache[index].tag = tag;
    }

    if (w == 'w') {
        cache[index].dirty = 1;
        stats.totwrite++;
    }
    else stats.totread++;

    stats.ser++;

} /* reference */

/*-------------------------------------------------------------------*\
  report_res
\*-------------------------------------------------------------------*/

void report_res()
{
    const int width = 10;

    assert(stats.clean+stats.dirty == stats.missread+stats.misswrite);

    /* Produce a statement of cache metrics. */

    fprintf(res_file,"%-14s %*s %*s %*s\n\n","",
            width,"Total",width,"Reads",width,"Writes");
    fprintf(res_file,"%-14s %*u %*u %*u\n","References",
            width,stats.ser,width,stats.totread,width,stats.totwrite);
    fprintf(res_file,"%-14s %*u %*u %*u\n","Hits",
            width,stats.hitread+stats.hitwrite,
            width,stats.hitread,width,stats.hitwrite);
    fprintf(res_file,"%-14s %*u %*u %*u\n","Misses",
            width,stats.missread+stats.misswrite,
            width,stats.missread,width,stats.misswrite);
    fprintf(res_file,"%-14s %*u %*u %*u\n"," - compulsory",
            width,stats.compulsory,width,stats.comread,
            width,stats.comwrite);
    fprintf(res_file,"%-14s %*u %*u %*u\n"," - replacement",
            width,stats.replacement,width,stats.repread,
            width,stats.repwrite);
    fprintf(res_file,"\n");
```

```
    fprintf(res_file,"%-20s %u/%u\n","Clean/dirty misses:",
            stats.clean,stats.dirty);
    fprintf(res_file,"%-20s %f\n","Hit/miss ratio:",
            (double) (stats.hitread+stats.hitwrite) /
            (double) (stats.missread+stats.misswrite));
    fprintf(res_file,"%-20s %f%%\n","Miss rate:",
            100 - (double) (stats.hitread+stats.hitwrite) /
            (double) (stats.totread+stats.totwrite) * 100.0 );

    free(cache);
    fclose(res_file);

} /* report_res */

/* ATOM uses CC: */

/* Local Variables:
   compile-command: "cc -c Next_Gen.anal.c -o /usr/tmp/Next_Gen.anal.o"
   compilation-scroll-output: t
   End: */
```

C A "minimal standard" random number generator

I next describe a random number generator (RNG) by Park and Miller [PM88], which I have not actually used. I mention it anyway, because it is a milestone in the literature. In the context of this thesis it is not directly relevant, because it does not generate 64-bit values. Nilsson has used the generator for his experiments with Nilsson's Sort, cf., Section 3.6.2, in his Ph.D. thesis[Nil96]. Formally, the "minimal standard" RNG of Park and Miller is a *prime modulus multiplicative linear congruential generator*. Whew! Such one is parametrised by a large, prime integer m, and an integer $a \in \{2, \ldots, m-1\}$. Given a *seed* $z_0 \in \{1, \ldots, m-1\}$, for $n = 1, 2, \ldots$ it generates real numbers u_n by the iterative recipe:

$$u_n = z_n/m \tag{10}$$

$$z_{n+1} = f(z_n) \tag{11}$$

$$f(z) = az \bmod m \tag{12}$$

Because $a, z_0 \neq 0$, and arithmetic is done in the prime field $\mathbb{F}_m$, no z_n will take the value zero. In other words, the sequence does not collapse. $a = 1$ is prohibited for obvious reasons.

This type of generator was proposed by Lehmer [Leh51]. Actually, it is *pseudo random*, as the sequence of reals produced depends deterministically on the seed z_0 and the parameters m and a. However, as Park and Miller emphasise, careful selection of these makes the z_n "statistically indistinguishable from a sequence drawn at random (albeit without replacement [$z_i = z_{i+(m-1)}$]) from the set $\{1, \ldots, m-1\}$." If m is large enough, so a high resolution is secured, the u_n can analogously be regarded uniformly distributed on the interval $[\frac{1}{m}, 1-\frac{1}{m}]$. Park and Miller lay down two conditions, which explicitly require "good" parameters to have this characteristic.

With a third condition they additionally require, that it must be possible to implement the function f efficiently in a portable fashion with arithmetic operations, see [PM88, T_i, p. 1194]. Park and Miller recommend $m = 2^{31} - 1$, a Mersenne prime, and $a = 16807$ as a good tradeoff, and give a `Pascal` implementation [PM88, Integer Version 2, p. 1195] guaranteed to work if `maxint` $\geq m$. From the Internet a `C++` adaption, RANDGEN, by Chris Nevison can be downloaded[26].

[26] `http://cs.colgate.edu/faculty/nevison.pub/OOPTools/RandGen/RandomNumbers.htm`

References

[ADADC$^+$98] Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the sorting record: Experiences in tuning NOW-Sort. In *Proceedings of the 1998 Symposium on Parallel and Distributed Tools (SPDT '98)*, 1998.

[Aga96] Ramesh C. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 240–246. ACM, New York, 1996. *SIGMOD Record* 25(2), 1996.

[AH97] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997.

[AHNR98] Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57(1):74–93, 1998.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[AN] Arne Andersson and Stefan Nilsson. Implementing radixsort. To appear in ACM Journal of Experimental Algorithmics. Available for download from `http://www.nada.kth.se/ snilsson/public/papers.html`.

[AN94a] Arne Andersson and Stefan Nilsson. Faster searching in tries and quadtrees — an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, volume 855 of *Lecture Notes in Computer Science*, pages 82–93. Springer-Verlag, Berlin/Heidelberg, Germany, 1994.

[AN94b] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 714–721. IEEE, New York, 1994. Available for download from `http://www.nada.kth.se/ snilsson/public/papers.html`.

[And95] Arne Andersson. Sublogarithmic searching without multiplications. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 655–663. IEEE, New York, 1995.

[And96a] Arne Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 135–141. IEEE, New York, 1996.

[And96b] Arne Andersson. Sorting and searching revisited. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT '96)*, volume 1097 of *Lecture Notes in Computer Science*, pages 185–197. Springer-Verlag, Berlin/Heidelberg, Germany, 1996.

[Ano85] Anonymous. A measure of transaction processing power. *Datamation*, 31(7):112–118, 1985. Also in *Readings in Database Systems*, M. H. Stonebraker ed., Morgan Kaufmann, San Mateo, 1989.

[AT96] Arne Andersson and Mikkel Thorup. A pragmatic implementation of monotone priority queues. World Wide Web postscript document, 1996. From the Fifth DIMACS Implementation Challenge 1995-1996. `http://www.diku.dk/users/mthorup/PAPERS/papers.html`.

[BAG97] Amir M. Ben-Amram and Zvi Galil. When can we sort in $o(n \log n)$ time? *Journal of Computer and System Sciences*, 54(2):345–370, 1997.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[BH89] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, 1989.

[BM93] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software — Practice and Experience*, 23(11):1249–1265, 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Rondald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[DHKP97] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

[FW93] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[FW94] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gol93] S. R. Goldsmith. *Simulation of Multiprocessors, Speed and Accuracy.* PhD thesis, Stanford University, Stanford, New Jersey, 1993.

[Hag94] Torben Hagerup. Optimal parallel string algorithms: Merging, sorting and computing the minimum. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 382–391. ACM, New York, 1994.

[Hag98] Torben Hagerup. Sorting and searching on the word RAM. In *15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373, pages 366–398. Springer-Verlag, Berlin/Heidelberg, Germany, 1998.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, San Mateo, second edition, 1996.

[Jen96] Robert J. Jenkins, Jr. ISAAC and RC4. World Wide Web hypertext document, 1996. `http://ourworld.compuserve.com/homepages/bob_jenkins/isaac.htm`.

[Jon97] Neil D. Jones. *Computability and Complexity — From a Programming Perspective.* The MIT Press, Cambridge, Massachusetts, 1997.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[KR84] David Kirkpatrick and Stefan Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1984.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, second edition, 1988.

[KT97] Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, Berlin/Heidelberg, Germany, 1997.

[Leh51] D. H. Lehmer. Mathematical methods in large scale computing units. *Annu. Comput. Lab. Harvard Univ.*, 26:141–146, 1951.

[LL99] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300. ACM, New York, 1995. *SIGPLAN Notices* 30(6), 1995.

[LW94] Alvin R. Lebeck and David A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, 1994.

[LW97] Alvin R. Lebeck and David A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Computer Simulation*, 7(1):42–77, 1997.

[MBM93] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. In *Computing Systems*, volume 6, pages 5–27. USENIX Association, Berkeley, California, Winter 1993.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, Berlin/Heidelberg, Germany, 1984.

[MGA92] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Memspy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review*, 20(1), 1992.

[MGA95] Margaret Martonosi, Anoop Gupta, and Thomas E. Anderson. Tuning memory performance of sequential and parallel programs. *Computer*, 28(4):32–40, 1995.

[Mil96] Peter Bro Miltersen. Lower bounds for static dictionaries on rams with bit operations but no multiplication. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 442–453. Springer-Verlag, Berlin/Heidelberg, Germany, 1996.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press,, Cambridge, Massachusetts, 1995.

[Mus97] David R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.

[NBC+94] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. AlphaSort: A RISC machine sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 233–242. ACM, New York, 1994. *SIGMOD Record* 23(2), 1994.

[Nil96] Stefan Nilsson. *Radix Sorting & Searching.* PhD thesis, Lund University, Sweden, Department of Computer Science, 1996.

[PM88] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[PS82] W. J. Paul and J. Simon. Decision trees and random access machines. In *LANDA: Logic and Algorithmic: An International Symposium Held in Honour of Ernst Specker*, pages 331–340. L'Enseignement Mathematique, Universite de Geneve, Geneva, Switzerland, 1982.

[Ram96] Rajeev Raman. Priority queues: Small, monotone and trans-dichotomous. In *Proceedings of the Fourth Annual European Symposium*, volume 1136 of *Lecture Notes in Computer Science*, pages 121–137. Springer-Verlag, Berlin/Heidelberg, Germany, 1996.

[RR98] Naila Rahman and Rajeev Raman. An experimental study of word-level parallelism in some sorting algorithms. In *Proceedings of the Second Workshop on Algorithm Engineering*, pages 193–203, 1998.

[RR99] Naila Rahman and Rajeev Raman. Analysing cache effects in distribution sorting. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, 1999.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM—a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205. ACM, New York, 1994. *SIGPLAN Notices* 29(6), 1994.

[Spo99] Maz Spork. Design and analysis of cache-conscious programs. Master's thesis, Department of Computer Science, University of Copenhagen, Denmark, 1999.

[SS63] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[Tam83] M. Tamminen. Analysis of N-trees. *Information Processing Letters*, 16(3):131–137, 1983.

[Tan90] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall International, Inc., Englewood Cliffs, New Jersey, third edition, 1990.

[Tho96] Mikkel Thorup. On RAM priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67. ACM, New York. SIAM, Philadelphia, 1996.

[Tho97] Mikkel Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift and bitwise boolean operations. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 352–359. ACM, New York. SIAM, Philadelphia, 1997.

[Tho98] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555. ACM, New York. SIAM, Philadelphia, 1998.

[Tur36] A. M. Turing. On computable numbers with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.

[vEB77] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964.

[YSL98] C.-L. Yang, B. Sano, and A. R. Lebeck. Exploiting instruction level parallelism in geometry processing for three-dimensional graphics applications. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 14–24. IEEE, New York, 1998.