Hardware-Aware Algorithms and Data Structures

Gabriel Moruz

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Hardware-Aware Algorithms and Data Structures

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Gabriel Moruz
July 31, 2007

Abstract

Various computer hardware components are affecting the running time of algorithms in different proportions, or may have severe implications on the accuracy of algorithms. In this dissertation we propose algorithms and data structures that are efficient and robust with respect to different hardware factors. The hardware factors affecting the running time that we consider include branch mispredictions occurring in the processor, memory transfers occurring between consecutive levels of the memory hierarchy in modern computers, as well as the high rate sequential access on modern hard-disks which is a major motivation for developing streaming algorithms. Regarding the factors affecting the accuracy of algorithms, we consider soft memory errors which determine corruptions in RAM memories.

Branch mispredictions incur significant performance losses for algorithms. First, we show that the running time of randomized Quicksort is adaptive, i.e. it depends on the presortedness of the input. We prove theoretically that the number of element swaps performed by Quicksort depends on the number Inv of inversions in the input and show experimentally that the number of element swaps is closely correlated to the number of branch mispredictions. We then give lower bound trade-offs between comparisons and branch mispredictions for sorting and adaptive sorting, and propose sorting and adaptive sorting algorithms matching these bounds. Finally, we show experimentally that for random queries perfectly balanced binary search trees can be outperformed by skewed binary search trees, i.e. trees for which at any given node there is a fixed ratio between the nodes in the left and right subtrees. This happens because the skewed binary search trees perform more comparisons, but fewer branch mispredictions, compared to perfectly binary search trees for random queries.

Memory transfers occurring between consecutive levels of the memory hierarchy in modern computers are modeled in the I/O model and cache oblivious model, and the complexity of algorithms is given by the number of memory transfers performed. We introduce I/O lower bounds for adaptive sorting algorithms and give two algorithms that are optimal and I/O optimal with respect to Inv.

In a streaming setting, algorithms are restricted to access data sequentially, while having at their disposal a working memory that can be accessed for free, but which is usually much smaller than the problem size. We give general reductions from parallel algorithms to streaming algorithms, and show that we can obtain optimal algorithms (up to poly-log factors) for several combinatorial problems, such as sorting, connected components, minimum spanning tree,

biconnected components, or maximal independent set.

To analyze memory corruptions, we use the faulty-memory RAM model proposed by Finocchi and Italiano. In this model, any memory cell can get corrupted at any time during the execution of an algorithm, with no possibility of distinguishing between corrupted and uncorrupted cells. The number of corruptions is bounded by a parameter δ, known to the algorithm, and $O(1)$ corruption-free cells are provided. An algorithm is denoted resilient if it works correctly on the set of uncorrupted values. Our contributions include efficient resilient priority queues and resilient dictionaries.

Acknowledgments

I am deeply grateful to my adviser Gerth Stølting Brodal, for the countless hours of discussions and for all the support he offered me throughout my Ph.D. studies. He has guided my first steps in research on algorithms and it has been a real pleasure to work with him.

Many thanks to the people forming the algorithms group at University of Aarhus. In particular, I would like to thank my co-author Rolf Fagerberg, now at University of Southern Denmark, and my co-authoring fellow Ph.D. students Allan Grønlund Jørgensen and Thomas Mølhave for all the work done together. Also, I would like to thank Lars Arge for establishing the great group that MADALGO is becoming.

I would like to thank Camil Demetrescu for agreeing to work with me during my abroad stay in Rome. Also, many thanks to the algorithms group at University of Rome "La Sapienza" for the helpful discussions. In particular, I would like to thank my co-authors Camil Demetrescu, Bruno Escoffier, and Andrea Ribichini for all the research conducted together during the period that I spent in Rome.

I am very thankful to all my colleagues and friends in Aarhus, in particular to Allan, Anders, Bjarke, Christopher, Claus, Daniel, Doina, Fitzi, Irit, Jan, Jesper, Jesus, Jooyong, Johan, Kevin, Kristoffer, Manuel, Marco, Martin, Michael, Mikkel, Mirka, Philipp, Roland, Rune, and Tord, for all the great moments spent together. Special thanks to Bartek for his help as a mentor, to Chris for co-winning two table football tournaments, to Saurabh and Kirill for their always cheerful attitude, to Henrik and Thomas for organizing the weekly soccer events, and to Gosia and Darek for gluing the group together.

I am very grateful to all the administrative and technical staff for ensuring that everything went smoothly, in particular (not exclusively) to: Mogens Nielsen, Uffe Engberg, Lene Kjeldsteen, Karen K. Møller, Ellen Lindstrøm, Hanne F. Jensen, Else Magård, and Michael Glad. Also, I would like to thank Danish National Research foundation for funding my studies at BRICS.

Last, but not least, I am grateful to my parents and my sister for their constant love and support over the years, and for being there for me all the time. Without them I could have never made it.

Gabriel Moruz,
Århus, July 31, 2007.

Contents

List of Figures

Chapter 1

Introduction

A journey of a thousand miles begins with a single step.
— Lao-tzu

Since the early days of computer science, the design and analysis of algorithms emerged as a fundamental research area. Having in mind that every program running in a computer is based on an algorithm, developing fast algorithms is a key factor towards developing fast software. In this dissertation, we show that a major point in developing fast algorithms is taking advantage of the characteristics of the underlying hardware. We focus on hardware factors that have a great influence over the running time or the reliability of algorithms. Among the factors affecting the running time we focus on branch mispredictions, block transfers between the consecutive levels of the memory hierarchy, and the high sequential access rate on modern hard-disks which motivates streaming. We consider also memory corruptions caused by soft memory errors in RAM memories and consequently design algorithms and data structures aware of possibly corrupted data.

Typically, the performance of algorithms is quantified by measures like instructions performed by the CPU or memory accesses. From this perspective, efficient algorithms for an endless series of problems have been introduced during the past decades. However, a number of assumptions with respect to the actual behavior of the hardware have been made. Since these assumptions do not always hold, this creates a potential gap between the theoretical algorithm design and analysis on one hand and the actual behavior of algorithms in practice on the other hand. For instance, a typical assumption is that all the instructions have the same execution time. Should this be the case for most instructions, in the case of conditional branches it is not always true, since they may take as much as 30 CPU cycles to execute [103]. Another standard assumption is that all memory accesses take the same time. This is certainly not the case in practice, since, due to memory hierarchies found in most computers, accessing data stored in the CPU registers is about 10,000,000 times faster than accessing data stored on disk [61]. A problem getting more and more important to address, motivated by the memory design technologies and the seeming unending increase in storage requirements, is that the cells in RAM memories may get corrupted in practice by e.g. alpha particles, radiation or cosmic rays, and

therefore assuming at all times a reliable memory may have dire consequences on the output of algorithms.

The remainder of this chapter is structured as follows. We briefly describe modern hardware in Section 1.1, and then we show in Section 1.2 why traditional models used for analyzing algorithms are not always accurate. We then introduce the algorithmic problems we consider in Section 1.3, and a summary of the contributions of the present dissertation in Section 1.4. This chapter concludes with an outline of the rest of the dissertation in Section 1.5.

1.1 Hardware

In this section we briefly introduce the hardware architecture of modern computers, emphasizing the factors that have a strong effect on the running time of algorithms in practice. Also, we show that memory corruptions that occur in the RAM memories can have dramatic effects on the output of algorithms. A thorough discussion concerning the hardware is given in Chapter 2.

Nowadays computers are cutting-edge machineries, consisting of highly complex components, and hardware technologies evolve at an incredibly fast pace. For instance, based on an empirical observation, Moore's law [119] states that the number of transistors contained on an integrated circuit doubles every 24 months. Following this law, nowadays CPU's may contain as much as 125 millions transistors, in the case of Intel Pentium 4 Extreme [61]. Furthermore, similar laws state that storage capacity for RAM memories increases at the same rate as processing power, and that the rate of progression in hard-disk storage over the past decades has sped up more than once.

The running time of algorithms in practice is the result of interaction of a wide range of factors, occurring at all hardware levels. In the following we give brief descriptions of the hardware factors discussed in the present dissertation.

Instructions count. The classical way of determining the running time of algorithms was counting the number of instructions performed by the CPU. Even though there are several more factors affecting the running time, instruction count is still a good indicator for the performance of algorithms. However, due to improvements in CPU design, such as increasing the frequencies and the ability to execute multiple instructions per cycle, modern processors are able to execute more instructions in a time unit and consequently the number of instructions performed is diminishing its influence over the running time.

Branch mispredictions. To increase the clock speed, modern CPUs include instruction pipelines in their architecture, where the instructions are prefetched before being executed. When a conditional branch enters the pipeline, its outcome is not known prior to its execution and thus its direction must be predicted to ensure the prefetching of the following instructions. If the branch is incorrectly predicted, the whole pipeline must be flushed, since the instructions in the pipeline correspond to a wrong execution path. This obviously leads to a performance loss, which increases proportionally with the length of the pipeline.

In such a case, we say that a *branch misprediction* occurs. Since the pipelines are getting longer and longer (e.g. as much as 18 instructions for Pentium P4 and 31 for Intel Prescott), branch mispredictions are having an increasing influence over the running time of algorithms in practice.

To predict the outcome of conditional branches, modern CPUs employ *branch predictors*, which implement *branch prediction schemes*. There are two major categories of branch prediction schemes, static and dynamic. In a static prediction scheme, a conditional branch is predicted the same way every time, and its direction is set based on simple heuristics, e.g. predict forward branches taken and backward branches not taken. Dynamic branch prediction schemes are more complex and use the execution history to predict the direction of a branch. The increased complexity of dynamic branch predictors proves to be very useful in increasing the amount of succesfully predicted branches, achieving a rate of success of about 90%.

I/O transfers. Modern computers have several memory levels, each level having smaller size and access time than the next one. Typically, a desktop computer contains CPU registers, L1, L2, and L3 caches, main memory and hard-disk. The access time increases from one or even half a cycle for registers and level 1 cache to around 10, 100 and 10,000,000 cycles for level 2 cache, main memory and disk, respectively [61]. Therefore, the I/O transfers between memory and disk often become a bottleneck with respect to the running time of a given algorithm, and minimizing the number of I/O transfers is a necessary step towards achieving fast running times.

Streaming. Data stream processing has gained increasing popularity in the last few years as an effective paradigm for processing massive data sets. Huge data streams arise in several modern applications, including database systems, IP traffic analysis, sensor networks, and transaction logs [55,56,110]. Streaming is an effective paradigm also in scenarios where the input data is not necessarily represented as a data stream. Due to high sequential access rates of modern disks, streaming algorithms can be effectively deployed for processing massive files on secondary storage [62].

Memory corruptions. Memory devices continually become smaller, work at higher frequencies and lower voltages, and in general have increased circuit complexity [32]. Unfortunately, these improvements come at the cost of reliability [113, 114]. A number of factors, such as alpha particles, infrared radiation, and cosmic rays, can cause *soft memory errors* where a bit flips and as a consequence the value stored in the corresponding memory cell is corrupted. An unreliable memory can cause a variety of problems in most software, ranging from harmless to the very serious, such as breaking cryptographic protocols [18,120], taking control of a Java Virtual Machine [57] or breaking smart-cards and other security processors [5, 6, 108]. Also, the output of algorithms that are unaware of possibly corrupted memories may be severely affected. A typical example concerns merging two sorted sequences, where a single corrupted memory cell

may induce as much as $\Theta(n^2)$ inversions in the output sequence [51].

1.2 Modelling hardware

The running time of algorithms is usually analyzed by counting the instructions performed by the CPU on an abstract model of computation, e.g. the RAM model. However, we argued that in practice some other hardware factors, in addition to CPU instructions, can have a non-negligible effect over the running time of an algorithm. In this section we briefly introduce the models that handle the different hardware factors.

Traditional models. Traditionally, the time complexity of algorithms is computed by counting the number of instructions performed by the CPU. Below we introduce two of the most popular models of computation, the RAM model and the comparison model.

RAM model. The RAM (Random Access Machine) model is an abstraction of a real-world computer. It consists of a CPU and a memory of unlimited size, where CPU instructions and memory accesses take the same amount of time. The complexity of an algorithm is given by the number of instructions and memory accesses performed.

Comparison model. In this model, the complexity of an algorithm is given by the number of comparisons performed. Though it does not include all the instructions performed by the CPU, the comparison model is suitable when dealing with problems where comparisons are the key instructions, e.g. sorting and searching.

Branch mispredictions. When analyzing the branch misprediction complexity of an algorithm, we will use a static prediction scheme. In such a scheme, every branch is predicted the same direction at all times. Basically, we assign for each branch *a priori* a direction in which it will be predicted during the execution of the algorithm. A major drawback in using static prediction schemes to analyze the branch misprediction complexity of algorithms is that they are less accurate than the dynamic prediction schemes in practice. However, static prediction schemes are simple to analyze and are suited for hard problems, such as sorting and searching, where the output of a comparison is very hard to predict.

I/O transfers. Several models have been proposed to capture the effect of memory hierarchies.We describe two of the most successful, the I/O model and the cache-oblivious model.

I/O model. The I/O model was introduced by Aggarwal and Vitter [1]. It is a model closer to real world hardware and models a simple two-level memory hierarchy consisting of a fast memory of size M and a slow infinite memory. The data transfers between the slow and fast memory

are performed in *blocks* of size B of consecutive data. In this model the algorithm has full control of the memory management, including location of the data and the replacement policy. The performance of an algorithm is given by its I/O complexity, which is obtained by counting the number of transfers it performs between the slow and the fast memories.

Cache-oblivious model. A drawback of the I/O model is the assumption that the size M of the fast memory and the block size B are known, which does not always hold in practice. Moreover, as the modern computers have multiple memory levels with different sizes and block sizes, different parameters are required at the different memory levels. Frigo et al. [52] proposed the *cache-oblivious model*, which is similar to the I/O model, but assumes no knowledge about M and B. In short, a cache-oblivious algorithm is an algorithm described in the RAM model, but analyzed in the I/O model with an analysis valid for any values of M and B. The power of this model is that if a cache-oblivious algorithm performs well on a two-level memory hierarchy with arbitrary parameters, it performs well between all the consecutive levels of a multi-level memory hierarchy. Since the algorithm is oblivious of the memory it operates on, a standard assumption of this model is an optimal replacement strategy for the underlying memory levels. Also, most algorithms require a tall cache assumption, i.e. $M = \Omega(B^{1+\varepsilon})$.

Streaming. In the recent years, due to increased popularity of data stream processing, several streaming models have been proposed. In such models, the input data is accessed sequentially, using a small amount of working memory, which is usually much smaller than the input size [13,62,87,88]. Typical parameters in streaming models include the number p of passes over the input stream and the number s of bits contained by the working memory. In the following we list some of the streaming models.

Classical streaming. In classical streaming, the input stream is read-only, and we are interested in algorithms that require one (or few) passes for solving a given problem. Given the little amount of resources that a streaming algorithm has, typical results obtained in this model are approximations of an exact solution.

W-Stream. The W-Stream model was introduced by Demetrescu et al. [37]. In this model, at each pass we operate with an input stream and an output stream. The streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i + 1$.

StrSort. StrSort model is just W-Stream augmented with a sorting primitive that can be used at each pass to reorder the output stream at no cost, and is motivated by the fact that sorting is a well-studied problem in the context of massive inputs. Sorting provides a significant amount of computational power, making it possible to solve several graph problems using poly-log passes and working space [2].

Memory corruptions. In practice, cells in RAM memories may get corrupted as a result of soft memory errors. Since most algorithms assume a reliable memory, they are not designed to handle memory corruptions and their output may be severely affected. A classical example concerns merging two sorted sequences, where a single memory corruption may induce as many as $\Theta(n^2)$ inversions [51]. To design algorithms that are aware of memory corruptions, we use the *faulty-memory random access machine*, introduced by Finocchi and Italiano [51]. A faulty-memory RAM is a random access machine where the content of memory cells can get corrupted at *any time* and at *any location*. Corrupted cells cannot be distinguished from uncorrupted cells. The model is parametrized by an upper bound δ on the number of corruptions occurring during the lifetime of an algorithm. It is assumed that $O(1)$ reliable memory cells are provided, a reasonable assumption since CPU registers are considered reliable. Also, copying an element is considered an atomic operation, i.e. the elements are not corrupted while being copied. An algorithm is *resilient* if it is able to achieve a correct output at least for the uncorrupted values. This is the best one can hope for, since the output can get corrupted just after the algorithm finishes its execution. For instance a resilient sorting algorithm guarantees that there are no inversions between the uncorrupted elements in the output sequence.

1.3 Algorithmic problems studied

In this section we give an overview of the problems studied throughout the rest of the dissertation. We are interested in developing algorithms and data structures that are aware of the various hardware factors affecting the running time. We develop algorithms for fundamental problems, such as sorting, adaptive sorting, and basic algorithms for undirected graphs (e.g. connected components, minimum spanning tree, biconnected components, maximal independent set). Also, we study fundamental data structures, such as priority queues and static and dynamic dictionaries.

Sorting and adaptive sorting Given a sequence of elements from a totally ordered universe, sorting means producing a sequence containing all the elements in non-decreasing order.

A well known fact concerning sorting is that optimal sorting algorithms perform $\Theta(n \log n)$ comparisons [34, Section 9.1]. However, in practice there are many cases where the input sequences are already nearly sorted, i.e. have low disorder according to some measure [74, 85]. In such cases one can hope for a sorting algorithm to be faster.

In order to quantify the disorder of input sequences, several *measures of presortedness* have been proposed, e.g. see [43, 74, 82]. One of the most commonly considered measures is *Inv*, the number of inversions in the input, defined by $Inv(X) = |\{(i,j) \mid i < j \ \wedge x_i > x_j\}|$ for a sequence $X = (x_1, \ldots, x_N)$. Other examples of measures include: *Runs*, the number of boundaries between ascending subsequences; *Max*, the largest difference between the ranks of an element

in the input and the sorted sequence; *Dis*, the largest distance determined by an inversion. A sorting algorithm is denoted *adaptive* if the time complexity is a function dependent on the size as well as the presortedness of the input sequence [85]. For an overview concerning adaptive sorting, see e.g. the survey by Estivill-Castro and Wood [45]. More recent works include [39–41, 90, 93].

Manilla [82] introduced the concept of optimality of an adaptive sorting algorithm in the comparison model. An adaptive sorting algorithm S is optimal with respect to some measure of presortedness $\mathcal{D}$, if for some constant $c > 0$ and for all inputs X, the time complexity $T_A(X)$ satisfies

$$T_A(X) \leq c \cdot \max(N, \log |\text{below}(X, \mathcal{D})|) \ ,$$

where $\text{below}(X, \mathcal{D})$ is the number of permutations of the input sequence Y for which $\mathcal{D}(Y) \leq \mathcal{D}(X)$ and $\log x$ denotes $\log_2 x$. By the usual information theoretic lower bound, this is asymptotically the best possible. In particular, an adaptive sorting algorithm that is optimal with respect to the measure *Inv* performs $\Theta(N(1 + \log(1 + Inv/N)))$ comparisons [58].

Priority queues. A priority queue is a data structure that contains a set of elements drawn from a totally ordered universe, supporting two operations, INSERT and DELETE-MIN. The INSERT operation inserts an element, while the DELETE-MIN operation returns and removes the element having the smallest key from the data structure. In addition, priority queues may support various operations such as decrease-key, deletions, melding, in addition to the two standard ones,

Static and dynamic dictionaries. A dictionary is a data structure that contains a totally ordered set of elements. Given a search key x, a static dictionary returns a boolean value stating whether x is contained in the data structure or not. A dynamic dictionary supports also updates, i.e. insertions and deletions, in addition to searches.

Graph algorithms. Consider an undirected graph $G = (V, E)$. A *connected component* of G is a maximal set of vertices $V' \subseteq V$ such that there exists a path between any pair of vertices in V'. A *biconnected component* of a graph G is a maximal subset of edges E' such that in the subgraph induced by E' there are two node-disjoint paths between any pair of vertices.

Consider a connected undirected graph $G = (V, E)$ having weighted edges, i.e. every edge (u, v) is assigned a cost $w(u, v)$. A *minimum spanning tree* T of G is a tree containing all the vertices in V and having minimal cost, i.e. $\sum_{(u,v)\in T} w(u, v)$ is minimal.

1.4 Contributions

In this section we list the main contributions of the present dissertation. We give detailed description of our results in Chapters 4-10, which present the work

published in [24, 28–31, 36, 70][1].

Cache-aware and cache-oblivious adaptive sorting (Chapter 4, [28]). We recall that sorting algorithms are denoted adaptive if their running times depend not only on the input size, but also on some measure of presortedness quantifying how close to being sorted the input is. A popular measure of presortedness is the number *Inv* of inversions in the input sequence. In [28] we first provide lower bounds on the number of I/O transfers for various measures of presortedness. We then introduce two adaptive sorting algorithms which are optimal with respect to *Inv*. From both algorithms we derive adaptive sorting algorithms that are optimal with respect to *Inv* in both the I/O model and the cache-oblivious model.

Quicksort is adaptive (Chapter 5, [29]). Quicksort was introduced by Hoare [64,65]. It is a simple in-place, randomized sorting algorithm that became very popular over the last decades. A text-book result states that Quicksort is optimal (expected case), i.e. it performs expected $\Theta(n \log n)$ comparisons [34]. A known fact is that Quicksort is not adaptive, since it performs expected $O(n \log n)$ comparisons even when the input sequence is sorted. In [29] we demonstrate empirically that the running time of Quicksort is adaptive with respect to the number of inversions *Inv* in the input. Differences in running times close to a factor of two are observed between instances with low and high *Inv* values. We prove that Quicksort performs expected $O(n(1 + \log(1 + Inv/n)))$ element swaps, where *Inv* denotes the number of inversions in the input sequence. This result provides a theoretical explanation for the observed behavior in practice, and gives new insights on the behavior of the Quicksort algorithm. We also give some experimental results on the adaptive behavior of Heapsort and Mergesort.

Optimal tradeoffs between comparisons and branch mispredictions for sorting algorithms (Chapter 6, [30]). In [30] we consider tradeoffs between the number of branch mispredictions and the number of comparisons for sorting algorithms in the comparison model, where each comparison is followed by a branch. We prove that a sorting algorithm using $O(dn \log n)$ comparisons performs $\Omega(n \log_d n)$ branch mispredictions. We show that Multiway MergeSort achieves this tradeoff by adopting a multiway merger with a low number of branch mispredictions. For adaptive sorting algorithms we similarly obtain that an algorithm performing $O(dn(1+\log(1+Inv/n)))$ comparisons must perform $\Omega(n \log_d(1 + Inv/n))$ branch mispredictions, where *Inv* is the number of

[1]The paper **Optimal Resilient Dynamic Dictionaries** by Gerth S. Brodal, Rolf Fagerberg, Irene Finocchi, Fabrizio Grandoni, Giuseppe F. Italiano, Allan G. Jørgensen, Gabriel Moruz, and Thomas Mølhave [24], has been accepted for publication as a merged paper. The original contributions were **Resilient Search Trees: Randomization and Prejudice**, by Irene Finocchi, Fabrizio Grandoni, and Giuseppe F. Italiano, and **Optimal Resilient Dynamic Dictionaries**, by Gerth S. Brodal, Rolf Fagerberg, Allan G. Jørgensen, Gabriel Moruz, and Thomas Mølhave. The contributions of the latter will appear as a technical report in [27].

inversions in the input. This tradeoff can be achieved by the algorithm GenericSort by Estivill-Castro and Wood by adopting a multiway division protocol and a multiway merging algorithm with a low number of branch mispredictions.

Skewed binary search trees (Chapter 7, [31]). It is well known that binary search trees achieve the best performance for a random search when they are perfectly balanced. Should this statement be true for the number of comparisons, which is indeed minimal when the trees are perfectly balanced, it does not hold for the running time. In [31], we study static skewed binary search trees, where for each node the ratio between the number of nodes in the left subtree and the size of the tree is a fixed constant. We show experimentally that skewed binary search trees can outperform perfectly balanced search trees by as much as 15% in running time. These improvements are due to the fact that during a search branching to the left or right at a node does not necessarily have the same cost, because of branch mispredictions. Basically, the more skewed the search tree is, the more comparisons and fewer branch mispredictions it performs. Also, previous work has shown that a dominating factor over the running time for a search is the number of cache faults performed, and that an appropriate memory layout of a binary search tree can reduce the number of cache faults by several hundred percent. We give an experimental study of various memory layouts of static skewed binary search trees, where each element in the tree is searched for with a uniform probability.

Parallel algorithms are good for streaming (Chapter 8, [36]). We show in [36] how to simulate parallel algorithms to obtain efficient algorithms in the W-Stream model. Motivated by the fact that simulating classical PRAM algorithms does not always help in achieving good results in the W-Stream model, we introduce a new model of computation denoted RPRAM (Relaxed PRAM). Basically a RPRAM is a classical PRAM where a processor is no longer constrained to access $O(1)$ memory cells in a parallel round, but can access as much as all the cells in the memory. By simulating RPRAM algorithms in the W-Stream model we are able to achieve optimal W-Stream algorithms (up to poly-log factors) for several classical combinatorial problems, such as sorting, connected components, minimum spanning tree, biconnected components, and maximal independet set.

Resilient priority queues (Chapter 9, [70]). We recall that in the faulty-memory RAM memory cells can get corrupted at any time and at any place during the execution of an algorithm, and the number of corruptions is upper bounded by a parameter δ. An algorithm or data structure is denoted resilient if it works correctly on the set of uncorrupted values. In particular, for priority queues, the DELETEMIN operation returns the minimum uncorrupted value or some corrupted value. In [70] we introduce a resilient priority queue which uses $O(n)$ space to store n elements, and supports both insert and deletemin operations in $O(\log n + \delta)$ time amortized. Our priority queue matches the performance of classical optimal priority queues in the RAM model when the

number of corruptions tolerated is $O(\log n)$. We prove matching worst case lower bounds for resilient priority queues storing only structural information in the uncorruptible registers between operations.

Optimal resilient dictionaries (Chapter 10, [24]). We introduce in [24] two optimal resilient static dictionaries, a randomized one and a deterministic one. The randomized dictionary supports searches in $O(\log n+\delta)$ expected time using $O(\log \delta)$ random bits in the worst case under the assumption of an oblivious adversary. The deterministic dictionary supports searches in $O(\log n + \delta)$ time in the worst case. We introduce a dynamic resilient dictionary supporting searches in $O(\log n + \delta)$ time in the worst case, which is optimal, and updates in $O(\log n+\delta)$ amortized time. Our dynamic dictionary supports range queries in $O(\log n + \delta + k)$ worst case time, where k is the size of the output.

1.5 Outline

The remainder of the dissertation is structured as follows. We give a thorough discussion about hardware in Chapter 2, and we discuss the various computation models, including related work, in Chapter 3. Chapters 4-10 contain the technical contributions of this dissertation. In Chapter 4 we discuss cache-aware and cache-oblivious adaptive sorting. Chapter 5 is devoted to showing that the running time of Quicksort is adaptive with respect to the number of inversions in the input, and we show adaptive behaviors also for Mergesort and Heapsort. We prove lower and upper bounds stating optimal tradeoffs between branch mispredictions and comparisons for sorting and adaptive sorting algorithms in Chapter 6, and then we discuss the skewed binary search trees in Chapter 7. In Chapter 8 we focus on streaming algorithms, and show how to achieve efficient algorithms for several fundamental combinatorial problems by simulating parallel algorithms. Finally, we introduce resilient priority queues in Chapter 9 and optimal resilient dictionaries in Chapter 10.

Chapter 2

Hardware

Hardware /nm./: the part of the computer that you can kick.
— Unknown

In this chapter we give a thorough description of the architecture of nowadays computers, emphasizing the key components affecting the running time. We first describe the CPU in Section 2.1, and then the memory hierarchy in Section 2.2. Finally, in Section 2.3 we discuss memory corruptions.

2.1 Processor

The processor is the core of any computing system. Nowadays processors are highly complex electronic machineries, and processor technologies evolve at amazing rates. For instance, due to fast progress in nanotechnology, in 1997 the Intel MMX processor was designed on 350nm (nano-meters) technology, nowadays processors use 65nm technology, and the predictions state that in about 2010 processor technology will be done on 32nm [97, 98]. This huge progress allows a typical nowadays processor to contain as many as 125 million transistors on a die size of only 112mm^2 [17].

We introduce in Figures 2.1 and 2.2 the architectures of two popular processors, the Pentium 4 and AMD Opteron respectively, as illustrated in [17] and [73]. The two architectures are very similar and the key components are the same. They both contain a L1 data cache and L1 instruction cache, the latter denoted Execution Trace Cache for the Intel architecture, as well as an unified L2 cache. Both architectures handle integer operations separately from floating point and multimedia operations. The main difference between them is that the instruction cache for the Pentium architecture contains decoded instructions unlike the AMD. Due to the more extensive documentation at our disposal, in the remainder of this section we restrict ourselves to give more detail for the Pentium 4.

The typical execution flow for the Intel Pentium 4 architecture starts with prefetching, decoding and storing the resulted micro-operations (μops) in the Execution Trace Cache. The μops are then scheduled for execution in one of the integer Arithmetic and Logical Units (ALU) or one of the two blocks dedicated

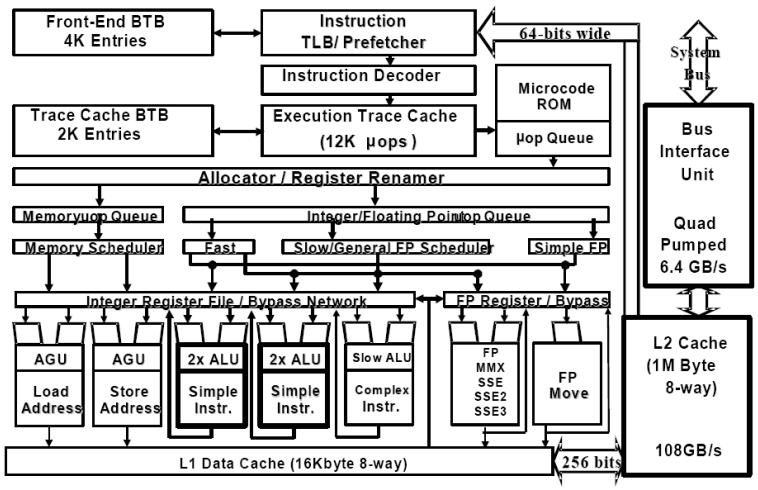

Figure 2.1: The architectures for Intel Pentium 4 [17].

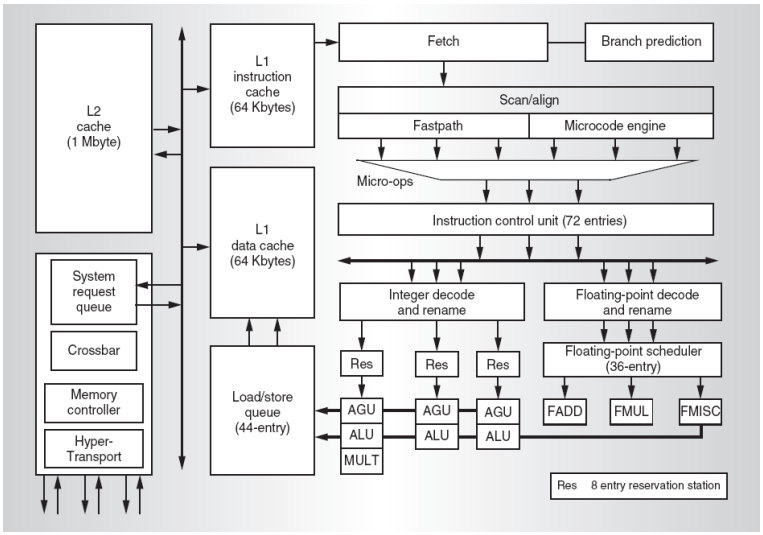

Figure 2.2: The architectures for AMD Opteron [73].

to floating point (FP) instructions. The instructions are executed once the operands are fetched from the data cache.

To improve the instruction execution of the CPU, a series of enhancements and optimizations have been performed. For instance, in the case of conditional branches, a branch predictor has been added to predict the output of a given branch prior to its execution, such that the instructions that follow the branch can be prefetched. A thorough description of branch predictors is given in Section 2.1.2. Another enhancement concerns store-to-load forwarding and concerns scheduling the instructions to the various execution units. When an instruction is scheduled for execution, its data may be the result of another

instruction not yet executed, and the processor needs to make sure the given instruction has this data. For this purpose, the Pentium 4 processor employs a Store Forwarding Buffer which contains the results of previous instructions, or previous stores for the L1 data cache.

A recent and very significant improvement in processor design concerns developing multi-core processors. A multi-core processor contains several execution cores, and the main motivation for using them lies in the relationship between frequency and power. For instance, by overclocking a processor to achieve a 13% gain in performance, the price to pay is an increase by 73% in power consumed, while by losing 13% in frequency by underclocking the processor, the power consumption reduces to about a half. Therefore, using two under-clocked cores in a multi-core processor, one can achieve 73% more in performance for the power used by a regular single-core processor [98], or for the same performance significant gains are achieved in power consumption.

In the following we analyze the components that are the main time consumers for the processors in practice and which are discussed in the remainder of the dissertation. We first describe the execution core, and then we focus on branch mispredictions. The L1 and L2 caches as well as the TLB (Translation Lookaside Buffer), though parts of the processor, will be discussed in Section 2.2, which is devoted to the study of the memory hierarchy.

2.1.1 Execution engine

Processor performance often refers to the amount of time it takes to execute some given application, or the ability to run multiple applications in a given period of time. Usually it is quantified by multiplying the frequency and the IPC (Instructions Per Cycle) [118], hence in this acception the performance of a processor is given by the speed at which the processor executes instructions. For a long time processor design engineers worked mostly on increasing the frequency, which reached 3 Ghz in 2002 from only 5 Mhz in 1983. By 2002, due to limitations generated by power densities and the resulting heat, increasing the IPC receives more and more attention. Two direction have been adopted to increase the IPC. The first one consists in extending the number of instructions as well as increasing the instruction set performed by a processor in a cycle, and the second one concerns developing multi-core processors [98].

Taking a closer look at the architecture of the Pentium 4, see e.g. Figure 2.1, we notice that there are three execution blocks dedicated to integer instructions and two for floating point instructions. Furthermore, two of the integer execution blocks operate at double speed and can execute two instructions in a cycle.

2.1.2 Branch mispredictions

Nowadays CPUs have high memory bandwidth and increased pipelines, e.g. Intel Pentium 4 Prescott has a 31 stage pipeline. The high memory bandwidth severely lowers the effect of caching over the actual running time when computation takes place in the internal memory. More precisely, faster memory

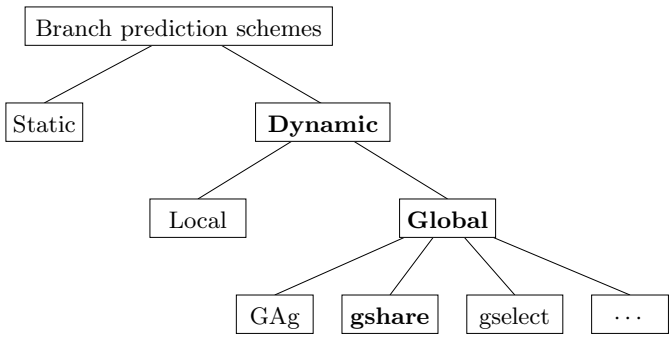

Figure 2.3: A classification of the branch prediction schemes. The most popular branch predictors in each category are emphasized.

transfers imply that the overhead incurred by a cache miss is less costly and thus has a smaller effect on the running time of an algorithm.

When a conditional branch enters the execution pipeline of the CPU, its outcome is not known and therefore must be predicted. If the prediction is incorrect, the pipeline is flushed as it contains instructions corresponding to a wrong execution path. Obviously, each branch misprediction results in performance losses, which increase with the length of the pipeline.

Several branch prediction schemes have been proposed. A classification of the branch prediction schemes is given in Figure 2.3.

In a static prediction scheme, every branch is predicted in the same direction every time according to some simple heuristics, e.g. all forward branches taken, all backward branches not taken. Although simple to implement, their accuracy is low and therefore they are not widely used in practice.

The dynamic schemes use the execution history when predicting a given branch. In the local branch prediction scheme (see Figure 2.4, left) the direction of a branch is predicted using its past outputs. It uses a *pattern history table* (*PHT*) to store the last branch outcomes, indexed after the lower n bytes of the address of the branch instruction. However, the direction of a branch might depend on the output of other previous branch instructions and the local prediction schemes do not take advantage of it. To deal with this issue global branch prediction schemes were introduced [122]. They use a *branch history register* (BHR) that stores the outcome of the most recent branches. The different global prediction schemes vary only in the way the prediction table is looked up.

Three global branch prediction schemes proved very effective and are widely implemented in practice [83]. The *GAg* (Figure 2.4, middle) uses only the last m bits of the *BHR* to index the pattern history table, while *gshare* address the *PHT* by xor-ing the last bits n of the branch address with the last m bits of the *BHR*. Finally *gselect* concatenates the *BHR* with the lower bits of the branch address to obtain the index for the *PHT*.

The predictions corresponding to the entries in the *PHT* are usually obtained by the means of *two-bit saturating counters*. A two-bit saturating counter is an automaton consisting of four states, as shown in Figure 2.5.

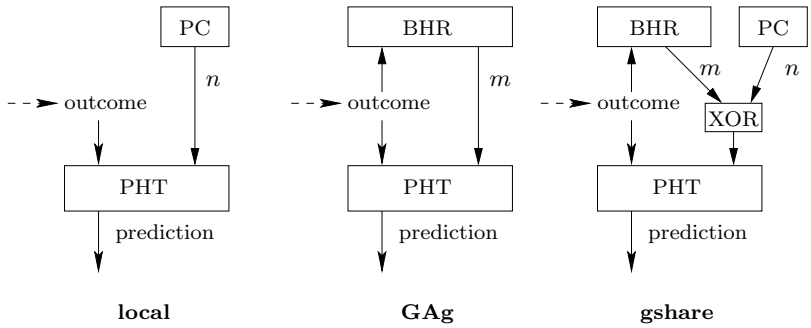

Figure 2.4: Branch prediction schemes.

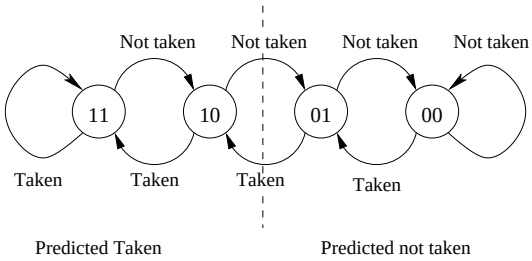

Figure 2.5: Two-bit saturating counter.

Note that for the dynamic branch prediction schemes the same index in the *PHT* might correspond to several branches which would affect each other's predictions, constructively or destructively. This is known as the *aliasing effect* and reducing its negative effects is one of the main research areas in branch prediction schemes design.

We show the effect of branch mispredictions over the running time by the means of a simple example. We generate a random array a of fixed length $n = 2 \times 10^7$, with $a[i]$ chosen uniformly at random in $[1 \dots 100]$. In a left-to-right scan we count the number of elements greater and smaller or equal than a given parameter $param$, see e.g. the C source code in Figure 2.6. We measure the running time of this code in two different settings, optimized with optimization -O3 and using no optimizations respectively, for different values of parameter $param$ in the range $[0, \dots, 101]$. Intuitively, the inner branch is easy to predict when the value of $param$ is close to the endings of the interval, and becomes very hard to predict when the value of $param$ gets closer to the middle of interval. In Figure 2.7 we present the results of our experiments, where each plotted value represents the average over three different runs. When using no optimizations, the chart has a symmetric behavior, the maximum running time is achieved for $param \approx 50$, and the maximum running time is larger by about 75% than the minimum running time. When using optimization -O3, the running times are significantly faster, and the maximum running time is about 400% larger than the minimum running time. However, the chart does not exhibit a symmetric behavior, but the maximum running time is achieved fir $param \approx 65$, and the difference in running times for $param \approx 0$ and $param \approx 100$ is of about 300%. By the means of a brief inspection of the assembly code, we explain the

different behaviors by the fact that in the optimized code when the branch is taken, corresponding to a value larger than *param*, the code contains a jump instruction more than in the case when the branch is not taken, and this does not happen in the case when the code is not optimized.

```
for(i=0;i<n;i++)
if(a[i]>param)
  g++;
else
  s++;
```

Figure 2.6: Simple example demonstrating the effect that branch mispredictions have over the running time. The size of the array a is $n = 2 \times 10^7$ and the values $a[i]$ are generated uniformly at random in $[1, \ldots, 100]$.

Sanders and Winkel [103] gave a distribution based sorting algorithm and show that in certain cases branch mispredictions can be avoided by using predicated instructions available in certain processors, e.g. Intel Itanium. A predicated instruction is an instruction which is associated a predicate. The instruction is executed if the predicate is true, and is discarded otherwise.

2.2 Memory hierarchy

This section is devoted to the study of the memory hierarchy found in modern computers. Nowadays computers contain several memory levels, each level having smaller size and access time than the next one. Typically, a desktop computer contains CPU registers, L1, L2, and L3 caches, main memory and hard-disk. From the CPU to the hard-disk, each level has a smaller size and access time compared to the next level, see e.g. Figure 2.10. The access time increases from one or even half a cycle for registers to around 1,000,000 cycles for the disk [61]. To amortize the cost of a memory access per element, data is

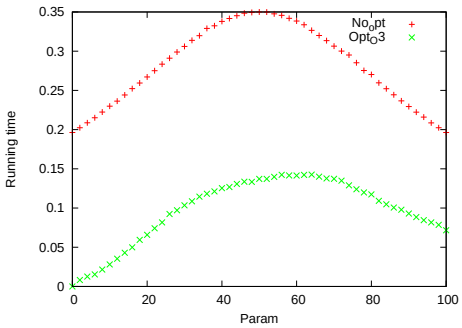

Figure 2.7: The running time of the code in Figure 2.6, using optimization -O3 and no optimization at all, for different values of parameter *param*.

transferred in blocks between consecutive memory levels. Upon a data request for a given level, if the data requested is contained in that level we say we have a *cache hit*, and we say a *cache miss* or *cache fault* occurred otherwise. When a cache fault occurs, the data required is fetched from the next level at the cost of a block transfer, and a block is evicted from the current level. To decide which block to be evicted, several replacement strategies were proposed, out of which LRU (Least Recently Used) and its variants became very popular.

Similarly to branch mispredictions, we demonstrate the effect that memory transfers have over the running time using a simple example. We consider an array of size n, where n is a parameter, and access its elements in a cyclic manner for a constant number of steps r, see. e.g. the source code in Figure 2.8. We compiled our code using optimization level -O3, and to avoid compiler optimizations disregarding the memory accesses we compute the sum of the elements visited and return it.

```
r=800000000;s=0;
for(i=0;i<n-1;i++)
  a[i]=i+1;
a[n-1]=0;

/* start timing */
for(i=0;i<r;i++)
{
  s+=a[k];
  k=a[k];
}
/* end timing */
return s;
```

Figure 2.8: Simple example demonstrating the effect that memory transfers have over the running time. We consider an array of size n and perform $r = 8 \times 10^8$ element accesses in a cyclic manner, by computing the sum. We measure the running time of the code indicated by comments.

We are interested in the running time of the code indicated by the appropriate comments, when varying the size n of the array. In Figure 2.9 we show the experimental results. First, we note that for $\log n \approx 28$ the running time increases by a factor of about 2000 (Figure 2.9, left). This step corresponds to the limit when the array still fits in the internal memory. Intuitively, as long as the arrays fits in the internal memory the memory accesses take little time and the computation is very fast. When the array does not fit in the internal memory, initially the first part of array fills completely the main memory. After this moment, given the fact that the replacement policy usually removes the least recently accessed block, each block access results in a cache miss and, given the large access times for disks, the running time increases dramatically. However, in Figure 2.9 (right), by restricting the range of the y-axis, we show

that a similar behavior is observed between the other levels of the memory hierarchy, resulting in running time increases for $\log n \approx 12$ and $\log n \approx 17$, corresponding to thresholds when the array does not fit in the L1 and the L2 cache respectively.

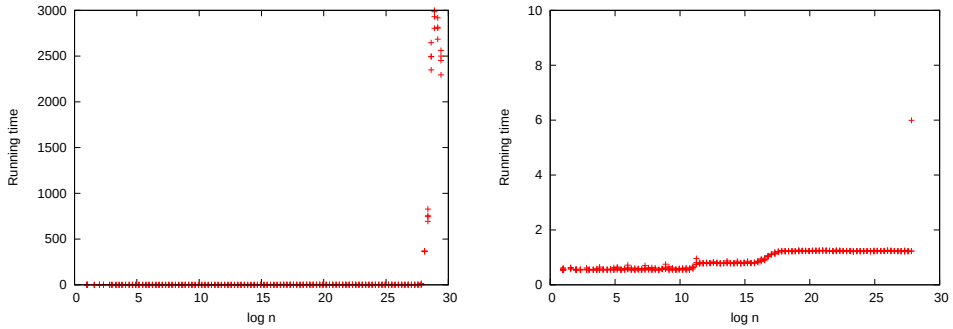

Figure 2.9: Running time of the highlighted code in Figure 2.8. The right chart plots the same data as the left chart, but restricts the range of the y-axis.

Given their effect over the running time, it becomes obvious that minimizing memory transfers, especially between disk and the main memory, is a key factor in achieving fast algorithms in practice. We describe the structure of the different memory levels and give more insights for their behavior in practice. We start by describing the registers and caches, then we move on to introduce the main memory, and we conclude by presenting the hard-disk.

2.2.1 Registers and caches

The registers are memory cells with very fast access times built inside the main core of the processor. The data is loaded into the registers from the cache memory before being processed and is stored still in the registers after processing. Since there is a big gap between the speeds at which the processor and the RAM memory operate, modern computers employ a few intermediate memories, called cache memories, which have small sizes, but fast access times.

We briefly introduce the organization of cache memories. As with all levels in the memory hierarchy, a cache memory is a collection of disjointed blocks of constant size. We first describe the direct mapped and fully-associative caches. The direct mapped cache are easy to implement in practice, but achieve rather

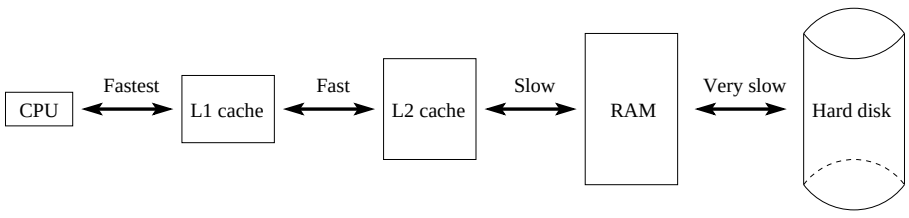

Figure 2.10: Typical memory hierarchy in a modern computer.

poor performances, while fully associative caches yield good performances at high design cost. We then introduce the set associative caches, which are a compromise between direct-mapped caches and fully-associative caches and which are widely implemented in practice.

A cache is denoted *direct mapped* when the location of each block in the cache is given by its memory address. In this case the replacement policy becomes trivial, since the page to be evicted is given by the memory address of the incoming cache line. Given the trivial replacement policy, direct-mapped caches are easy to implement in practice. However, because the replacement policy is decided by the memory address and not by the usage, the use of direct-mapped caches can cause significant performance losses by *thrashing*, when useful information is evicted from the cache. For instance, repeated interleaved accesses of two memory blocks mapping to the same cache location results in a cache miss for each access.

A *fully-associative* cache dramatically improves the performance of direct-mapped caches by using a much more sophisticated replacement strategy. Deciding which page to be evicted considers the usage of the different blocks instead of their memory addresses. A typical strategy is LRU (Least Recently Used), which replaces the oldest block with respect to the time of the last usage. Unfortunately, the very good performance achieved by fully-associative caches in practice involves non-negligible costs in circuitry required to keep track of the usage of all the blocks in the cache.

To combine the performance of the fully-associative caches and the simple design of the direct-mapped ones, hardware designers introduced *n-way set associative* caches. A n-way set associative cache is a collection of n direct-mapped caches, referred to as sets [115]. The replacement strategy decides the block to be evicted by first choosing one of the sets according to LRU or any other replacement strategy, and then decides the block to be evicted in the chosen set according to the memory address of the incoming block. This way, the circuit complexity is maintained simple enough, and thrashing is reduced. For a more comprehensive discussion on cache memories, we refer the interested reader to [63, 111].

We now describe the cache memories in a typical computer.

CPU caches. Modern computers include at least two levels of cache memories in their architecture. Usually processors include in their architecture two L1 cache memories, namely the instruction cache and the data cache, to store instructions to be executed and the data required by the programs respectively. Because the two L1 caches work independently, we achieve a significant increase in performance. On the other hand, it is common that there is only one L2 and (if available) L3 cache, which is unified, meaning that it contains both instructions and data. For better performance, a typical design has the L1 caches on the CPU chip, the L2 cache inside the CPU package, and the L3 cache (if available) on the motherboard.

TLB. The Translation Lookaside Buffer (TLB) is a small fully-associative cache memory used to store associations between virtual memory addresses and physical memory addresses. Since translating a virtual memory address

into a physical memory address is a rather costly operation, storing this information in a cache inside the CPU is necessary to avoid severe performance losses. Some processors, such as UltraSPARC, go even further and include two TLBs in their designs, to store associations between virtual and physical memory addresses for instructions and data separately [115].

2.2.2 Main memory

Typically, the main memory of a computer is a RAM (Random Access Memory), has much larger size than the cache memories and is dedicated to store vast amounts of information required by the various applications. The time required for accessing a block in a RAM is approximately the same for all blocks and is reasonably small.

2.2.3 Hard-disk

Unlike RAM memories which consist of integrated circuits, the hard-disk is a highly mechanical component. It consists of several platters, and each face of a platter is assigned a read/write head. The surface of each face of a platter is divided into a number of concentric circles denoted tracks, and each track is further subdivided in a number of sectors. The data is written to and read from the sectors. We note that since the inner tracks have a much shorter length than the outer tracks, the number of sectors in a track is not constant. To access the data on a platter, the head first identifies the track to access and then the platter is rotated until the required location is encountered. We note that the read/write head can move only inwards and outwards and can not be placed always at the exact location to access, but merely on the right track and most of the times rotating the platter is required.

The major advantage of hard-disks over RAM memories is that they are non-volatile and thus the information stored is persistent upon power interruptions. However, the non-volatility comes at the price of very slow access times. For example, typical access times are a few tens of nanoseconds for main memories and at least ten milliseconds for hard-disks [111]. As a final remark, the time spent on positioning the head on the correct track is much larger than the time needed to rotate the platter and therefore accessing the data sequentially is much faster than performing random accesses.

2.3 Memory corruptions

Memory devices continually become smaller, work at higher frequencies and lower voltages, and in general have increased circuit complexity [32]. Unfortunately, these improvements come at the cost of reliability [113, 114]. A number of factors, such as alpha particles, infrared radiation, and cosmic rays, can cause *soft memory errors*, sometimes referred as *single-event upsets*, where a bit flips and as a consequence the value stored in the corresponding memory cell is corrupted. An unreliable memory can cause problems in most software ranging from the harmless to the very serious, such as breaking cryptographic

protocols [18, 120], taking control of a Java Virtual Machine [57] or breaking smart-cards and other security processors [5, 6, 108]. Furthermore, many modern computing centers consist of relatively cheap of-the-shelf components, and the large number of individual memories involved in these clusters substantially increase the frequency of memory corruptions in the system. Hence it is crucial that the software running on these machines is robust. Since the amount of cosmic rays increases dramatically with altitude, soft memory errors are of special concern in fields like avionics or space research. Furthermore, soft memory error rates are expected to rise for both DRAM and SRAM memories [113].

At the hardware level, the soft memory errors can be handled by means of error detection mechanisms such as redundancy or error correcting codes. Unfortunately, implementing these mechanisms incur penalties with respect to performance, size and money, and therefore memories using these technologies are rarely found in large scale computing clusters or ordinary workstations. Recent solutions to cope with soft-memory errors in cache memories include replicating some "hot" blocks, i.e. blocks that are accessed often, in cache memories or even an extra fully-associative L1 cache devoted to storing these hot blocks [123, 124].

On the software level, a series of low-level techniques have been proposed for dealing with the soft memory errors, many of them coping with corrupted instructions. Examples include algorithm based fault tolerance [67], assertions [101], control flow checking [121], or procedure duplication [96].

For more details concerning soft memory errors we refer the interested reader to comprehensive surveys [102, 109].

As a final remark, high energy particles originating from alpha particles can cause errors not only in memory locations but also in combinational logic in the processor. In this case the shrinking geometries and increased pipeline depths play a major role in increasing the frequency of combinational logic errors, and studies estimate that between 1992 and 2011 the frequency of these errors will increase nine orders of magnitude up to a level comparing to that of soft memory errors in memories not implementing any error protection [107].

Chapter 3

Modeling Hardware

Make everything as simple as possible, but not simpler.
— Albert Einstein

When counting the occurrences of the different hardware factors (e.g. CPU instructions, branch mispredictions, I/O transfers etc.) during the lifetime of an algorithm, models are employed to provide frameworks in which algorithms are easy to design and analyze. Basically, a model can be seen as a simplification of the targeted hardware by taking into account only the relevant features. Models are required to be simple enough to make algorithm design and analysis as simple as possible, but in the same time they need to be accurate in capturing the hardware they model.

In this chapter we introduce several models used for modeling the different hardware factors affecting the running time of the algorithms, as well as relevant related work for each of them. In Section 3.1 we introduce two popular models used for decades in algorithm design, the *RAM model* and the *comparison model*. In Section 3.2 we describe various approaches employed to achieve branch mispredictions complexities for searching and sorting. We introduce two models handling memory transfers, the *I/O model* and the *cache oblivious model* in Section 3.3, while in Section 3.4 we present several models used to design and analyze streaming algorithms. This chapter concludes with Section 3.5, which is devoted to models related to memory corruptions and computation with unreliable information.

3.1 Traditional models

In the traditional setting, the complexity of algorithms is given by the number of instructions performed by the processor. In this section we introduce two such models, the RAM model in Section 3.1.1 and the comparison model in Section 3.1.2. The RAM model counts all the instructions performed by the processor, whereas the comparison model is more restrictive by considering only comparisons and disregarding other instructions.

3.1.1 RAM model

The RAM model has been employed for decades in the design and analysis of algorithms. It is a simple model which consists of a finite program operating over a memory consisting of an infinite sequence of registers [3, 33]. Also, input and output devices are provided. Each register is identified by an address, which is required to access the contents of any given register. Depending on the set of instructions supported, there have been proposed several RAMs. In the standard model, the instructions supported include assignment, addition and subtraction, indirect addressing, branching, as well as read and write instructions. Other RAMs include SRAM (Successor RAM), where the only arithmetic instruction supported is increment by one, MRAM which is RAM augmented with multiplication and division, and RAMs supporting bit-wise boolean instructions. For a detailed description of the various RAMs and their properties we refer the interested reader to [116].

To measure the time and space complexities, there can be employed at least two methods, the *logarithmic method* and the *uniform method* [116]. In the logarithmic method, the cost of an instruction is given by the sum of the logarithms of the quantities involved, including the operands and the addresses involved. In the uniform method, each instruction is considered to take the same amount of time, regardless of how large the values involved are.

3.1.2 Comparison model

The comparison model can be seen as a restrictive version of the RAM model, since in this model we are interested only in the number of comparisons performed by the algorithm, disregarding other instructions or memory accesses. This model is particularly useful for analyzing algorithms such as sorting and searching for which comparisons are the key instructions. A text-book result concerns sorting algorithms that decide the sorted order based exclusively on comparisons between input elements, and states that such algorithms perform $\Omega(n \log n)$ comparisons in the worst case [34, Section 8.1].

3.2 Branch mispredictions

Several approaches have been proposed for analyzing the branch misprediction complexity of algorithms. A very convenient way to count branch mispredictions is to employ static prediction schemes, where for each branch we specify the direction in which it will be predicted. This approach is particularly effective in the cases where branches outcomes are not correlated and do not contain patterns, and therefore dynamic prediction doesn't help much. For instance, in the case of sorting the outcome of a branch comparing elements is usually very hard to predict because it is strongly dependent on the input. A similar argument holds for searching. Basically, to count the branch mispredictions, we use a variant of the comparison model where for each branch instructions we assign prior to the execution the direction in which the given branch is predicted at all times during the lifetime of the algorithm. Our work in this

model is presented in Chapters 5, 6, and 7. In Chapter 5 we prove theoretically that the number of element swaps performed by Quicksort is related to the number of inversions in the input. We then demonstrate experimentally that the number of element swaps are closely correlated with the number of branch mispredictions, and that the running time is greatly influenced by the number of branch mispredictions. In Chapter 7 we analyze static search trees. A textbook result states that a search tree has to be perfectly balanced to achieve the best running time for a random query. However, in perfectly balanced search trees in the case of a random query branching left or right is (almost) equally likely to happen at any node visited, and therefore branch prediction becomes very hard. To overcome this problem, we analyze skewed binary search trees, where at each node we set a fixed ratio between the nodes in the left and right subtrees. This way, branching in the smaller subtree becomes less likely, and the probability of accurate branch prediction increases as the tree becomes more skewed, at the expense of more nodes to visit, which involves more comparisons and more cache misses. We demonstrate experimentally that skewed search trees can outperform perfectly binary search trees, the improvements observed in the running time being of 15%. Finally, in Chapter 6 we first give lower bound-tradeoffs between branch mispredictions and comparisons for comparison based sorting and adaptive sorting algorithms, when each comparison corresponds to a branch that can be correctly predicted or mispredicted. We employ the classical decision tree model annotated with labels for the edges, such that for each node one of the edges to the children is assigned a label implying that the corresponding branch is correctly predicted, and the other one is assigned a label meaning that a branch misprediction occurred. We give algorithms matching the lower bounds for sorting and sorting that is adaptive to the number of inversions in the input.

Kaligosi and Sanders [72] studied the behavior of Quicksort with a skewed pivot. This means that during partitioning the input the pivot is chosen such that the sizes of the resulted subsequences are deliberately uneven. This way, the algorithm performs more comparisons, but less branch mispredictions since comparisons against a skewed pivot are easier to predict. They employ a dynamic prediction scheme to analyze the number of branch mispredictions performed and complement their theoretical results with experimental results.

3.3 External memory models

External memory models are motivated by the fact that traditional models are not always adequate in practice, due to the memory hierarchy found on modern computers. In particular, when the input does not fit in the main memory the I/O transfers between hard-disk and the main memory often become a bottleneck for the running time, since the cost of a disk transfer is counted in millions of CPU cycles. Several models have been proposed to capture the effect of memory hierarchies. In this section we describe two popular models to capture the disk transfers, the *I/O model* and the *cache-oblivious model*.

3.3.1 I/O model

The I/O model was introduced by Aggarwal and Vitter [1]. Motivated by the fact that the disk transfers are the most time consuming, it consists of a simple two-level memory hierarchy containing a fast memory of size M and a slow infinite memory, see e.g. Figure 3.1. The processor can access only the data

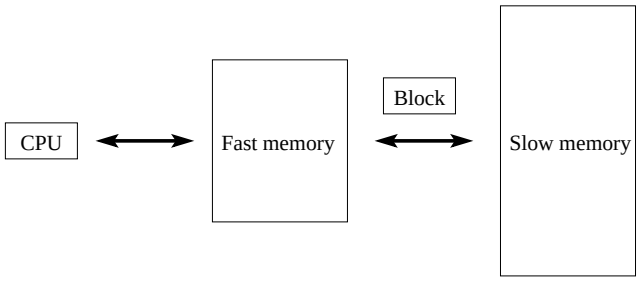

Figure 3.1: The I/O model

in the fast memory, and the data transfers between the slow and fast memories are performed in blocks of size B of consecutive data. Also, in this model algorithms have full control of the memory management, meaning it is the responsibility of the algorithm to decide the memory location for each piece of data as well as the replacement policy. The I/O complexity of an algorithm is given by number of transfers between the slow and the fast memories. A comprehensive list of I/O efficient algorithms for different problems have been proposed, e.g. see the surveys by Vitter [117] and Arge [8]. Among the fundamental results concerning the I/O model is that sorting a sequence of size N requires $\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os [1].

3.3.2 Cache-oblivious model

The I/O model assumes that the size M of the fast memory and the block size B are known, which does not always hold in practice. Another limitation of the I/O model is that it considers only a two-level memory hierarchy, whereas modern computers have multiple memory levels with different sizes and block sizes, see e.g. Figure 3.2. Therefore, an algorithm that behaves well on the whole memory hierarchy would have to consider all the parameters, e.g. block size and memory size, for all the memory levels.

To overcome these limitations, Frigo *et al.* [52] proposed the *cache-oblivious model*. It is similar to the I/O model, but assumes no prior knowledge about M and B. Basically, a cache-oblivious algorithm is an algorithm designed in the I/O model, which does not use M and B in its description, and therefore its analysis hold for any values of M and B. Unlike the I/O model, in the cache-oblivious model memory management is oblivious to the algorithm and an optimal memory replacement strategy is assumed. The power of this model is that if a cache-oblivious algorithm performs well on a two-level memory hierarchy with arbitrary parameters, it performs well between all the consecutive levels of a multi-level memory hierarchy.

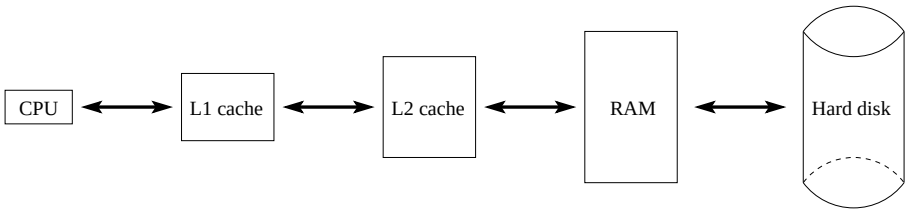

Figure 3.2: Typical memory hierarchy in a modern computer.

Many problems have been addressed in the cache-oblivious model (see e.g. the surveys by Arge *et al.* [10], Brodal [21], and Demaine [35]). Among these there are several optimal cache-oblivious sorting algorithms. Frigo *et al.* [52] gave two optimal cache-oblivious algorithms for sorting: *Funnelsort* and a variant of *Distributionsort*. Brodal and Fagerberg [22] introduced a simplified version of Funnelsort, *Lazy Funnelsort*. All these sorting algorithms are optimal and their I/O complexity is $\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$. Also, all these algorithms require a *tall cache* assumption, i.e. $M = \Omega(B^{1+\varepsilon})$ for a constant $\varepsilon > 0$. In [23] it is shown that a tall cache-assumption is required for all optimal cache-oblivious sorting algorithms.

3.4 Streaming models

Streaming algorithms are algorithms designed to access their input data sequentially. They usually have at their disposal a working memory which is typically much smaller than the size of the input, and it can be used to store information which can be accessed at any time during the execution of the algorithm at no cost. Relevant parameters in streaming models include the number p of passes over the input and the size s of the working memory, usually counted in bits. For a comprehensive survey of algorithmic techniques for processing data streams, we refer the interested reader to the extensive bibliographies in [13,88].

The motivation for developing streaming algorithms is two-fold. First, streaming algorithms are employed in a variety of applications where data is to be accessed "on the fly", such as sensor networks, IP traffic monitoring, transaction logs [55, 56, 110]. Secondly, streaming algorithms are particularly useful in applications that process much more data than the size of the internal memory. This is because modern hard-disks have high transfer rates in the case of sequential access and thus in certain cases streaming algorithms become an efficient alternative to external memory algorithms.

In this section we describe three streaming models. We start by introducing the classical streaming, then we discuss the W-Stream model and we conclude by presenting the StrSort model.

3.4.1 Classical streaming

In the classical read-only streaming model, algorithms are constrained to access the input data sequentially in one (or few) passes, using only a small amount of working memory, typically much smaller than the input size [62,87,88]. Among the problems that have been studied in this model under the restriction that $p = O(1)$, we recall statistics and data sketching problems (see, e.g., [4,48,54]), which can be typically approximated using poly-logarithmic working space, and graph problems (see, e.g., [14,46,47,84]), most of which require a working space linear in the vertex set size.

3.4.2 W-Stream

In the W-Stream model, at each pass we operate with an input stream and an output stream. The streams are pipelined in such a way that the output stream produced at pass i is given as input stream at pass $i + 1$. Despite the use of intermediate streams, which allows achieving effective space-passes trade-offs for fundamental graph problems, most classical lower bounds in read-only streaming hold also in this model [37]. In the original paper, Demetrescu *et al.* [37] introduced the W-Stream model and gave algorithms that achieve efficient trade-offs between space and passes for computing the connected components on undirected graphs and single-source shortest paths in directed graphs with small non-negative integer edge weights. In Chapter 8 we investigate how to turn efficiently parallel algorithms into W-Stream algorithms. We employ a relaxed version the classical PRAM model, which we denote RPRAM, and where a processor is allowed to read any number of cells in a parallel round. This is motivated by the fact that when simulating parallel algorithms in W-Stream we store the state of a processor in the working memory and the information contained in its registers can be processed against an arbitrary number of items in the stream in a single pass. By using simulations, we show how to obtain optimal algorithms (up to poly-log factors) in the W-Stream model for several combinatorial problems, such as sorting, connected components, minimum spanning tree, biconnected components, set maximal independent set. For a series of these algorithms we also give more efficient algorithms designed directly in *wstr*, without using any simulations. We conclude by discussing the limitations of our approach.

3.4.3 StrSort

The StrSort model was introduced by Aggarwal *et al.* [2] and is just W-Stream augmented with a sorting primitive that can be used at each pass to reorder the output stream for free. Sorting provides a lot of computational power and allows solving several graph problems using poly-log passes and working space [2].

3.5 Faulty-memory RAM

Dealing with unreliable information has been addressed in the algorithmic community in a number of settings. The liar model focuses on algorithms in the

comparison model where the outcome of a comparison is possibly a lie. Several fundamental algorithms in this model, such as sorting and searching, have been proposed [19, 78, 99]. In particular, searching in a sorting sequence takes $O(\log n)$ time, even when the number of lies is proportional to the number of comparisons [19]. A standard technique used in the design of algorithms in the liar model is query replication. Unfortunately, this technique is not of much help when memory cells, and not comparisons, are unreliable.

Aumann and Bender [12] proposed fault-tolerant (pointer-based) data structures. To incur minimum overhead, their approach allows a certain amount of data, expressed as a function of the number of corruptions, to be lost upon pointer corruptions. In their framework memory faults are detectable upon access, i.e. trying to access a faulty pointer results in an error message. This model is not always appropriate, since in many practical applications the loss of valid data is not permitted. Furthermore, a pointer can get corrupted to a valid address and therefore an error message is not issued upon accessing it.

Kutten and Peleg [76, 77] introduced the concept of *fault local mending* in the context of distributed networks. A problem is fault locally mendable if there exists a correction algorithm whose running time depends only on the (unknown) number of faults. Some other works studying network fault tolerance include [38, 53, 59, 60, 71, 79, 92].

Finocchi and Italiano [51] introduced the *faulty-memory RAM*. In this model memory corruptions can occur at any time and at any place during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. Motivated by the fact that registers in the processor are considered uncorruptible, $O(1)$ safe memory locations are provided. The model is parametrized by an upper bound, δ, on the number of corruptions occurring during the lifetime of an algorithm. Finally, moving values is considered an atomic operation, i.e. elements do not get corrupted while being copied. An algorithm is resilient if it works correctly, at least on the set of uncorrupted cells in the input. In particular, a resilient searching algorithm returns a positive answer if there exists an uncorrupted element in the input equal to the search key. If there is no element, corrupted or uncorrupted, matching the search key, the algorithm returns a negative answer. If there is a corrupted value equal to the search key, the answer can be both positive and negative.

Several problems have been addressed in the faulty-memory RAM. In the original paper [51], lower bounds and (non-optimal) algorithms for sorting and searching were given. In particular, sorting takes $O(n \log n + \delta^2)$ time and searching in a sorted array takes $\Omega(\log n + \delta)$ time. Matching upper bounds for sorting and randomized searching, as well as a $O(\log n + \delta^{1+\varepsilon})$ deterministic searching algorithm, were then given in [49]. More recently, resilient search trees that support searches, insertions, and deletions in $O(\log n + \delta^2)$ amortized time [50] were introduced. Finally, in [94] it was empirically shown that resilient sorting algorithms are of practical interest.

Our contributions include developing priority queues supporting both insertions and delete-min operations in $O(\log n + \delta)$ amortized time [70], as well as proposing optimal static and dynamic dictionaries supporting searches in $O(\log n + \delta)$ worst case time and updates in $O(\log + \delta)$ amortized time [27].

Independently to our work, Finocchi, Italiano, and Grandoni proposed randomized search trees with $O(\log n + \delta)$ expected amortized cost per operation and a deterministic search tree with $O(\log n + \delta^{1+\varepsilon})$ worst case time per operation. They also prove lower bounds stating that resilient search trees support searches in $\Omega(\log n + \delta)$ worst-case time, under some reasonable assumptions. Part of these results and part of the results in [27] are published in [24].

Chapter 4

Cache-Aware and Cache-Oblivious Adaptive Sorting

The paper is interesting and uses cute reductions.
— Anonymous reviewer

In this chapter we introduce the results in [28]. In Section 4.1 we apply the lower bound technique from [11] to obtain lower bounds on the number of I/Os for comparison based sorting algorithms that are adaptive with respect to different measures of presortedness. In Section 4.2 we present a linear time reduction from adaptive sorting to general (non-adaptive) sorting, directly implying comparison optimal and I/O-optimal cache-aware and cache-oblivious algorithms with respect to measure *Inv*. In Section 4.3 we describe a cache-aware generic sorting algorithm, *cache-aware GenericSort* based on *GenericSort*, introduced in [44], and characterize its I/O adaptiveness. Section 4.4 introduces a cache-oblivious version of GenericSort. In Section 4.5 we introduce a new greedy division protocol for GenericSort, interesting in its own right due to its simplicity. We prove that the resulting algorithm, *GreedySort*, is comparison optimal with respect to measure *Inv*. We show that using our division protocol we obtain both cache-aware and cache-oblivious algorithms that are optimal with respect to *Inv*. In the remainder of this chapter, sorted means sorted in increasing order.

4.1 I/O lower bounds

In this section we show lower bounds on the number of I/Os performed by comparison based sorting algorithms that are adaptive with respect to several measures of presortedness.

Theorem 4.1 *A comparison based sorting algorithm must perform at least* $\Omega(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\frac{Inv}{N})))$ *I/Os for sorting input sequences of size* N *and* Inv *inversions, assuming* $M=\Omega(B^2)$.

Proof. Consider an adaptive sorting algorithm A and some input sequence X of size N. Let $T_A(X)$ and $I/O_A(X)$ denote the number of comparisons and

Measure of presortedness	I/Os	Comparisons [45]
Dis	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Dis)\right)\right)$	$\Omega(N(1+\log(1+Dis)))$
Exc	$\Omega\left(\frac{N}{B}\left(1+Exc\log_{\frac{M}{B}}(1+Exc)\right)\right)$	$\Omega(N+Exc\log(1+Exc))$
Enc	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Enc)\right)\right)$	$\Omega(N(1+\log(1+Enc)))$
Inv	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}\left(1+\frac{Inv}{N}\right)\right)\right)$	$\Omega\left(N\left(1+\log\left(1+\frac{Inv}{N}\right)\right)\right)$
Max	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Max)\right)\right)$	$\Omega(N(1+\log(1+Max)))$
Osc	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}\left(1+\frac{Osc}{N}\right)\right)\right)$	$\Omega\left(N\left(1+\log\left(1+\frac{Osc}{N}\right)\right)\right)$
Reg	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Reg)\right)\right)$	$\Omega(N(1+\log(1+Reg)))$
Rem	$\Omega\left(\frac{N}{B}\left(1+Rem\log_{\frac{M}{B}}(1+Rem)\right)\right)$	$\Omega(N+Rem\log(1+Rem))$
Runs	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+Runs)\right)\right)$	$\Omega(N(1+\log(1+Runs)))$
SMS	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+SMS)\right)\right)$	$\Omega(N(1+\log(1+SMS)))$
SUS	$\Omega\left(\frac{N}{B}\left(1+\log_{\frac{M}{B}}(1+SUS)\right)\right)$	$\Omega(N(1+\log(1+SUS)))$

Figure 4.1: Lower bounds on the number of I/Os and the number of comparisons.

the number of I/Os performed by a comparison based sorting algorithm A for sorting an input sequence X respectively.

Recall that below(X, Inv) denotes the set of all permutations Y for the input sequence with $Inv(Y) \leq Inv(X)$. Consider the decision tree of A (see e.g. [34, Section 8.1]) restricted to the inputs in below(X, Inv). The tree has at least $|\text{below}(X, Inv)|$ leaves and therefore A performs at least $\log|\text{below}(X, \mathcal{D})|$ comparisons in the worst case. Therefore, for any sequence X, there is a sequence $Y \in \text{below}(X, Inv)$, such that $\log|\text{below}(X, Inv)| \leq T_A(Y)$.

Using the decision tree translation by Arge *et al.* [11, Theorem 1] we get:

$$\log(|\text{below}(X, Inv)|) \leq N\log B + \max_{Y\in\text{below}(X,Inv)} I/O_A(Y)\left(B\log\left(\frac{M}{B}\right)+3B\right).$$

Since $\log(|\text{below}(X, Inv)|) = \Omega(N(1+\log(1+\frac{Inv}{N})))$ [58], we obtain that $\max_{Y\in\text{below}(X,Inv)} I/O_A(Y) = \Omega(\frac{N}{B}(1+\log_{\frac{M}{B}}(1+\frac{Inv}{N})))$, given $M = \Omega(B^2)$. □

Using a similar technique we obtain lower bounds on the number of I/Os for other measures of presortedness, assuming that $M = \Omega(B^2)$. Figure 4.1 lists these lower bounds. For definitions of the different measures, refer to [45].

4.2 GroupSort

In this section we describe a reduction to derive *Inv* adaptive sorting algorithms from non-adaptive sorting algorithms. The reduction is cache-oblivious and requires $O(N)$ comparisons and $O(N/B)$ I/Os.

The basic idea is to distribute the input sequence into a sequence of buckets $S_1, \ldots, S_k$ each of size at most $32(Inv/N)^2$, where the elements in bucket S_i are all smaller than or equal to the elements in S_{i+1}. Each S_i is then sorted independently by a non-adaptive cache-oblivious sorting algorithm [22, 52]. During the construction of the buckets $S_1, \ldots, S_k$ some elements might fail to get inserted into an S_i and are instead inserted into a *fail set* F. It will be guaranteed that at most half of the elements are inserted into F. The fail set F is sorted recursively and merged with the sequence of sorted buckets.

The S_i buckets are constructed by scanning the input left-to-right by inserting an element x into the rightmost bucket S_k if $k = 1$ or $x \geq \min(S_k)$ and otherwise inserting x in F. During the construction we generate increasing bucket capacities $\beta_j = 2 \cdot 4^j$, which will be used for $\alpha_j = N/(2 \cdot 2^j)$ insertions into F. If during construction $|S_k| > \beta_j$, the bucket S_k is split into two buckets S_k and S_{k+1} by computing its median using the cache-oblivious selection algorithm from [16] and distributing its elements relatively to the median. This ensures $|S_i| \leq \beta_j$ for $1 \leq i \leq k$. We maintain the invariant $|S_k| \geq \beta_j/2$ if there are at least two buckets by repeatedly concatenating the two last buckets after an increment of i. Since $\beta_{j-1} = \beta_j/4$, this ensures $\beta_j/2 \leq |S_k| \leq \frac{3}{4}\beta_j$ after this concatenation process. If only one bucket remains, then $|S_k| \leq \frac{3}{4}\beta_j$.

The pseudo-code of the reduction is given in Figure 4.2. We assume that $S_1, \ldots, S_k$ are stored consecutively in an array by storing the start index and the minimum element from each bucket on a separate stack, i.e. the concatenation of S_{k-1} and S_k can be done implicitly in $O(1)$ time. The fail set F is stored as a list of subsets $F_1, \ldots, F_j$, where F_i stores the elements inserted into F while the bucket size is β_i. Similarly $F_1, \ldots, F_j$ are stored consecutively in an array.

Theorem 4.2 GroupSort *is cache-oblivious and is comparison optimal and I/O-optimal with respect to Inv, assuming* $M = \Omega(B^2)$.

Proof. Consider the last bucket capacity β_j and fail set size α_j. Each element x inserted into the fail set F_j induces in the input sequence at least $\beta_j/2$ inversions, since $|S_k| \geq \beta_j/2$ when x is inserted into F_j and all elements in S_k appeared before x in the input and are larger than x.

For $i = \lceil \log \frac{Inv}{N} \rceil + 1$, we have $\alpha_i \cdot \frac{\beta_i}{2} = \frac{N}{2 \cdot 2^i} \cdot \frac{2 \cdot 4^i}{2} \geq Inv$, i.e. F_i is guaranteed to be able to store all failed elements. This immediately leads to $j \leq \lceil \log \frac{Inv}{N} \rceil + 1$, and $\beta_j = 2 \cdot 4^j \leq 32 \left(\frac{Inv}{N}\right)^2$. The fail set F has size at most $\sum_{i=1}^{j} \alpha_i = \sum_{i=1}^{j} N/(2 \cdot 2^i) \leq N/2$.

Taking into account that the total size of the fail sets is at most $N/2$, the number of comparisons performed by GroupSort is given by the following recurrence:

$$T(N) = T\left(\frac{N}{2}\right) + \sum_{i=1}^{k} T_{\text{Sort}}(|S_i|) + O(N) \ ,$$

where the $O(N)$ term accounts for the bucket splittings and the final merge of S and F. The $O(N)$ term for splitting buckets follows from that when a bucket with β_j elements is split then at least $\beta_j/4$ elements in a bucket have been inserted since the most recent bucket splitting or increase in bucket capacity, and we can charge the splitting of the bucket to these recent $\beta_j/4$ elements.

```
procedure GroupSort(X)
Input: Sequence X = (x_1, ..., x_N)
Output: Sequence X sorted
begin
    S_1 = (x_1); F_1 = (); β_1 = 8; α_1 = N/4; j = 1; k = 1;
    for i = 2 to N
        if k = 1 or x_i ≥ min(S_k)
            append(S_k, x_i);
            if |S_k| > β_j
                (S_k, S_{k+1}) = split(S_k); k = k + 1;
        else
            append(F_j, x_i);
            if |F_j| > α_j
                β_{j+1} = β_j · 4; α_{j+1} = α_j/2; j = j + 1;
                while k > 1 and |S_k| < β_j/2
                    S_{k-1} = concat(S_{k-1}, S_k); k = k − 1;
    S = concat(sort(S_1), sort(S_2), ..., sort(S_k));
    F = concat(F_1, F_2, ..., F_j);
    GroupSort(F);
    X = merge(S, F);
end
```

Figure 4.2: Linear time reduction to non-adaptive sorting.

Since $T_{\text{Sort}}(N) = O(N \log N)$ and each $|S_i| \leq \beta_j = O((\frac{Inv}{N})^2)$ the number of comparisons performed by GroupSort is:

$$T(N) = T\left(\frac{N}{2}\right) + O\left(N\left(1 + \log\left(1 + \left(\frac{Inv}{N}\right)^2\right)\right)\right) .$$

Since F is a subsequence of the input, Inv for the recursive call is at most Inv for the input. As $\sum_{i=0}^{\infty} \frac{N}{2^i} \log \frac{Inv}{N/2^i} = N \log \frac{Inv}{N} \sum_{i=0}^{\infty} \frac{1}{2^i} + N \sum_{i=0}^{\infty} \frac{i}{2^i}$, it follows that GroupSort performs $T(N) = O\left(N\left(1 + \log\left(1 + \frac{Inv}{N}\right)\right)\right)$ comparisons, which is optimal.

The cache-oblivious selection algorithm from [16] performs $O(N/B)$ I/Os and the cache-oblivious sorting algorithms [22, 52] perform $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os for $M = \Omega(B^2)$. Since GroupSort otherwise does sequential access to the input and data structures, we get that GroupSort is cache-oblivious and the number of I/Os performed is given by the recurrence:

$$I/O(N) = I/O\left(\frac{N}{2}\right) + O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}}\left(1 + \left(\frac{Inv}{N}\right)^2 \cdot \frac{1}{B}\right)\right)\right) .$$

It follows that GroupSort performs $O(\frac{N}{B}(1 + \log_{\frac{M}{B}}(1 + \frac{Inv}{N})))$ I/Os provided $M = \Omega(B^2)$, which by Theorem 4.1 is I/O-optimal. □

Pagh *et al.* [90] gave a related reduction for adaptive sorting on the RAM model. Their reduction assumes that a parameter q is provided such that the

number of inversions is at most qN. A valid q is found by selecting increasing values for q such that the running time doubles for each iteration. In the cache oblivious setting the doubling approach fails, since the first q value should depend on the unknown parameter M. We circumvent this limitation of the doubling technique by selecting the increasing β_j values internally in the reduction.

4.3 Cache-aware GenericSort

Estivill-Castro and Wood [44] introduced a generic sorting algorithm, *GenericSort*, as a framework for adaptive sorting algorithms. It is a generalization of Mergesort, and is described using a generic division protocol, i.e. an algorithm for splitting an input sequence into two or more subsequences. The algorithm works as follows: consider an input sequence X; if X is sorted then the algorithm returns; if X is "small", then X is sorted using some alternate non-adaptive sorting algorithm; otherwise, X is divided according to the division protocol and the resulting subsequences are recursively sorted and merged.

In this section we modify GenericSort to achieve a generic I/O-adaptive sorting algorithm. Consider an input sequence $X = (x_1, \ldots, x_N)$ and some division protocol DP such that DP splits the input in $s \geq 2$ subsequences of roughly equal sizes in a single scan, visiting each element of the input exactly once. To avoid testing whether X is sorted before applying the division protocol, we derive a new division protocol DP' by modifying DP to identify the longest sorted prefix of X: we scan the input sequence until we find some i such that $x_i < x_{i-1}$. Denote $S = (x_1, \ldots, x_{i-1})$ and $X' = (x_i, \ldots, x_N)$. We apply DP to X', recursively sort the resulting s subsequences, and finally merge them with S. The adaptive bounds for GenericSort proved in [44, Theorem 3.1] are not affected by these modifications, and we have the following theorem.

Theorem 4.3 *Let $\mathcal{D}$ be a measure of presortedness, d and s constants, $0 < d < 2$, and DP a division protocol that splits some input sequence of size N into s subsequences of size at most $\lceil \frac{N}{s} \rceil$ each using $O(N)$ comparisons.*

- *the modified GenericSort performs $O(N \log N)$ comparisons in the worst case;*
- *if for all sequences X, the division of a suffix of X into $X_1, \ldots, X_s$ by DP satisfies that $\sum_{j=1}^{s} \mathcal{D}(X_j) \leq d\lfloor \frac{s}{2} \rfloor \cdot \mathcal{D}(X)$, then the modified GenericSort performs $O\left(N\left(1 + \log(1 + \mathcal{D}(X))\right)\right)$ comparisons.*

We now describe a cache-aware version of the modified GenericSort provided that the division protocol DP works in a single scan of the input. Let T be the recursion tree of GenericSort using the new division protocol DP'. We obtain a new tree T' by contracting T top-down such that every node in T' corresponds to a subtree of height $O(\log_s(M/B))$ in T and each node in T' has a fanout of at most m, where $m = \Theta(M/B)$. There are $O(m)$ sorted prefixes for every node in T'. In cache-aware GenericSort, for each node of T' we scan its input

sequence and distribute the elements accordingly to one of the $O(m)$ output sequences. Each output sequence is a linked list of blocks of size $\Theta(B)$. If the size of the input sequence is at most M, then we sort it in internal memory, hence performing $O(N/B)$ I/Os. Theorem 4.4 gives a characterization of the adaptiveness of cache-aware GenericSort in the I/O model. It is an I/O version of Theorem 4.3.

Theorem 4.4 *Let $\mathcal{D}$ be a measure of presortedness, d and s constants, $0 < d < 2$ and $s \leq \frac{M}{2B}$, and DP a division protocol that splits some input sequence of size N into s subsequences of size at most $\lceil \frac{N}{s} \rceil$ each using $O(\frac{N}{B})$ I/Os. If DP performs the splitting in one scan visiting each element of the input exactly once, then:*

- *cache-aware GenericSort performs $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os in the worst case;*
- *if for all sequences X, the division of a suffix of X into $X_1, \ldots, X_s$ by DP satisfies that $\sum_{j=1}^{s} \mathcal{D}(X_j) \leq d\lfloor \frac{s}{2} \rfloor \cdot \mathcal{D}(X)$, then cache-aware GenericSort performs $O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}}(1 + \mathcal{D}(X))\right)\right)$ I/Os.*

Proof. We analyze the I/Os performed at the nodes of T' separately for the nodes having input sizes less than or equal to M and greater than M.

At a node with input X and $|X| > M$, $O(m + |X|/B) = O(|X|/B)$ I/Os are performed to read the input and to write to the at most $m - 1$ sorted output prefixes and m sequences to be recursively sorted. If we charge $O(1/B)$ I/Os per element in the input this will pay for the I/Os required at the node.

At a node with input X and $|X| \leq M$, $O(1 + |X|/B)$ I/Os are performed. These I/Os can be charged to the parent node, since at the parent we will already charge $O(1 + |X|/B)$ I/Os to write the output X.

By Theorem 4.3 we have that the sum of the depths in T reached by the elements in the input X is bounded by $O(N(1 + \log(1 + \mathcal{D}(X))))$. Since each node in T' spans $\Theta(\log \frac{M}{B})$ levels from T, we get that cache-aware GenericSort performs $O(\frac{N}{B} + N(1 + \log(1 + \mathcal{D}(X)))/(B \log \frac{M}{B})) = O(\frac{N}{B}(1 + \log_{\frac{M}{B}}(1 + \mathcal{D}(X))))$ I/Os, where the N/B term counts for the I/Os at the root of T'. □

The power of cache-aware GenericSort lies in its generality, meaning that using different division protocols we obtain sorting algorithms that are I/O adaptive with respect to different measures of presortedness. For example, using the straight division protocol, we achieve I/O optimality with respect to *Runs*. Using the odd-even division protocol, we obtain an algorithm that is I/O optimal with respect to *Dis* and *Max*. Furthermore, the different division protocols can be combined as shown in [45] in order to achieve I/O optimality with respect to more measures of presortedness.

4.4 Cache-oblivious GenericSort

We give a cache-oblivious algorithm that achieves the same adaptive bounds as the cache-aware GenericSort introduced in Section 4.3. It works only for

division protocols that split the input into two unsorted subsequences. It is based on a modification of the k-merger used in FunnelSort [22, 52].

A k-merger is a binary tree stored using the recursive van Emde Boas layout. The edges contain buffers of variable sizes and the nodes are binary mergers. The tree and the buffer sizes are recursively defined: consider an output sequence of size k^3 and h the height of the tree. We split the tree at level $\frac{h}{2}$ yielding $k^{\frac{1}{2}} + 1$ subtrees, each of size $O(k^{\frac{1}{2}})$. The buffers at this level have sizes $k^{\frac{3}{2}}$. See [22] for further details.

Consider DP division protocol that scans the input a single time and DP' the modified DP as introduced in Section 4.3. Each node of the k-merger corresponds to a node in the recursion tree of GenericSort using DP' as the division protocol. Therefore, each node has a fanout of three and becomes a ternary merger. The resulting unsorted sequences are pushed in the buffers to the children, while the sorted prefix is stored as a list of memory chunks of size $O(N^{\frac{2}{3}})$ for an input buffer of size N.

Our algorithm uses a single $N^{\frac{1}{3}}$-merger. It fills the buffers in a top-down fashion and then merges the resulted sorted subsequences in a bottom-up manner. The $N^{\frac{1}{3}}$ output buffers at the leaves of the k-merger are sorted using a non-adaptive I/O-optimal cache oblivious sorting algorithm [22, 52].

Lemma 4.1 *The $N^{\frac{1}{3}}$-merger and the sorted subsequences use $O(N)$ space.*

Proof. Consider the $N^{\frac{1}{3}}$-merger and an input sequence of size N. The total size of the inner buffers is $O(N^{\frac{2}{3}})$ [52]. The memory chunks storing the sorted subsequences use $O(N)$ space because there are $N^{\frac{1}{3}}$ nodes in the merger and the size of a single memory chunk is $O(N^{\frac{2}{3}})$. Adding the input sequence, we conclude that the $N^{\frac{1}{3}}$-merger and the sorted subsequences take $O(N)$ space together. □

Lemma 4.2 *Cache-oblivious GenericSort and cache-aware GenericSort have the same comparison and I/O complexity, for division protocols that split the input into two subsequences.*

Proof. Consider $\ell = \frac{1}{3} \log N$ the height of the $N^{\frac{1}{3}}$-merger of the cache-oblivious GenericSort.

We first prove that cache-aware and cache-oblivious GenericSort have the same comparison complexity. For some element x_i let d_i be its depth in the recursion tree of the GenericSort using DP' as a division protocol. If $d_i \leq \ell$ then x_i reaches the same level in the recursion tree of cache-oblivious GenericSort, because the two algorithms have the same recursion trees at the top ℓ levels. If $d_i > \ell$ then the number of comparisons performed by cache-oblivious GenericSort for x_i is $O(\log N) = O(d_i)$ because $d_i > l = \Omega(\log N)$.

We analyze the number of I/Os used by cache-aware and cache-oblivious GenericSort. Consider an element x_i that reaches level d_i in the recursion tree of cache-aware GenericSort.

If $d_i < \ell$ then x_i is placed in a sorted prefix at a node in the $N^{\frac{1}{3}}$-merger. In this case, cache-oblivious GenericSort spends linear I/Os when the size of the input reaches $O(M)$ because the $N^{\frac{1}{3}}$-merger together with the sorted subsequences take linear space by Lemma 4.1. Taking into account that the height of the $N^{\frac{1}{3}}$-merger is $O(\log(M/B))$ due to the tall cache assumption, it follows that $O(1 + d_i/(\log(M/B)))$ I/Os are performed by cache-oblivious GenericSort for getting x_i to its sorted subsequence.

If $d_i > \ell$ then x_i reaches an output buffer of the $N^{\frac{1}{3}}$-merger, where it is sorted using an optimal cache-oblivious sorting algorithm. In this case the number of I/Os performed for the sorting involving x_i is still $O(1/B + d_i/(B\log(M/B)))$, because both the $N^{\frac{1}{3}}$-merger and the optimal sorting algorithms require $O(1/B + d_i/(B\log(M/B)))$ I/Os for the sorting involving x_i, since $d_i = \Theta(\log N)$.

We obtain that cache-oblivious GenericSort performs $O\left(\frac{N}{B} + \frac{\sum_{i=1}^{n} d_i}{B\log(M/B)}\right)$ I/Os. Cache-aware GenericSort performs $O\left(\frac{N}{B} + \frac{\sum_{i=1}^{n} d_i}{B\log\frac{M}{B}}\right)$ I/Os too because the fanout of the nodes in the recursion tree is $O(\log\frac{M}{B})$. We conclude that cache-aware GenericSort and cache-oblivious GenericSort have the same I/O complexity. □

4.5 GreedySort

We introduce *GreedySort*, a sorting algorithm based on GenericSort using a new division protocol, *GreedySplit*. The protocol is inspired by a variant of the Kim-Cook division protocol, which was introduced and analyzed in [80]. Our division protocol achieves the same adaptive performance with respect to *Inv*, but is simpler and moreover facilitates cache-aware and cache-oblivious versions. It may be viewed as being of a greedy type, hence the name. We first describe GreedySort and its division protocol and then prove that it is optimal with respect to *Inv*. GreedySplit partitions the input sequence X into three subsequences S, Y, and Z, where S is sorted and Y and Z have balanced sizes, i.e. $|Z| \leq |Y| \leq |Z| + 1$. In one scan it builds an ascending subsequence S of the input in a greedy fashion and at the same time distributes the remaining elements in two subsequences, Y and Z, using an odd-even approach.

Lemma 4.3 *GreedySplit splits an input sequence X in the three subsequences S, Y and Z, where S is sorted and $Inv(X) \geq \frac{5}{4} \cdot (Inv(Y) + Inv(Z))$.*

Proof. Let $X = (x_1, \ldots x_N)$. By construction S is sorted. Consider an inversion in Y, $y_i > y_j$, $i < j$ and i_1 and j_1 the indices in X of y_i and y_j respectively. Due to the odd-even construction of Y and Z, there exists an $x_k \in Z$ such that in the original sequence X we have $i_1 < k < j_1$.

We prove that there is one inversion between x_k and at least one of x_{i_1} and x_{j_1}, for any $i_1 < k < j_1$. Indeed, if $x_{i_1} > x_k$, we get an inversion between x_{i_1} and x_k. If $x_{i_1} \leq x_k$, we get an inversion between x_{j_1} and x_k, because we

assume that $y_i > y_j$ which yields $x_{i_1} > x_{j_1}$. Let $z_i, \ldots, z_{j-1}$ be all the elements from Z which appear between y_i and y_j in the original sequence. We know that there exists at least an inversion between $z_{\lfloor i+j \rfloor/2}$ and y_i or y_j. The inversion $(y_i, z_{\lfloor (i+j)/2 \rfloor})$ can be counted for two different pairs in Y, $(y_i, y_{i+2\lfloor (j-i)/2 \rfloor})$ and $(y_i, y_{i+1+2\lfloor (j-i)/2 \rfloor})$. Similarly, the inversion $(z_{\lfloor (i+j)/2 \rfloor, j})$ can be counted for two different pairs in Y. Taking into account that the inversions involving elements of Y and elements of Z appear in X, but neither in Y nor Z, we have that $Inv(X) \geq Inv(Y) + Inv(Z) + Inv(Y)/2$. In a similar manner we obtain $Inv(X) \geq Inv(Y) + Inv(Z) + Inv(Z)/2$. Summing the two equations we obtain $Inv(X) \geq \frac{5}{4}(Inv(Y) + Inv(Z))$. □

Theorem 4.5 *GreedySort performs $O(N(1+\log(1+Inv(X)/N)))$ comparisons to sort a sequence X of size N, i.e. it is comparison optimal with respect to Inv.*

Proof. Similar to [80], we first prove the claimed bound for the upper levels of recursion where the total number of inversions is greater than $N/4$ and then prove that the total number of comparisons for the remaining levels is linear. Let $Inv_i(X)$ denote the total number of inversions in the subsequences at the i^{th} level of recursion. By Lemma 4.3, $Inv_i(X) \leq \left(\frac{4}{5}\right)^i Inv(X)$.

We want to find the first level ℓ of the recursion for which $\left(\frac{4}{5}\right)^\ell Inv(X) \leq \frac{N}{4}$, which yields $\ell = \left\lceil \frac{\log(4Inv(X)/N)}{\log(5/4)} \right\rceil$.

At each level of recursion GreedySort performs $O(N)$ comparisons. Therefore at the first ℓ levels of recursion the total number of comparisons performed is $O(\ell \cdot N) = O(N(1+\log(1+Inv(X)/N)))$. We now prove that the remaining levels perform a linear number of comparisons.

Let $|(X,i)|$ denote the total size of Ys and Zs at the ith level of recursion. As each element in Y and Z is obtained as a result of an inversion in the sequence X, we have $|(X,i)| \leq Inv_{i-1}(X)$. Using Lemma 4.3 we obtain: $|(X,\ell+i)| \leq Inv_{\ell+i-1}(X) \leq \left(\frac{4}{5}\right)^{i-1} \cdot \left(\frac{4}{5}\right)^\ell \cdot Inv(X) \leq \left(\frac{4}{5}\right)^{i-1} \frac{N}{4}$. Taking into account that the sum of the $|(X,\ell+i)|$s is $O(N)$ and that at each level $\ell+i$ we perform a linear number of comparisons with respect to $|(X,\ell+i)|$, it follows that the total number of comparisons performed at the lower levels of the recursion tree is $O(N)$. We conclude that GreedySort performs $O(N(1+\log(1+\frac{Inv}{N})))$ comparisons. □ We derive both cache-aware and cache-oblivious algorithms

by using our greedy division protocol in both the cache-aware and the cache-oblivious GenericSort frameworks described in Sections 4.3 and 4.4. In both cases the division protocol considered does not identify the longest prefix of the input, but simply apply the greedy division protocol. We prove that these new algorithms, *cache-aware GreedySort* and *cache-oblivious GreedySort* achieve the I/O-optimality with respect to *Inv* under the tall cache assumption $M = \Omega(B^2)$.

Theorem 4.6 *Both cache-aware GreedySort and cache-oblivious GreedySort are I/O-optimal with respect to Inv, provided that $M = \Omega(B^2)$.*

Proof. From Theorem 4.5 the average number of levels of recursion for an element is $O(1+\log(1+Inv/N))$. In Theorem 4.4 each element is charged $O(\frac{1}{B})$

I/Os for every $\Theta(\log \frac{M}{B})$ levels. This implies that cache-aware GreedySort performs $\Theta(\frac{N}{B}(1 + \log_{\frac{M}{B}}(1 + \frac{Inv}{N})))$ I/Os, which is optimal by Theorem 4.1. Similar observations apply to cache-oblivious GreedySort based on the proof of Lemma 4.2. □

Chapter 5

On the Adaptiveness of Quicksort

Given how well studied quicksort is, it is amazing that this basic fact was not previously known.
— Anonymous reviewer

In this chapter we present the results published in [29]. Our main contribution is demonstrating that the running time of Quicksort is correlated with the number of inversions in the input, but we give results concerning adaptive behavior also for Mergesort and Heapsort. In Section 5.1 we prove that the expected number of element swaps performed by randomized Quicksort depends on the number *Inv* of inversions in the input. In Section 5.2 we describe our experimental setup, and in Section 5.3 we describe and discuss our experimental results. Parts of our proof of Theorem 5.1 were inspired by the proof by Seidel [105, Section 5] concerning the expected number of comparisons performed by randomized Quicksort.

5.1 Expected number of swaps by randomized Quicksort

In this section we analyze the expected number of element swaps performed by the classic version of randomized Quicksort where in each recursive call a random pivot is selected. The C code for the specific algorithm considered is given in Figure 5.1. The parameters `l` and `r` are the first and last element, respectively, of the segment of the array `a` to be sorted.

Theorem 5.1 *The expected number of element swaps performed by randomized Quicksort is at most* $n + n \ln\left(\frac{2Inv}{n} + 1\right)$.

We assume that the n input elements are distinct. In the following, let $(x_1, \ldots, x_n)$ denote the input sequence, and let π_i be the rank of x_i in the sorted sequence. The number of inversions in the input sequence is denoted by *Inv*. The main observation used in the proof of Theorem 5.1 is that an element x_i that has not yet been moved from its input position i is swapped during a partitioning step if and only if the selected pivot x_j satisfies $i \leq \pi_j < \pi_i$ or $\pi_i < \pi_j \leq i$, or x_i is itself the pivot element. This is seen by inspection of the

```
#define Item int
#define random(l,r) (l+rand() % (r-l+1))
#define swap(A, B) { Item t = A; A = B; B = t; }

void quicksort(Item a[], int l, int r)
   { int i;
     if (r <= l) return;
     i = partition(a, l, r);
     quicksort(a, l, i-1);
     quicksort(a, i+1, r);
   }

int partition(Item a[], int l, int r)
  { int i = l-1, j = r+1, p = random(l,r);
    Item v = a[p];
    for (;;) {
        while (++i < j && a[i] <= v);
        while (--j > i && v <= a[j]);
        if (j <= i) break;
        swap(a[i], a[j]);
      }
    if (p < i) i--;
    swap(a[i], a[p]);
    return i;
  }
```

Figure 5.1: C code for randomized Quicksort.

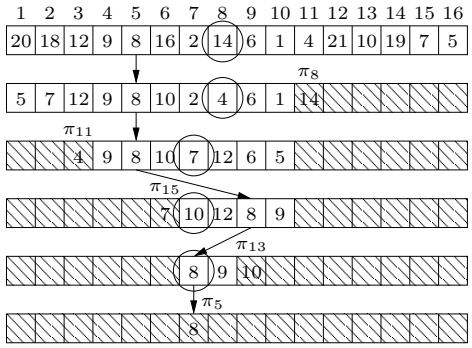

Figure 5.2: The partitions involving element 8.

code, noting that after a partitioning step, the pivot element x_j resides at its final position π_j. We shall only need the "only if" part.

Fact 5.1 *When x_i is swapped the first time, the pivot x_j of the current partitioning step satisfies $i \leq \pi_j < \pi_i$ or $\pi_i < \pi_j \leq i$, or x_i is itself the pivot element.*

Figure 5.2 illustrates how the element $x_5 = 8$ is moved during the execution of randomized Quicksort. Circled elements are the selected pivots. The first two selected pivots 14 and 4 do not cause 8 to be swapped, since 8 is already correctly located with respect to the final positions of of the pivots 14 and 4. The first pivot causing 8 to be swapped is $x_{15} = 7$, since $\pi_5 = 7$, $\pi_{15} = 6$, and $5 \leq \pi_{15} < \pi_5$.

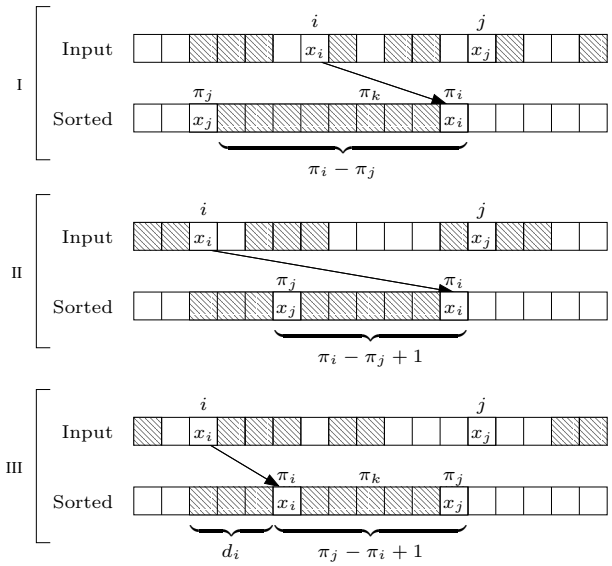

Figure 5.3: The three different cases of Lemma 5.2.

In the succeeding recursive calls after the first swap of an element x_i, the positions of x_i in the array are unrelated to i and π_i. Eventually, x_i is either picked as a pivot or becomes a single element input to a recursive call (the base case is reached), after which x_i does not move further.

In the following we let $d_i = |\pi_i - i|$, i.e. the distance of x_i from its correct position in the sorted output. The correlation between Inv and the d_i values is captured by the following lemma:

Lemma 5.1 $Inv \leq \sum_{i=1}^{n} d_i \leq 2Inv$.

Proof. For the left inequality, $Inv \leq \sum_{i=1}^{n} d_i$, we consider the following algorithm: If there is an element x_i not at its correct position, move x_i to position π_i, such that position π_i temporarily contains both x_i and x_{π_i} in sorted order. Next move x_{π_i} to its correct position, and repeat moving an element from the position temporarily containing two elements to its correct position, until we move an element to position i. Repeat until the sequence is sorted. By moving element x_i from position i to its correct position π_i, we move x_i over the $d_i - 1$ elements at positions between i and π_i and possibly the current element at position π_i. This decreases the number of inversions in the sequence by at most d_i, namely any inversions between x_i and each of the at most d_i elements moved over. In the final sorted sequence there are no inversions, hence we have $Inv \leq \sum_{i=1}^{n} d_i$.

For the right inequality, $\sum_{i=1}^{n} d_i \leq 2Inv$, consider some x_i with $\pi_i \geq i$. In the input sequence there are at least d_i inversions between x_i and other input elements, since there are at least d_i elements less than x_i with indices greater than i in the input sequence. A similar argument holds for the case when $\pi_i < i$. Taking into account that we may count the same inversion twice, we obtain $\sum_{i=1}^{n} d_i \leq 2Inv$. □

The constants in Lemma 5.1 are the best possible. For even n, the sequence $(2,1,4,3,6,5,\ldots,n,n-1)$ has $Inv = n/2$ and $\sum_{i=1}^{n} d_i = n$, i.e. $(\sum_{i=1}^{n} d_i)/Inv = 2$, whereas the sequence $(n, n-1, n-2, n-3, \ldots, 3, 2, 1)$ has $Inv = n(n-1)/2$ and $\sum_{i=1}^{n} d_i = n^2/2$, i.e. $(\sum_{i=1}^{n} d_i)/Inv = 1 + \frac{1}{n-1}$ which converges to one for increasing n.

For the proof of Theorem 5.1 we make the following definition:

Definition 5.1 *For $i \neq j$ let X_{ij} denote the indicator variable that is one if and only if there is a recursive call to* `quicksort` *where x_j is selected as the pivot in the partition step and x_i is swapped during this partition step.*

Note that x_j can at most once become a pivot, since after a partition with pivot x_j the inputs to the recursive calls do not contain x_j. Furthermore note that the elements swapped in a partition step with pivot x_j are the elements in the input to the partition which are placed incorrectly relatively to the final position π_j of x_j.

There are three cases where $X_{ij} = 0$: (i) x_j is never selected as a pivot, i.e. there exists a recursive call where x_j is the only element to be sorted; (ii) x_j is selected as a pivot in a recursive call and x_i is not in the input to this recursive call; and (iii) x_j is selected as a pivot in a recursive call and x_i is in the input to this recursive call, but x_i is not swapped because it is placed correctly relatively to the final position π_j of x_j.

Lemma 5.2

$$\Pr[X_{ij} = 1] \leq \begin{cases} 0 & \text{if } \pi_j < i \leq \pi_i \text{ or } \pi_i \leq i < \pi_j\,, \\ \frac{1}{|\pi_i - \pi_j|+1} & \text{if } i \leq \pi_j < \pi_i \text{ or } \pi_i < \pi_j \leq i\,, \\ \frac{1}{|\pi_i - \pi_j|+1} - \frac{1}{|\pi_i - \pi_j|+1+d_i} & \text{otherwise.} \end{cases}$$

Proof. For the case (i) where x_j is never selected as a pivot for a partition, we in the following adopt the convention that x_j is considered the pivot for the recursive call where the input consists of x_j only. This ensures that each element becomes a pivot exactly once.

We first note that the probability that x_i is in the input to the recursive call with pivot x_j is $\frac{1}{|\pi_i - \pi_j|+1}$, since this is the probability that x_j is the first element chosen as a pivot among the $|\pi_i - \pi_j|+1$ elements x_k with $\pi_i \leq \pi_k \leq \pi_j$ or $\pi_j \leq \pi_k \leq \pi_i$ (if the first pivot x_k among the $|\pi_i - \pi_j| + 1$ elements is not x_j, then the selected pivot x_k will cause x_i and x_j to not appear together in any input to succeeding recursive calls).

To prove the lemma we consider the three different cases depending on the relative order of i, π_i, and π_j. In the following we assume $i \leq \pi_i$. The cases where $\pi_i < i$ are symmetric. The three possible scenarios are shown in Figure 5.3.

First consider the case where $\pi_j < i \leq \pi_i$, see Figure 5.3 (I). If a pivot x_k is selected with $\pi_j < \pi_k \leq \pi_i$ before x_j becomes a pivot, then x_i and x_j do not appear together in any input to succeeding recursive calls, so x_i cannot be involved in the partition with pivot x_j. The only other possibility is that x_j is a pivot before any element x_k with $\pi_j < \pi_k \leq \pi_i$ becomes a pivot, but then by

Fact 5.1 x_i has not been moved when x_j becomes a pivot, and the partitioning with pivot x_j does not swap x_i.

For the second case, where $i \leq \pi_j < \pi_i$, see Figure 5.3 (II), we bound the probability that X_{ij} equals one by the probability that x_i is in the input to the recursive call with pivot x_j. As argued above, this probability is $\frac{1}{|\pi_i - \pi_j|+1}$.

For the last case where $i \leq \pi_i < \pi_j$, see Figure 5.3 (III), we consider the probability that x_i is in the input to the recursive call with pivot x_j and x_i is not swapped. This is at least the probability that x_j is the first element chosen as a pivot among the $|\pi_i - \pi_j| + 1 + d_i$ elements x_k with $i \leq \pi_k \leq \pi_j$, since then by Fact 5.1 x_i has not been moved yet when x_j becomes the pivot, and the partitioning with pivot x_j does not swap x_i. It follows that the probability that x_i is in the input to the recursive call with pivot x_j and x_i is not swapped, is at least $\frac{1}{|\pi_i - \pi_j|+1+d_i}$. Since the probability that x_i is in the input to the recursive call with pivot x_j is $\frac{1}{|\pi_i - \pi_j|+1}$, the lemma follows. □

Using Lemma 5.1 and Lemma 5.2 we now have the following proof of Theorem 5.1.

Proof. Theorem 5.1 The `for`-loop in the partitioning procedure in Figure 5.1 only swaps non-pivot elements and each element is swapped at most once in the loop. The loop is followed by one swap involving the pivot. Since a swap of two elements x_i and x_k not involving the pivot x_j are counted by the two indicator variables X_{ij} and X_{kj}, the expected number of swaps is at most

$$
\begin{aligned}
& \mathrm{E}\left[\sum_{j=1}^{n}\left(1+\frac{1}{2}\sum_{i=1,i\neq j}^{n} X_{ij}\right)\right] \\
&= n+\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1,i\neq j}^{n}\Pr(X_{ij}=1) \\
&\leq n+\frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{d_i}\frac{1}{k+1}+\sum_{k=1}^{\infty}\left(\frac{1}{k+1}-\frac{1}{k+1+d_i}\right)\right) && (5.1)\\
&\leq n+\frac{1}{2}\sum_{i=1}^{n}\left(2\sum_{k=1}^{d_i}\frac{1}{k+1}\right) \\
&= \sum_{i=1}^{n}\sum_{k=1}^{d_i+1}\frac{1}{k} \\
&\leq \sum_{i=1}^{n}(1+\ln(d_i+1)) && (5.2)\\
&\leq n+n\ln\frac{\sum_{i=1}^{n}(d_i+1)}{n} && (5.3)\\
&\leq n+n\ln\left(\frac{2Inv}{n}+1\right) && (5.4)
\end{aligned}
$$

where (5.1) follows from Lemma 5.2, (5.2) follows from $\sum_{i=1}^{n}\frac{1}{i} \leq 1+\ln n$, (5.3) follows from the concavity of the logarithm function, and (5.4) follows from

Lemma 5.1. □

It should be noted that the upper bound achieved in (5.3) using the concavity of the logarithm function can be much larger than the value (5.2). As an example, if there are $\Theta(n/\log n)$ d_i values of size $\Theta(n)$ and the rest of the d_i values are zero, then the difference between (5.2) and (5.3) is a factor $\Theta(\log n)$, i.e. the upper bound on the expected number of swaps stated in Theorem 5.1 can be a factor of $\log n$ from the actual bound.

5.2 Experimental setup

In the remainder of this chapter, we investigate whether classic, theoretically non-adaptive sorting algorithms can show adaptive behavior in practice. We find that this indeed is the case—the running times for Quicksort and Mergesort are observed to improve by factors between 1.5 and 4 when the *Inv* value of the input goes from high to low. Furthermore, the improvements for Quicksort are in very good concordance with Theorem 5.1, which shows this result to be a likely explanation for the observed behavior.

In more detail, we study how the number of inversions in the input sequence affects the number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of data cache misses of the version of Quicksort shown in Figure 5.1. We also study the behavior of two variants of Quicksort, namely the randomized version that chooses the median of three random elements as a pivot, and the deterministic version that chooses the middle element as a pivot. We furthermore investigate different experimental setups for the Quicksort in Figure 5.1. We reduce the number of branch mispredictions by unrolling the inner loops three times, we turn off the hardware prefetcher to incur more cache misses, and we study expensive comparisons in two distinct ways, by making the elements doubles and using a comparison function respectively. Finally, we study the behavior of the classic sorting algorithm Mergesort, which also has an adaptive bahavior.

The input elements are distinct 4 byte integers. We generate two types of input, having small d_i's and large d_i's, respectively. We generate the sequence with small d_i's by choosing each element x_i randomly in $[i-d, \ldots, i+d]$ for some parameter d, making sure it is different than its predecessors. The sequence with large d_i's is generated by letting $x_i = i$ with the exception of d random i's which are permuted in the input. We perform our experiments by varying the disorder (by varying d) while keeping the size n of the input sequence constant. For most experiments, the input size is 2×10^6, but we also investigate larger and smaller input sizes.

Our experiments are conducted on two different machines. The first machine has a Dual Intel P4 3.4 GHz CPU with 1 GB RAM, running linux 2.6.12, while the other has an AMD Athlon XP 2400+ 2.0 GHz CPU with 256 MB RAM, running linux 2.4.22. On both machines the C source code was compiled using gcc-3.3.2 with optimization level -O3. The number of branch mispredictions and L1 data cache misses was obtained using the PAPI library [91] version 3.0.

Source code and the plotted data are available at `ftp://ftp.brics.dk/RS/04/47/Experiments`.

5.3 Experimental results

5.3.1 Quicksort.

We first analyze the dependence of the version of Quicksort shown in Figure 5.1 on the number of inversions in the input.

Figure 5.4 shows our data for the AMD architecture. The number of comparisons is independent of the number of inversions in the input, as expected. For the number of element swaps, the plot is very close to linear when considering the input sequence with small d_i's. Since the x-axis shows $\log(Inv)$, this is in very good correspondence with the bound $O(n(1 + \log(1 + \frac{Inv}{n})))$ of Theorem 5.1 (recall that n is fixed in the plot). For the input sequence with large d_i's, the plot is different. This is a sign of the slack in the analysis (for this type of input) noted after the proof of Theorem 5.1. We will demonstrate below that this curve is in very good correspondence with the version of the bound given by Equation (5.2). The plots for the number of branch mispredictions and for the running time clearly show that they are correlated with the number of element swaps. For the number of branch mispredictions, this is explained by the fact that an element swap is performed after the two while loops stop, and hence corresponds to two branch mispredictions. For the running time, it seems reasonable to infer that branch mispredictions are a dominant part of the running time of Quicksort on this type of architecture. Finally, the number of data cache misses seems independent of the presortedness of the input sequence, in correspondence with the fact that for all element swaps, the data to be manipulated is already in the cache and therefore the element swaps do not generate additional cache misses.

Figure 5.5 show the same plots for the P4 architecture, except that we were not able to obtain data for L2 data cache misses. We note that the plots follow the same trends as in Figure 5.4. On the P4, the number of comparisons and the number of element swaps are approximately the same as on the Athlon, but the running time is affected by up to a factor of 1.8 on the P4, while only by up to a factor of 1.45 on the Athlon. One reason for this behavior is the number of branch mispredictions, which is slightly smaller for the Athlon. Also, the length of the pipeline, shorter for Athlon, makes the branch mispredictions more costly for a P4 compared to an Athlon.

Similar observations on the resemblance between the data for the two architectures apply to all our experiments. For this reason, and because of the extra data for L2 data cache misses that we have for Athlon, we for the remaining plots restrict ourselves to the Athlon architecture.

We now turn to the variants of Quicksort. Figure 5.6 shows the number comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the L2 data cache misses for the version of Quicksort that chooses as a pivot the median of three random elements in the input sequence. We note that the plots have a behavior similar to the ones for the version of

Quicksort shown in Figure 5.4. However, some improvements are noticed. The three-median pivot Quicksort performs around 15% less comparisons, due to the better choice of the pivot. This immediately triggers a slight improvement in the number of data cache misses. Although the number of element swaps remains approximately the same, the number of branch mispredictions increases due to the extra branches used for computing the median of three elements. Also, the running time increases because of the increased number of branch mispredictions and random number generations.

Figure 5.7 shows the same plots for the deterministic version of Quicksort that chooses the middle element as pivot. In this case we note that the number of comparisons does depend on the presortedness of the input. This is because for small disorder, the middle element is very close to the median and therefore the number of comparisons is close to $n \log n$, as opposed to $\approx 1.4n \log n$ expected for the randomized Quicksort [66]. The good pivot choice for small disorder in the input also triggers a smaller number of branch mispredictions. However, for large disorder, the number of comparisons is larger compared to randomized median-of-three Quicksort due to bad pivot choices. Also, the running time is affected by up to a factor of two by the disorder in the input.

Figure 5.8 and Figure 5.9 show that when varying the input size n, the behavior of the plots remains the same for randomized Quicksort. Hence, our findings do not seem to be tied to the particular choice of $n = 2 \times 10^6$.

In Figure 5.10 we demonstrate that the number of element swaps is very closely related to $\sum_{i=1}^{n} \log d_i$, cf. the comment after the proof of Theorem 5.1. Hence the reason for the non-linear shape of the previous plots for input sequences with large d_i's seems to be the slack introduced (for this type of input) after Equation (5.2) in the proof of Theorem 5.1. As in the other cases, the running time and the number of branch mispredictions follow the same trend as the number of swaps.

To reduce the number of branch mispredictions, we unroll the inner loops three times.

5.3.2 Mergesort.

We briefly demonstrate that also for Mergesort the actual running time varies with the number of inversions in the input. We focus on the binary merge process, and count the number of times there is an alternation in which of the two input subsequences provides the next element output. It is easy to verify that the number of such alternations is dominated by the running time of the Mergesort algorithm by Moffat [86] based on merging by finger search trees, which was proved to have a running time of $O(n(1 + \log(1 + \frac{Inv}{n})))$, i.e. the number of alternations by standard Mergesort is $O(n(1 + \log(1 + \frac{Inv}{n})))$. The plots in Figure 5.12 show a very similar behavior for the number of alternations, the number of branch mispredictions, and the running time. The number of alternations is clearly correlated to the number of branch mispredictions, and these appear to be a dominant factor for the running time of Mergesort. The number of data cache misses increases only slightly for large disorder in the input.

5.4 Conclusions and related work

In this chapter we demonstrate that, in spite of common knowledge, the running time of the randomized version of Quicksort is adaptive with respect to measure *Inv*. Even though the expected number of comparisons is $O(n \log n)$, we prove that the expected number of element swaps is $O(n(1 + \log(1 + Inv/n)))$. Furthermore, we demonstrate experimentally that the number of element swaps performed follows closely the number of branch mispredictions, which are an important factor affecting the running time when computation takes place in internal memory. We observe that Mergesort has an adaptive behavior too.

Elmasry and Hammad [42] gave an empirical study for optimal algorithms with respect to *Inv*, and compare these algorithms against Quicksort. For Quicksort they measure the number of comparisons and the running time, obtaining results that are consistent to ours. They demonstrate that, for a low number of inversions, Quicksort is outperformed by some other algorithms, but its running time is still competitive. On the other hand, when the input sequence has a high *Inv* value, Quicksort outperforms all the *Inv* optimal algorithms considered.

For Heapsort, Figure 5.11 shows the way the number of inversions in the input affects the number of comparisons, the number of elements swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses for input sequences of constant length $n = 2 \times 10^6$. The number of comparisons and the number of element swaps performed by Heapsort is affected slightly, while the number of branch mispredictions is affected in a more significant way, by a factor of 0.4. However, the number of L2 data cache misses is greatly affected, and varies by more than a factor of ten. The running time shows a virtually identical behavior with the data cache misses, except the increase is by a factor close to four. This suggests that data cache misses are the dominant factor for the running time for Heapsort on this architecture. We leave open the question of a theoretical analysis of the number of cache misses of Heapsort as a function of *Inv*.

An interesting sorting algorithm to be considered for study is Shellsort, introduced by Shell in [106] and improved over the years (see [104] for a comprehensive survey). Since it is based on Insertionsort, we expect Shellsort to outperform some optimal sorting algorithms for a very small number of inversions, because of a very small number of comparisons and branch mispredictions. Intuitively, Insertionsort performs $O(n)$ branch mispredictions, because the branch testing the element to be inserted against some element in the sequence should be correctly predicted with one exception, when the element gets inserted.

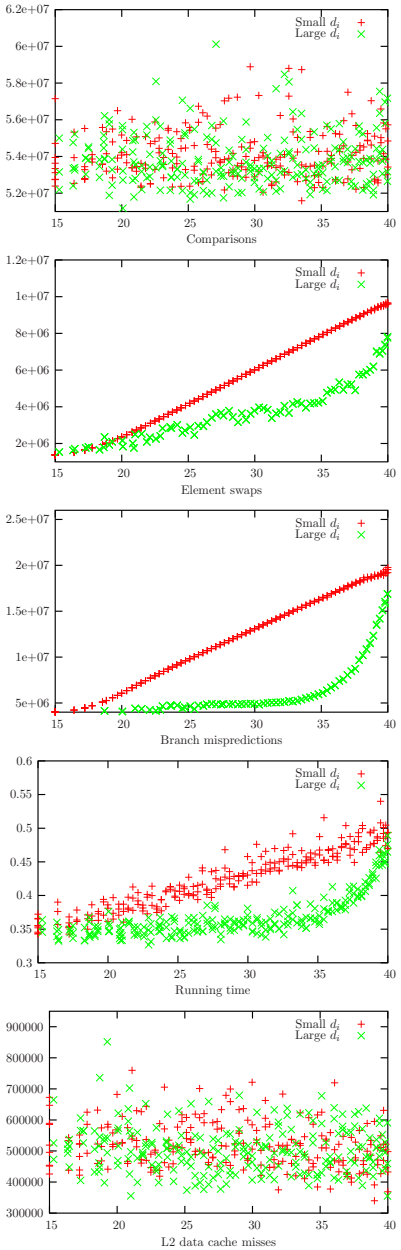

Figure 5.4: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 2 \times 10^6$. The x-axis shows $\log(Inv)$.

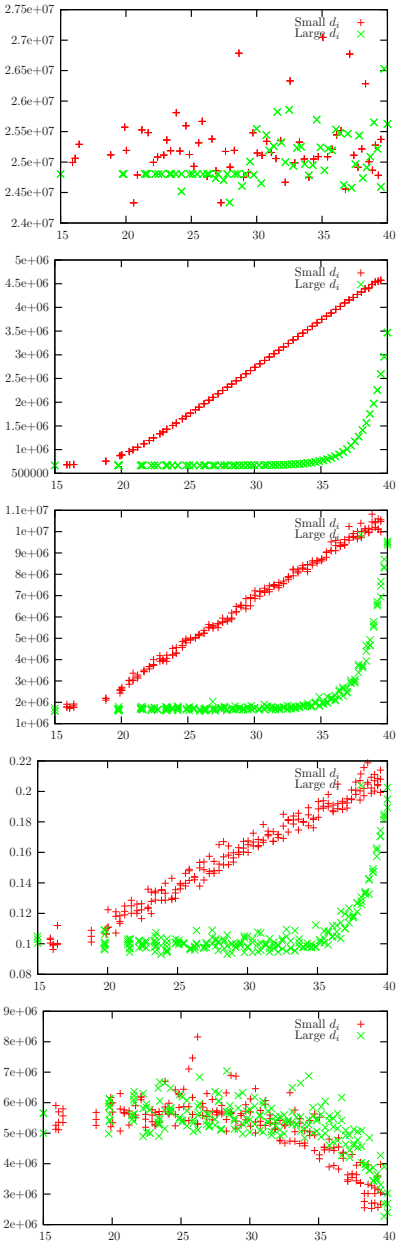

Figure 5.5: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses of randomized Quicksort on P4, for $n = 10^6$. The x-axis shows $\log(Inv)$.

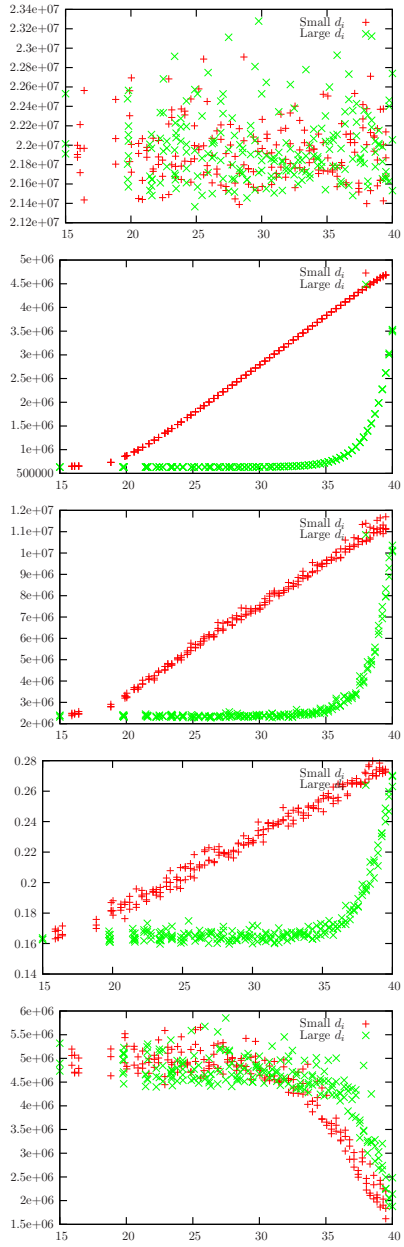

Figure 5.6: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by randomized median-of-three Quicksort on P4, for $n = 10^6$. The x-axis shows $\log(Inv)$.

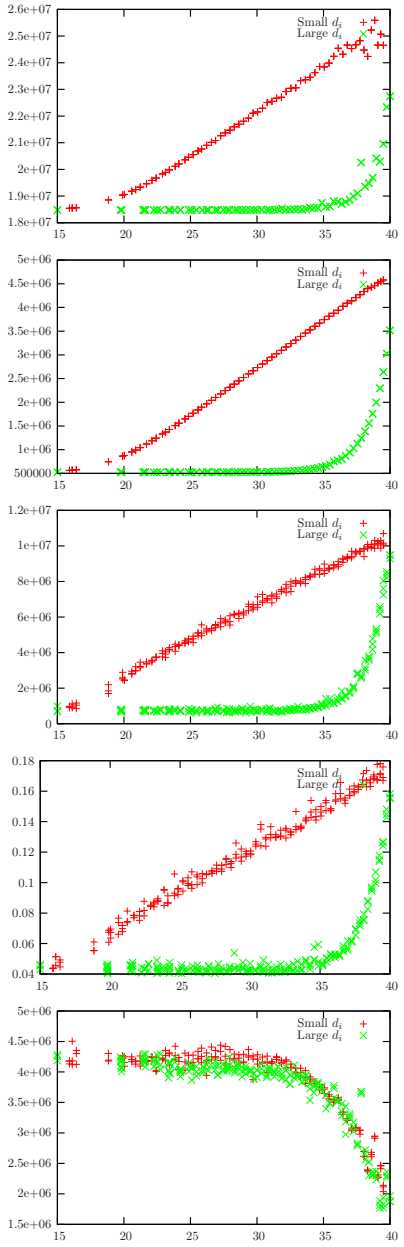

Figure 5.7: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by deterministic Quicksort on P4, for $n = 10^6$. The x-axis shows $\log(Inv)$.

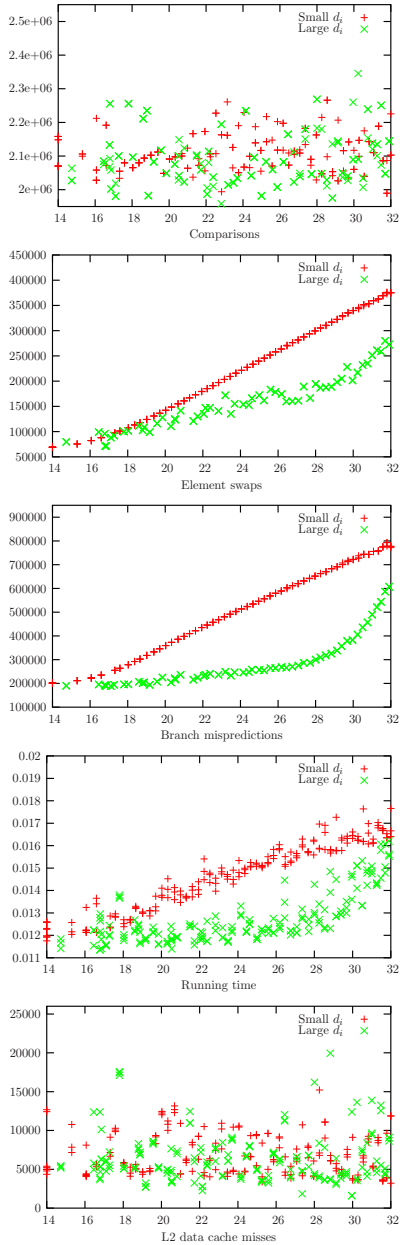

Figure 5.8: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 6 \times 10^4$. The x-axis shows $\log(Inv)$.

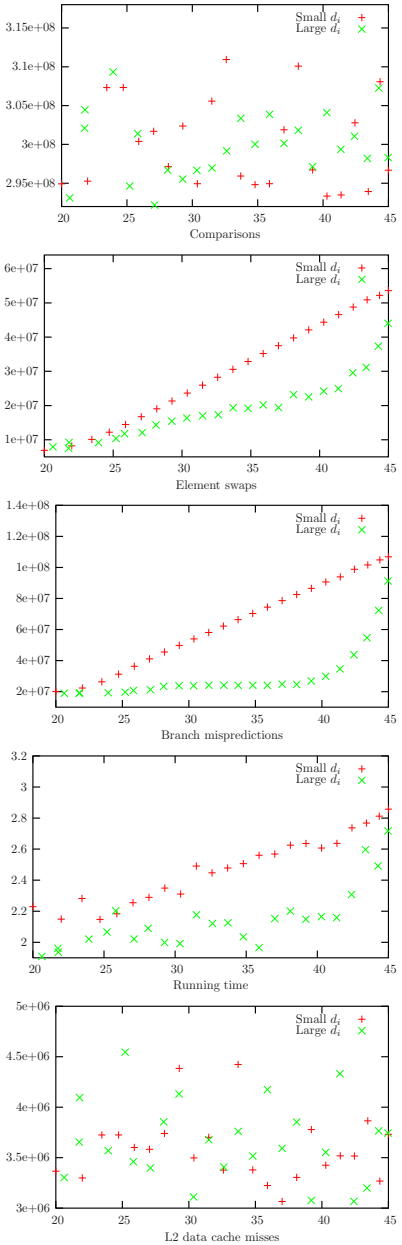

Figure 5.9: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for $n = 10^7$. The x-axis shows $\log(Inv)$.

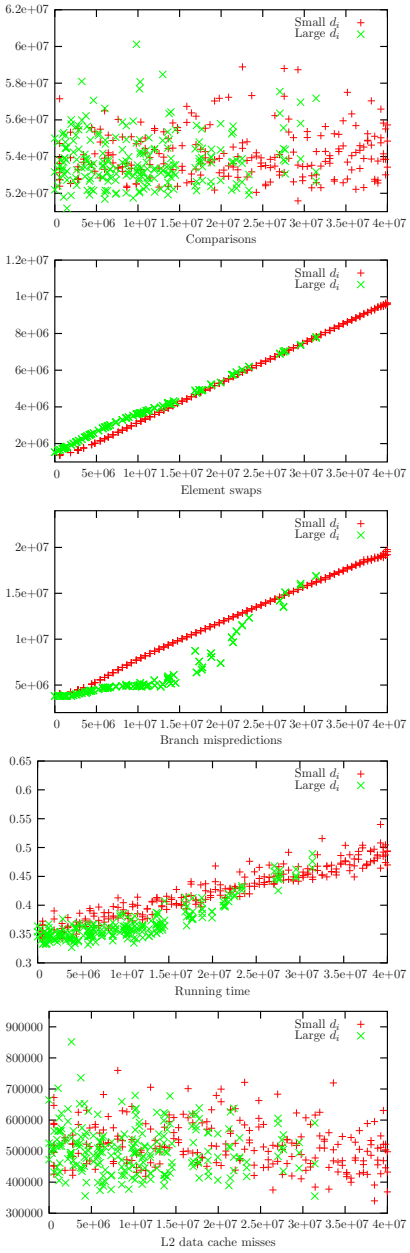

Figure 5.10: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by randomized Quicksort on Athlon, for the input size $n = 2 \times 10^6$. The x-axis shows $\sum_{i=1}^{n} \log(d_i + 1)$.

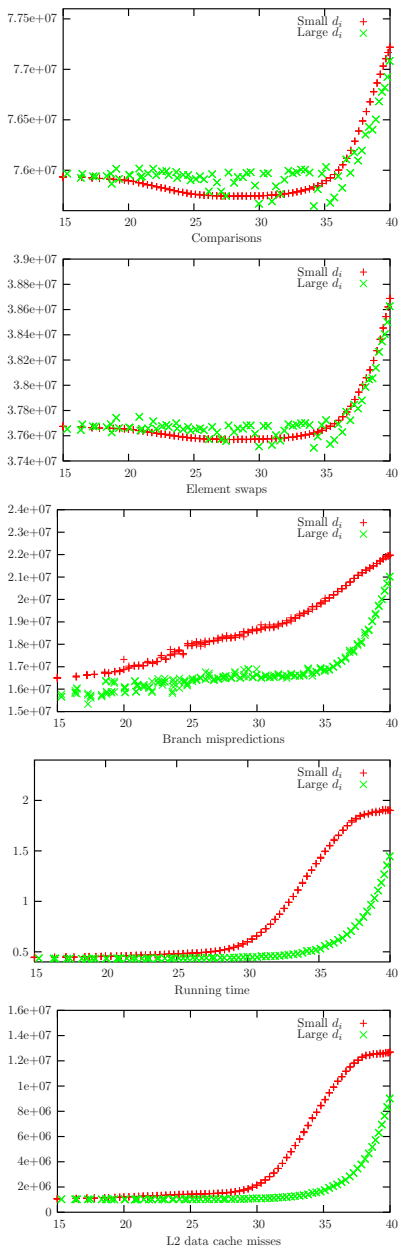

Figure 5.11: The number of comparisons, the number of element swaps, the number of branch mispredictions, the running time, and the number of L1 data cache misses performed by Heapsort on P4, for $n = 2 \times 10^6$. The x-axis shows $\log(Inv)$.

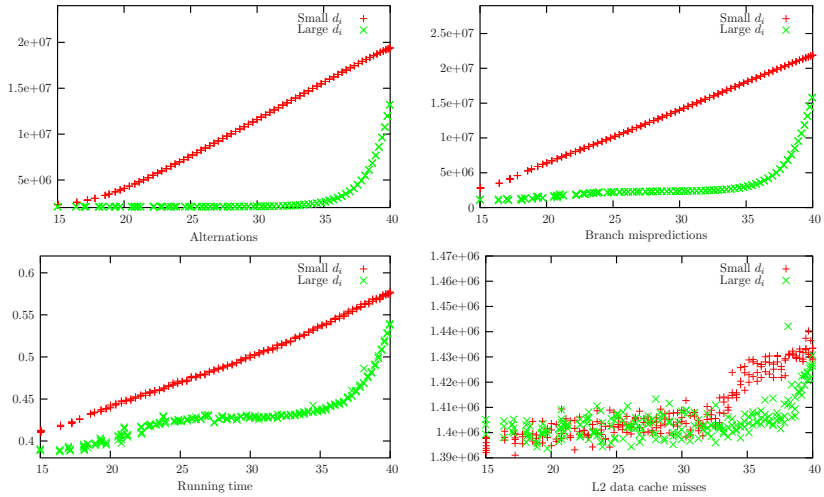

Figure 5.12: The number of alternations, the number of branch mispredictions, the running time, and the number of L2 data cache misses performed by Mergesort on Athlon, for $n = 2 \times 10^6$. The x-axis shows $\log(Inv)$.

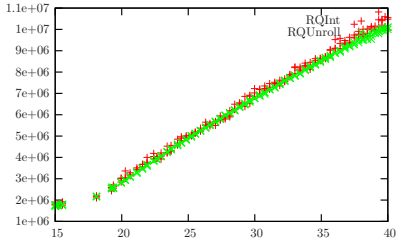

Figure 5.13: The number of branch mispredictions performed by Quicksort on P4 for small d_i's, with no loop unrolling and with three unrolls of the inner loops. The input size is $n = 10^6$, and the x-axis shows $\log(Inv)$.

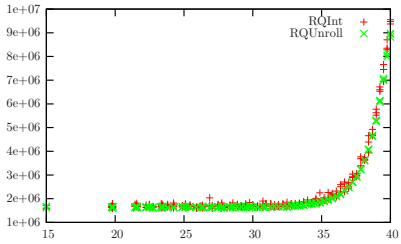

Figure 5.14: The number of branch mispredictions performed by Quicksort on P4 for large d_i's, with no loop unrolling and with three unrolls of the inner loops. The input size is $n = 10^6$, and the x-axis shows $\log(Inv)$.

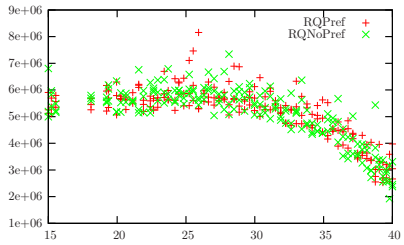

Figure 5.15: The number of data cache misses performed by randomized Quicksort on P4 for small d_i's, when the hardware prefetcher turned on and off. The input size is $n = 10^6$, and the x-axis shows $\log(Inv)$.

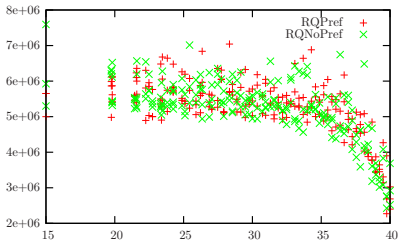

Figure 5.16: The number of data cache misses performed by randomized Quicksort on P4 for large d_i's, when the hardware prefetcher turned on and off. The input size is $n = 10^6$, and the x-axis shows $\log(Inv)$.

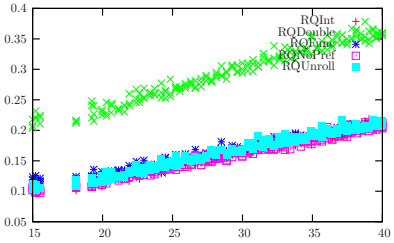

Figure 5.17: The running time of randomized Quicksort on P4 for small d_i's, for integer elements, double elements, comparison function, hardware prefetcher turned off, and the inner loops unrolled. The input size is $n = 10^6$ and the x-axis shows $\log(Inv)$.

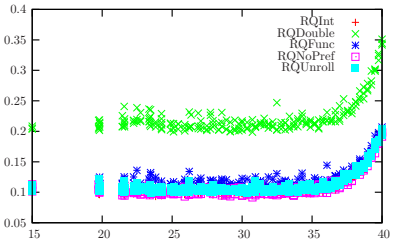

Figure 5.18: The running time of randomized Quicksort on P4 for large d_i's, for integer elements, double elements, comparison function, hardware prefetcher turned off, and the inner loops unrolled. The input size is $n = 10^6$ and the x-axis shows $\log(Inv)$.

Chapter 6

Trading Branch Mispredictions for Comparisons when Sorting

Cute result.
— Anonymous reviewer

In this chapter we present the work published in [30]. In Section 6.1 we prove lower bound tradeoffs between the number of comparisons and the number of branch mispredictions for comparison based sorting and adaptive sorting algorithms. Matching upper bounds are provided in Sections 6.2 and 6.3, where we show how variants of multiway MergeSort and GenericSort, respectively, achieve the optimal tradeoffs between comparisons and branch mispredictions.

6.1 Lower bounds for sorting

In this section we consider deterministic comparison based sorting algorithms and prove lower bound tradeoffs between the number of comparisons and the number of branch mispredictions performed, under the assumption that each comparison between two elements in the input is immediately followed by a conditional branch that might be predicted or mispredicted. This property is satisfied by most sorting algorithms.

Theorem 6.1 introduces a worst case tradeoff between the number of comparisons and the number of branch mispredictions performed by sorting algorithms.

Theorem 6.1 *Consider a deterministic comparison based sorting algorithm A that sorts input sequences of size n using $O(dn \log n)$ comparisons, $d > 1$. The number of branch mispredictions performed by A is $\Omega(n \log_d n)$.*

Proof. Let T be the decision tree corresponding to A (for a definition of decision trees see e.g. [34, Section 8.1]). By assumption, each node in the tree corresponds to a branch that can be either predicted or mispredicted. We label the edges corresponding to mispredicted branches with 1 and the edges corresponding to correctly predicted branches with 0. Each leaf is uniquely labeled with the labels on the path from the root to the given leaf. Assuming the depth of

the decision tree is at most D and the number of branch mispredictions allowed is k, each leaf is labeled by a sequence of at most D 0's and 1's, containing at most k 1's. By padding the label with 0's and 1's we can assume all leaf labels have length exactly $D+k$ and contain exactly k 1's. It follows that the number of labelings is at most the binomial coefficient $\binom{D+k}{k}$ and therefore the number of leaves is at most $\binom{D+k}{k}$.

Denoting the number of leaves by $N \geq n!$, we obtain that $\binom{D+k}{k} \geq N$, which implies $\log \binom{D+k}{k} \geq \log N$. Using $\log \binom{D+k}{k} \leq k(O(1) + \log \frac{D}{k})$ we obtain that:

$$k\left(O(1) + \log \frac{D}{k}\right) \geq \log N \ . \tag{6.1}$$

Consider $D = \delta \log N$ and $k = \varepsilon \log N$, where $\delta \geq 1$ and $\varepsilon \geq 0$. We obtain:

$$\varepsilon \log N \left(O(1) + \log \frac{\delta}{\varepsilon}\right) \geq \log N \ ,$$

and therefore $\varepsilon\left(O(1) + \log \frac{\delta}{\varepsilon}\right) \geq 1$. Using $\delta = O(d)$ we obtain $\varepsilon = \Omega(1/\log d)$. Taking into account that $\log N \geq \log(n!) = n \log n - O(n)$ we obtain $k = \Omega(n \log_d n)$. □

Manilla [82] introduced the concept of optimal adaptive sorting algorithms. Given an input sequence X and some measure of presortedness M, consider the set below(X, M) of all permutations Y of X such that $M(Y) \leq M(X)$. Considering only inputs in below(X, M), a comparison based sorting algorithm performs at least $\log |\text{below}(X, M)|$ comparisons in the worst case. In particular, an adaptive sorting algorithm that is optimal with respect to measure *Inv* performs $O(n(1 + \log(1 + Inv/n)))$ comparisons [45].

Theorem 6.2 introduces a worst case tradeoff between the number of comparisons and the number of branch mispredictions for comparison based sorting algorithms that are adaptive with respect to measure *Inv*.

Theorem 6.2 *Consider a deterministic comparison based sorting algorithm A that sorts an input sequence of size n using $O(dn(1+\log(1+Inv/n)))$ comparisons, where Inv denotes the number of inversions in the input. The number of branch mispredictions performed by A is $\Omega(n \log_d(1 + Inv/n))$.*

Proof. We reuse the proof of Theorem 6.1 by letting $N = |\text{below}(X, M)|$, for an input sequence X.

Using (6.1), with the decision tree depth $D = \delta n(1 + \log(1 + Inv/n))$ when restricted to inputs in below(X, M), $k = \varepsilon n(1+\log(1 + Inv/n))$ branch mispredictions, and $\log N = \Omega(n(1 + \log(1 + Inv/n)))$ [58], we obtain:

$$\varepsilon n \left(1 + \log\left(1 + \frac{Inv}{n}\right)\right)\left(O(1) + \log \frac{\delta}{\varepsilon}\right) = \Omega\left(n\left(1 + \log\left(1 + \frac{Inv}{n}\right)\right)\right) \ .$$

This leads to:

$$\varepsilon\left(O(1) + \log \frac{\delta}{\varepsilon}\right) = \Omega(1) \ ,$$

Measure	Comparisons	Branch mispredictions
Dis	$O(dn(1+\log(1+Dis)))$	$\Omega(n\log_d(1+Dis))$
Exc	$O(dn(1+Exc\log(1+Exc)))$	$\Omega(nExc\log_d(1+Exc))$
Enc	$O(dn(1+\log(1+Enc)))$	$\Omega(n\log_d(1+Enc))$
Inv	$O(dn(1+\log(1+Inv/n)))$	$\Omega(n\log_d(1+Inv/n))$
Max	$O(dn(1+\log(1+Max)))$	$\Omega(n\log_d(1+Max))$
Osc	$O(dn(1+\log(1+Osc/n)))$	$\Omega(n\log_d(1+Osc/n))$
Reg	$O(dn(1+\log(1+Reg)))$	$\Omega(n\log_d(1+Reg))$
Rem	$O(dn(1+Rem\log(1+Rem)))$	$\Omega(nRem\log_d(1+Rem))$
Runs	$O(dn(1+\log(1+Runs)))$	$\Omega(n\log_d(1+Runs))$
SMS	$O(dn(1+\log(1+SMS)))$	$\Omega(n\log_d(1+SMS))$
SUS	$O(dn(1+\log(1+SUS)))$	$\Omega(n\log_d(1+SUS))$

Figure 6.1: Lower bounds on the number of branch mispredictions for deterministic comparison based adaptive sorting algorithms for different measures of presortedness, given the upper bounds on the number of comparisons.

and therefore $\varepsilon = \Omega(1/\log\delta)$. Taking into account that $\delta = O(d)$ we obtain that $\varepsilon = \Omega(1/\log d)$, which leads to $k = \Omega(n\log_d(1+Inv/n))$. □

Using a similar technique, lower bounds for other measures of presortedness can be obtained. For comparison based adaptive sorting algorithms, Figure 6.1 states lower bounds on the number of branch mispredictions performed in the worst case, assuming the given upper bounds on the number of comparisons. For definitions of different measures of presortedness, refer to [45].

6.2 An optimal sorting algorithm

In this section we introduce *Insertion d-way MergeSort*. It is a variant of d-way MergeSort that achieves the tradeoff stated in Theorem 6.1 by using an insertion sort like procedure for implementing the d-way merger. The merger is proven to perform a linear number of branch mispredictions.

We maintain two auxiliary vectors of size d. One of them stores a permutation $\pi = (\pi_1, \ldots, \pi_d)$ of $(1, \ldots, d)$ and the other one stores the indices in the input of the current element in each subsequence $i = (i_{\pi_1}, \ldots, i_{\pi_d})$, such that the sequence $(x_{i_{\pi_1}}, \ldots, x_{i_{\pi_d}})$ is sorted. During the merging, $x_{i_{\pi_1}}$ is appended to the output sequence and i_{π_1} is incremented by 1 and then inserted in the vector i in a manner that resembles insertion sort: in a scan the value $y = x_{i_{\pi_1}}$ to be inserted is compared against the smallest elements of the sorted sequence until an element larger than y is encountered. This way, the property that the elements in the input sequence having indices $i_{\pi_1}, \ldots, i_{\pi_d}$ are in sorted order holds at all times. We also note that for each insertion the merger performs $O(1)$ branch mispredictions, even using a static branch prediction scheme.

Theorem 6.3 *Insertion d-way MergeSort performs $O(dn\log n)$ comparisons and $O(n\log_d n)$ branch mispredictions.*

Proof. For the simplicity of the proof, we consider a static prediction scheme where for the merging phase the element to be inserted is predicted to be larger than the minimum in the indices vector.

The number of comparisons performed at each level of recursion is $O(dn)$, since in the worst case each element is in the worst case compared against $d-1$ elements at each level. Taking into account that the number of recursion levels is $\lceil \log_d n \rceil$, the total number of comparisons is $O(dn \log_d n) = O(dn \log n)$.

In what concerns the number of branch mispredictions, for each element Insertion d-way MergeSort performs $O(1)$ branch mispredictions for each recursion level. That is because each element is inserted at most once in the indices array i at a given recursion level and for insertion sort each insertion is performed by using a constant number of branch mispredictions. Therefore we conclude that Insertion d-way MergeSort performs $O(n \log_d n)$ branch mispredictions. □

We stress that Theorem 6.3 states an optimal tradeoff between the number of comparisons and the number of branch mispredictions. This allows tuning the parameter d, such that Insertion d-way Mergesort can achieve the best running time on different architectures depending on the CPU characteristics, i.e. the clock speed and the pipeline length.

6.3 Optimal adaptive sorting

In this section we describe how d-way merging introduced in Section 6.2 can be integrated within *GenericSort* by Estivill-Castro and Wood [44], using a greedy-like division protocol. The resulting algorithm is proved to achieve the tradeoff between the number of comparisons and the number of branch mispredictions stated in Theorem 6.2.

GenericSort is based on MergeSort and works as follows: if the input is small, it is sorted using some alternate sorting algorithm; if the input is already sorted, the algorithm returns. Otherwise, it splits the input sequence into d subsequences of roughly equal sizes according to some *division protocol*, after which the subsequences are recursively sorted and finally merged to provide the sorted output.

The division protocol that we use, *GreedySplit*, is a generalization of the binary division protocol introduced in [28]. It partitions the input in $d+1$ subsequences $S_0, \ldots, S_d$, where S_0 is sorted and $S_1, \ldots, S_d$ have balanced sizes. In a single scan from left to right we build S_0 in a greedy manner while distributing the other elements to subsequences $S_1, \ldots, S_d$ as follows: each element is compared to the last element of S_0, if it is larger, it is appended to S_0; if not, it is distributed to an S_j such that at all times the i^{th} element in the input that is not in S_0 is distributed to $S_{1+i \bmod d}$. It is easy to see that S_0 is sorted and $S_1, \ldots, S_d$ have balanced sizes. For merging we use the insertion sort based merger introduced in Section 6.2.

Lemma 6.1 generalizes Lemma 3 in [28] to the case of d-way splitting.

Lemma 6.1 *If GreedySplit splits an input sequence X in $d+1$ subsequences*

$S_0, \ldots, S_d$, where S_0 is sorted and $d \geq 2$, then

$$Inv(X) \geq Inv(S_1) + \cdots + Inv(S_d) + \frac{d-1}{4}(Inv(S_1) + \cdots + Inv(S_d)) .$$

Proof. Let $X = (x_1, \ldots, x_n)$ and $S_i = (s_{i1}, \ldots, s_{it})$, for $1 \leq i \leq d$. For each s_{ij} denote by δ_{ij} its index in the input. By construction, S_i is a subsequence of X.

For some subsequence S_i consider an inversion $s_{ii_1} > s_{ii_2}$, with $i_1 < i_2$. By construction we know that for each subsequence S_k, with $k \neq i$, there exists some $s_{k\ell} \in S_k$ such that in the input sequence we have $\delta_{ii_1} < \delta_{k\ell} < \delta_{ii_2}$, see Figure 6.2. We prove that there exists at least an inversion between $s_{k\ell}$ and s_{ii_1} or s_{ii_2} in X. If $s_{k\ell} < s_{ii_2} < s_{ii_1}$ then there is an inversion between $s_{k\ell}$ and s_{ii_1}; if $s_{ii_2} < s_{k\ell} < s_{ii_1}$ then there are inversions in the input between $s_{k\ell}$ and both s_{ii_1} and s_{ii_2}; finally, if $s_{ii_2} < s_{ii_1} < s_{k\ell}$, there is an inversion between $s_{k\ell}$ and s_{ii_2}. Let $s_{k\ell_1}, \ldots, s_{k\ell_z}$ be all the elements in S_k such that $i_1 < \delta_{k\ell_1} < \cdots < \delta_{k\ell_z} < i_2$, i.e. all the elements from S_k that appear in the input between ranks δ_{ii_1} and δ_{ii_2}.

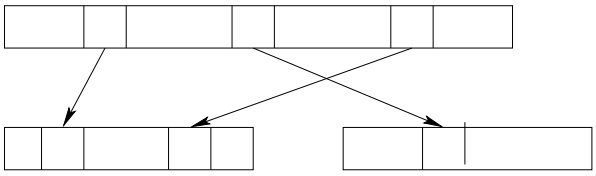

Figure 6.2: Greedy division protocol. Between any two elements in S_i there is at least one element in S_k in the input sequence.

We proved that there is an inversion between $s_{k\ell_{\lfloor (1+z)/2 \rfloor}}$ and at least one of s_{ii_1} and s_{ii_2}. Therefore, for the inversion (s_{ii_1}, s_{ii_2}) in S_i we have identified an inversion in X between an element in S_k and an element in S_i that is not present in any of $S_1, \ldots, S_d$. But this inversion can be counted for at most two different pairs in S_i, namely (s_{ii_1}, s_{ii_2}) and $(s_{ii_1}, s_{i(i_2+1)})$ if there is an inversion between s_{ii_1} and $s_{k\ell_{\lfloor (1+z)/2 \rfloor}}$ or (s_{ii_1}, s_{ii_2}) and $(s_{i(i_1-1)}, s_{ii_2})$ otherwise. In a similar manner in S_k the same inversion can be counted two times. Therefore, we obtain that for each inversion in S_i there is an inversion between S_i and S_k that can be counted four times. Taking into account that all the inversions in $S_1, \ldots, S_d$ are also in X, we obtain:

$$Inv(X) \geq Inv(S_1) + \cdots + Inv(S_d) + \frac{d-1}{4}(Inv(S_1) + \cdots + Inv(S_d)) .$$

□

Theorem 6.4 *GreedySort performs $O(dn(1+\log(1+Inv/n)))$ comparisons and $O(n \log_d(1 + Inv/n))$ branch mispredictions.*

Proof. We assume a static branch prediction scheme. For the division protocol we assume that at all times the elements are smaller than the maximum of S_0, meaning that branch mispredictions occur when elements are appended

to the sorted sequences. This leads to a total of $O(1)$ branch mispredictions per element for the division protocol, because the sorted sequences are not sorted recursively. For the merger, the element to be inserted is predicted to be larger than the minimum in the indices vector at all times. Following the proof of Theorem 6.3, we obtain that splitting and merging take $O(1)$ branch mispredictions per element for each level of recursion.

We follow the proof in [80]. First we show that at the first levels of recursion, until the number of inversions gets under n/d, GreedySort performs $O(dn(1 + \log(1+Inv/n))$ comparisons and $O(n(1+\log_d(1+Inv/n))$ branch mispredictions. Afterwards, we show that the remaining levels consume a linear number of branch mispredictions and comparisons.

We first find the level ℓ for which the number of inversions gets below n/d. Denote by Inv_i the total number of inversions in the subsequences at level i. Using the result in Lemma 6.1, we obtain $Inv_i \leq \left(\frac{4}{d+3}\right)^i Inv$. The level ℓ should therefore satisfy:

$$\left(\frac{4}{d+3}\right)^{\ell} Inv \leq \frac{n}{d} \,,$$

implying $\ell \geq \log_{\frac{d+3}{4}} \frac{Inv \cdot d}{n}$.

Taking into account that at each level of recursion the algorithm performs $O(dn)$ comparisons and $O(n)$ branch mispredictions, we obtain that for the first $\ell = \lceil \log_{\frac{d+3}{4}} \frac{Inv \cdot d}{n} \rceil$ levels we perform $O(dn \log_d(Inv/n)) = O(dn \log(Inv/n))$ comparisons and $O(n \log_d(Inv/n))$ branch mispredictions.

We prove that for the remaining levels we perform a linear number of comparisons and branch mispredictions.

Let $L(x)$ be the recursion level where some element x is placed in a sorted sequence and $L(x) \geq \ell$. For each level of recursion j, where $\ell \leq j < L(x)$, x is smaller than the maximum in the sorted subsequence S_0 and therefore there is an inversion between x and the maximum in S_0 that does not exist in the recursive levels $j+1, j+2, \ldots$. It follows that $L(x)-\ell$ is bounded by the number of inversions with x at level ℓ.

Taking into account that the total number of inversions at level ℓ is at most n/d and that for each element at a level we perform $O(d)$ comparisons, we obtain that the total number of comparisons performed at the levels $\ell + 1, \ell + 2, \ldots$ is $O(n)$. Similarly, using the fact that for each element at each level $O(1)$ mispredictions are performed, we obtain that the total number of branch mispredictions performed for the levels below ℓ is $O(n/d)$. □

Chapter 7

Skewed Binary Search Trees

The idea that the cost of branch predictions may alter the notion of what constitute a well balanced tree is fresh and interesting.
— Anonymous reviewer

In this chapter we show the results concerning skewed binary search trees, which were published in [31]. In Section 7.1 we describe skewed balanced search trees and give an upper bound on the running time performed for a random query. In Section 7.2 we give brief insights on the hardware issues that affect the running time in practice. For a random query we give upper bounds on the number of branch mispredictions in Section 7.3, while in Section 7.4 we introduce different memory layouts and give upper bounds on the number of cache misses. In Section 7.5 we describe the setup for the experiments we perform and in Section 7.6 we show and discuss our experimental results.

7.1 Skewed binary search trees

A skewed binary search tree is a binary search tree where there exists a constant α, $0 < \alpha \leq 1/2$, such that for each node v there is a fixed ratio between the number of nodes in the subtree rooted in the left child and the subtrees rooted at v. More precisely, $\text{size}(\text{left}(v)) = \lfloor \alpha \cdot \text{size}(v) \rfloor$, where $\text{size}(v)$ denotes the number of nodes in the subtree rooted at v.

Skewed binary search trees are the extreme unbalanced cases of BB[α] trees of Nievergelt and Reingold [89].

Theorem 7.1 (Mehlhorn, Section III.5.1) *The average path length P is at most $(1 + 1/n)\log(n + 1)/H(\alpha)$, where $H(\alpha) = -\alpha \log \alpha - (1 - \alpha)\log(1 - \alpha)$.*

In practice, due to hardware issues, the running time spent at a given node might depend on the next node to process, i.e. the left or right child. In Corollary 7.1 we analyze the running time for a random search in the case where the costs for visiting the left and right children of a given node are different.

Corollary 7.1 *Consider a skewed search tree T of balance α, and let c_l and c_r be the costs for branching left and right respectively. A random search has*

$$O((\alpha c_l + (1 - \alpha)c_r)\log n/H(\alpha)) \tag{7.1}$$

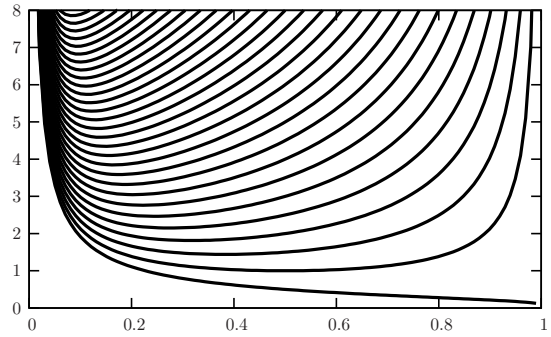

Figure 7.1: Bound on the expected cost for a random search, where the cost for visiting the left child is $c_l = 1$ and the cost for processing the right child is $c_r = 0, 1, 2, \ldots, 28$ ($c_r = 0$ being the lowest curve).

expected cost, where $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$.

Proof. Due to the linearity of expectation, the expected number of comparisons performed for a random search is equal to the average path length, which is $O(\log n / H(\alpha))$ cf. Theorem 7.1. If for branching left and right we have costs c_l and c_r, we obtain at a given node an expected cost of $\alpha c_l + (1-\alpha)c_r$, since the probabilities of branching left and right are α and $1-\alpha$ respectively. We conclude that the expected cost of a random search is $O((\alpha c_l + (1-\alpha)c_r)\log n / H(\alpha))$. □

In Figure 7.1 we show the function from the bound (7.1) on the expected cost for a random search where we consider different costs for visiting the left and the right child respectively. We note that in all the cases where $c_l \neq c_r$ the minimum occurs for α values different than $1/2$.

7.2 Hardware discussion

The running time of algorithms is usually analyzed by counting the instructions performed by the CPU. However, in practice, the running time of an algorithm can be severely affected by some other hardware factors besides the CPU instructions. We show that the branch mispredictions that occur in the CPU and the cache faults can have a major effect over the running time of searching in skewed binary search trees.

To increase the clock speed, modern CPUs include instruction pipelines in their architecture, where the instructions are prefetched before being executed. When a conditional branch enters the pipeline, its outcome is not known prior to its execution and thus its direction must be predicted to ensure the prefetching of the following instructions. If the branch is incorrectly predicted, the whole pipeline must be flushed, since the instructions in the pipeline correspond to a wrong execution path. This obviously leads to a performance loss, which increases proportionally with the length of the pipeline. In such a case, we say that a *branch misprediction* occurs. Since the pipelines are getting longer

and longer (e.g. 18 instructions for Pentium P4 and 31 for Intel Prescott), branch mispredictions are having an increasing influence over the running time of algorithms in practice.

In the traditional RAM model, all memory accesses are considered to have equal access times. In practice, nowadays computers have a hierarchy of memory layers, each of them having smaller size and access time than the next one, from the CPU registers to the hard-disk. The data can be transfered only between consecutive layers, and is performed in *blocks* of consecutive data rather than individual items.

7.3 Branch mispredictions

Branch mispredictions can dramatically affect the running time in practice. Even though in most of the cases the branch predictors incorporated in the CPU architectures are accurate and yield good performances, in certain algorithms the outcome of certain branches is hard to guess. Sorting and searching are two such examples, since they involve comparisons among elements and the outcome of an element comparison is usually hard to predict.

There are two major types of branch prediction schemes, namely static and dynamic. In static branch predictors, each branch is predicted in the same direction at all times, and the direction of the branch is either given at compile time or it follows some simple heuristics, e.g. forward branches predicted taken and backward branches predicted not taken. On the other hand, the dynamic branch prediction schemes predict the direction of the branches at runtime, taking advantage of the execution history. In the case of searching in a balanced search tree, since the number of nodes in the left and right subtrees of a given node are approximately the same, the outcome of any branch is hard to predict and hence we expect branch mispredictions in around half of the cases. On the other hand, for the skewed search trees, we expect the number of branch mispredictions to decrease when increasing the skewness, since the probability that the search key lies in the larger subtree is increasing. In Theorem 7.2 we prove an upper bound on the number of branch mispredictions performed for a skewed binary search tree when a static branch predictor is used.

Theorem 7.2 *The expected number of branch mispredictions performed for a random search in a skewed binary search tree of balance α is $O(\alpha \log n / H(\alpha))$, where $H(\alpha) = -\alpha \log \alpha - (1-\alpha)\log(1-\alpha)$, assuming a static branch predictor and $0 \leq \alpha \leq 1/2$.*

Proof. Since we consider $\alpha \leq 1/2$, for each non-leaf node of the search tree, the right subtree will have more nodes than the left subtree, hence visiting the right subtree next is more likely than visiting the left subtree. We use a static prediction scheme where for each node we predict that the search key is larger than the key stored at the given node. Using Corollary 7.1 with $c_l = 1$ and $c_r = 0$, we obtain that for a random search we perform expected $O(\alpha \log n / H(\alpha))$ branch mispredictions. □

7.4 Memory layouts

The difference in access times between the different layers of the memory hierarchy, especially from the internal memory to the hard disk, has led to several models that deal with capturing the cache effect. One of the most successful is the I/O model introduced by Aggarwal and Vitter [1] and consist of a two level memory hierarchy, containing a fast memory of bounded size M and a slow, infinite memory. The computation is performed in the fast memory and the data is transfered between the slow and fast memories in blocks of B consecutive items. The I/O complexity of an algorithm is given by the number of blocks transfered. Since in practice hardware architectures contain several memory levels with different values for the fast memory size M and the block size B, Frigo *et al.* [52] introduced the cache oblivious model. A cache oblivious algorithm is an algorithm whose analysis holds for any values of M and B. Most of the algorithms in this model assume a tall cache, i.e. $M = \Omega(B^2)$. For a comprehensive list of efficient external memory algorithms, e.g. refer to [8, 10, 21, 117].

We analyze different memory layouts for the static skewed binary search trees. For all the layouts the tree is stored as an array of n nodes, where each node is a structure containing two pointers to the left and the right subtree respectively together with an integer key. We note that the number of comparisons and branch mispredictions performed for searches is not affected by the way the tree is laid in memory, as they only depend on the height of the tree and the number of left turns on a path from the root to a certain leaf (for $\alpha < 1/2$, assuming a static branch prediction scheme). However, the number of cache faults can be dramatically affected by the memory layout, ranging from $O(1/\log B)$ to $O(1)$ I/Os for each node on a search path.

Consider a balanced binary search tree T of n nodes. The different memory layouts that we consider together with the expected number of I/Os for a random search are introduced below.

Random. Each node of T is stored at a random position in the array.

Since in this layout the nodes are stored at random locations in the array, for each node on a search path we perform an I/O, hence the expected number of I/Os is given by the average path length.

BFS. In this layout the nodes of the tree are stored according to the BFS traversal of T, where the nodes at a level are processed in a left-to-right order.

The first B nodes of the array contain the topmost subtree. In any practical setting, i.e. the tree is not severely skewed, the length of any path in this subtree is $\Theta(\log B)$. The top subtree is loaded into memory using in a single I/O, hence for the first $O(\log B)$ nodes on any path we use $O(1)$ I/Os. Afterward, for the remaining nodes on any search path we consume $O(1)$ I/Os per node, thus obtaining expected $O(1 + |P| - \log B)$ I/Os for a following a search path P.

Inorder. The tree is stored in the array according to the inorder traversal, i.e. the array is sorted.

Following a path from the root to a leaf takes $O(1)$ I/Os per node, except for possibly the last subtree of $\Theta(B)$ nodes, since they will be loaded using a single I/O. Considering the case when in a subtree of size B the length of a search path is $O(\log B)$, we obtain that for a search path P in this layout we perform between $O(|P|)$ and $O(1 + |P| - \log B)$ I/Os, where $|P|$ denotes the length of P, depending whether P reaches the bottom levels of the tree or not.

DFSl. The tree is laid out in the array according to a DFS traversal, where after visiting the root, the left child is traversed before the right child.

Since the left child is stored next to the parent, they are stored in the same block, hence branching left takes $O(1/B)$ I/Os. In what concerns the right child, accessing it requires $O(1)$ I/Os. Using Corollary 7.1 we obtain that for a random search we perform expected $O((\alpha/B + (1 - \alpha)) \log n/H(\alpha))$ I/Os, where $H(\alpha) = -\alpha \log \alpha - (1 - \alpha) \log(1 - \alpha)$.

DFSr. This layout is similar to DFSl, except for the fact that the right child is traversed first and the left child afterwards. Using a similar argument, we obtain that the number of I/Os performed for a random search is expected $O(\alpha + (1 - \alpha)/B) \log n/H(\alpha)$.

k-level grouping. Given a tree T, in this layout we first store the first k levels of T in the order given by a BFS traversal and then recursively store the subtrees rooted in the nodes at level $k + 1$, in a right-to-left order.

Choosing $k = \log B$, we obtain that following a search path P takes $P(1 + |P|/\log B)$ I/Os, each block is loaded using $O(1)$ I/Os and in each block we process $\Theta(\log B)$ nodes of the search path, except for possibly the last block loaded. Since the expected length of P is $O(\log n/H(\alpha))$, we obtain that the expected number of memory transfers is $O(1 + (1/H(\alpha)) \cdot \log_B n)$.

pqDFS. In a preprocessing phase, for each node v we assign its weight $w(v)$ as the number of nodes contained by the subtree rooted at v. Given a parameter p, we first store consecutively the p heaviest nodes in decreasing order with respect to their weights. The subtrees rooted at the children of the nodes on the frontier, if any, are then recursively stored in decreasing order of their weights. If two or more nodes have the same weight, no assumption can be made with respect to the order in which they will be stored. To implement this layout we use a priority queue, hence its name.

To optimize the number of memory transfers, we choose $p = \Theta(B)$ and thus the group of the p heavy nodes is stored in $O(1)$ memory blocks. For the children of the frontier of a group of p nodes the ratio between the weights of the lightest and heaviest children is at least α (for $0 \leq \alpha \leq 1/2$). This implies that each subtree in the frontier of the group has at most a fraction of $1/(B\alpha + 1)$ of the size of the subtree rooted at the root of the group. It follows that a search uses $O(\log_{B\alpha+1} n)$ I/Os, which is $O(\log_B n)$ for a constant $\alpha > 0$.

Skewed van Emde Boas. This layout is a variation of the van Emde Boas layout, which is known to match in the cache-oblivious model, i.e. where the parameters M and B are not known, the best bounds known for searching in the I/O-model. Given a node v and a tree, the weight of the node is given by the number of nodes in the subtree rooted in v. Given a tree of n nodes, we split it into a top subtree containing $\lceil\sqrt{n}\rceil$ nodes and $O(\sqrt{n})$ bottom subtrees. The top subtree contains the nodes with the highest weights and the bottom subtrees have as roots the children of the leaves of the top subtree. After the splitting phase, the top and the botttom subtrees are recursively stored in consecutive memory locations.

Since the top subtree contains the heaviest $\lceil\sqrt{n}\rceil$) nodes, by a similar argument to pqDFS the ratio between the weights of the lightest and heaviest root of the bottom subtrees is at least α (for $0 \leq \alpha \leq 1/2$). If the root of the tree has weight n, we obtain that the number of nodes in each of the bottom subtrees is at most $n/(\alpha\sqrt{n}+1)$ nodes. In the recursive layout, when $n = \Theta(B)$ searching in the corresponding subtree takes $O(1)$ I/Os. We obtain that a search takes $O(\log_{B\alpha+1} n)$ I/Os, which is $O(\log_B n)$ when α is constant.

7.5 Experimental setup

We analyze how the skewness factor α of the binary tree affects the running time in practice for the different layouts. To avoid additional costs inflicted over the running time by recursive calls, we use the iterative searching procedure in Figure 7.2. We generate a large sequence of random successful queries and measure the running time together with the number of comparisons, the number of branch mispredictions and the L1 data cache misses performed. We conduct our experiments on two standard Linux machines, having two different architectures. One of them has a P4 3.4 GHz CPU and 1 GB RAM, running linux 2.6.10. The other one has an AMD Athlon XP 2400+ 2.0 GHz CPU with 1GB RAM, running linux 2.6.8.1. To count the number of branch mispredictions and L1 data cache misses we use the PAPI 3.0 library. The code is compiled with gcc 3.3.2 using optimization level -O3. We will restrict ourselves to showing in the empirical results for AMD architecture. For the Pentium 4 processor the same behavior was observed as for the AMD architecture. The source code together with the scripts running the experiments and the plotted resulting data are available at `www.daimi.au.dk/~gabi/esa06.tar.gz`.

7.6 Experimental results

We demonstrate experimentally that in practice the skewed binary search trees can outperform the theoretically better balanced binary search trees, because of the different costs for branching left or right.

Since the number of branch mispredictions and the amount of computation (i.e. the number of comparisons) are independent on the memory layout, we can count them on any layout. The charts in Figure 7.3 are obtained by counting the number of comparisons (left) and the number of branch mispredictions

```
while(root!=NULLV)
{
if(key==t[root].key)
   return root;
if(key>t[root].key)
   root=t[root].right;
else
   root=t[root].left;
}
```

Figure 7.2: An iterative C source code for searching.

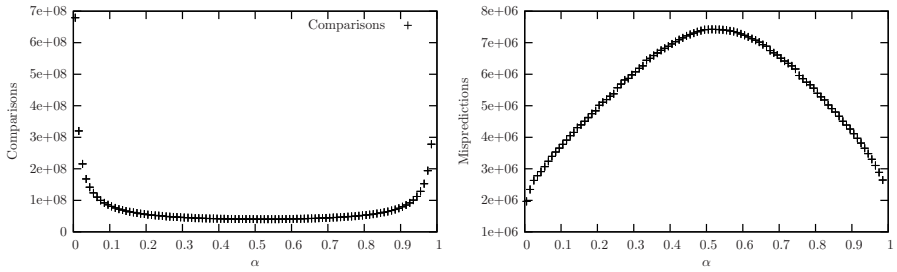

Figure 7.3: The number of comparisons (left) and branch mispredictions (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries.

(right) for a tree of 25×10^3 items and 10^6 queries. As expected, the number of comparisons achieves a minimum for perfectly balanced trees, i.e. for $\alpha \approx 0.5$, and increases with the skewness of the tree. In what concerns branch mispredictions, their number increases by a factor of 350% when decreasing the skewness, following the expectation in Theorem 7.2. Intuitively, this happens because the more nodes one of the subtrees rooted at the children of a given node has, the more likely is that a random search path will contain that child, hence the more likely the searching conditional branch will be correctly predicted. We observe that the number of branch mispredictions has a maximum for $\alpha \approx 0.52$ and that for very high values of α the number of branch mispredictions is greater by about 25% than for very low values. This is because of the rounding for small instances, i.e. the number in the left subtree is $\lfloor \alpha n \rfloor$ which yields a rightmost path for $\alpha n < 1$.

As previously stated, the number of cache faults performed for a random search depends not only on the skewness factor α, but also on the memory layout of the tree. We first analyze the layouts that do not use blocking, that is DFSl, DFSr, BFS, Inord and Rand. In Figure 7.4 we give the running time (left) and the number of cache misses (right) performed by 10^6 queries in a skewed search tree of 25×10^3 nodes. As expected, the Rand layout achieves the worst running time, since it performs one cache fault for each element on a given path. Inord and BFS achieve competitive running times, whereas DFSl and DFSr are best layouts that do not use blocking, with respect to both running time and cache misses performed. We note that the Inord layout

performs less cache faults and achieves better running times than BFS for very skewed trees, i.e. very small or very large values of α, whereas when the trees are almost balanced BFS outperforms Inord. Also, it is expected that DFSl and DFSr have symmetric charts for the number of cache misses and implicitly the running time, since they are symmetric layouts, where DFSr is efficient for $\alpha < 0.5$, since there are more nodes in the right subtree, and DFSl is more efficient for $\alpha > 0.5$. We recall that in the case of DFSr, since the right child is recursively stored after the root, branching right takes $O(1/B)$ I/Os whereas we spend $O(1)$ I/Os for branching left, whereas in the case of DFSl we spend $O(1/B)$ I/Os for branching left and $O(1)$ I/Os for branching right. We note that the minimum running time is achieved for $\alpha \approx 0.2$ in the case of DFSr and for $\alpha \approx 0.75$, and is better by around 15% compared to $\alpha = 0.5$. In DFSr, for $0.2 < \alpha \leq 0.5$, even though less comparisons are performed, both cache faults and branch mispredictions increase and the overall running time increases too.

We now analyze the blocked layouts. We conduct experiments for tuning the parameterized layouts, i.e. k-level grouping and pqDFS. Again, we perform 10^6 queries on a skewed search tree of 25×10^3 nodes, for different values of the parameters. For k-level grouping, we give experimental data for different values of the parameter k, i.e. the number of levels grouped together in the layout, for different values of α. For each pair of values for k and α, we perform three series of queries and select the median of the running times. For each value of the parameter k we choose the smallest running time among the different possible skewness factors. The data we obtained is shown in Figure 7.5 (left). The differences in the running times are up to 5%, and the minimum running time is achieved for $k = 2$, i.e. when two levels of the tree are grouped together. Thus, in our further experiments involving this layout we use this value.

We perform the same experiments for the pqDFS layout, varying the number p of the heaviest nodes grouped in a block, see Figure 7.5 (right). Unlike the k-level grouping, in this case the differences in the running times are very small. Since the minimum running time was obtained when grouping $p \approx 40$ nodes together, in the further experiments we are using $p = 40$.

We perform a comparative study for the blocked layouts, i.e. k-level grouping, pqDFS, and skewed van Emde Boas, together with DFSr, since it is the non-blocked layout that achieved the best running time. In Figure 7.6 we show the running times (left) and the number of cache misses (right) performed for these layouts on a skewed binary search tree of 25×10^3 nodes for 10^6 queries. We note that even though all layouts achieve approximately the same running times, at all times the skewed van Emde Boas is the fastest. The heuristics of grouping the heavy nodes achieves good results in practice, since pqDFS is faster than blocking k levels (bDFS). Finally, we note that DFSr is slightly slower than the blocked layouts. In what concerns the data cache misses, for all the algorithms the number of data cache misses is almost similar and is approximately the same regardless of the skewness factor for $\alpha < 0.5$, except for the case when the tree is extremely skewed, i.e. for very small values of α. We note when increasing the skewness factor α up to 0.5, the number of comparisons decreases, the number of cache misses is approximately the same except for extremely low values of α, whereas the number of branch mispredic-

tions is increasing. The resulting effect is that the minimum running time is achieved for $\alpha \approx 0.3$, and is better by a factor of 5% compared to the perfectly balanced search trees for all the blocked layouts. As stated before, for DFSr, the observed improvement in the running time is up to 15%. In what concerns the number of caches, the blocked layouts performed much better than the non-blocked layouts, as the skewed van Emde Boas and pqDFS layouts achieve significant improvements against BFS, Inord and Rand.

Finally, we study for which values of the skewness factor α we achieve the minimum running time when varying the size of the tree. We choose to perform our experiments on two of the layouts that achieved the best running times, namely pqDFS and the skewed van Emde Boas. For a given tree size, we vary the skewness factor α and for each value of α we perform three series of 10^6 queries and pick the median of the running times. We then measure the skewness factor for which the minimum running time was achieved. In Figure 7.7, we show the resulting data for both the AMD (left) and P4 (right) architectures. We notice that for both architectures the pqDFS achieves its best running time for smaller values of α than skewed van Emde Boas. Also, the best skewness factor is increasing while increasing the input size in the case of the AMD architecture, whereas for the P4 it has a constant behavior when increasing the input size.

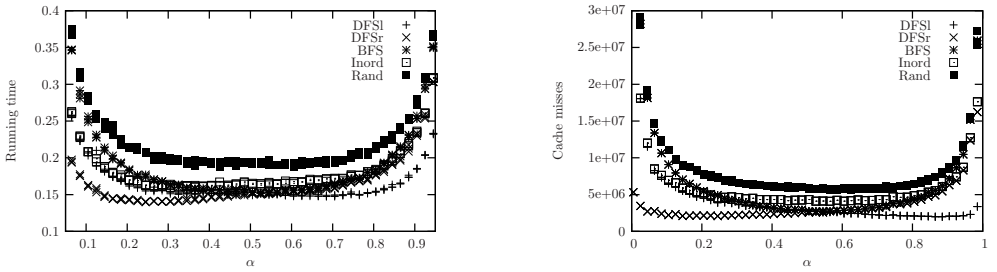

Figure 7.4: The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 items for 10^6 queries for the non-blocked layouts.

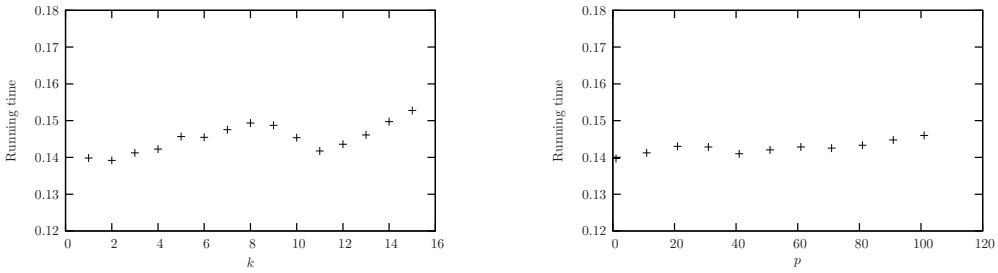

Figure 7.5: The best running times for k-level grouping (left) and pqDFS where p nodes are grouped together (right), for 10^6 queries and a skewed search tree of 25×10^3 nodes.

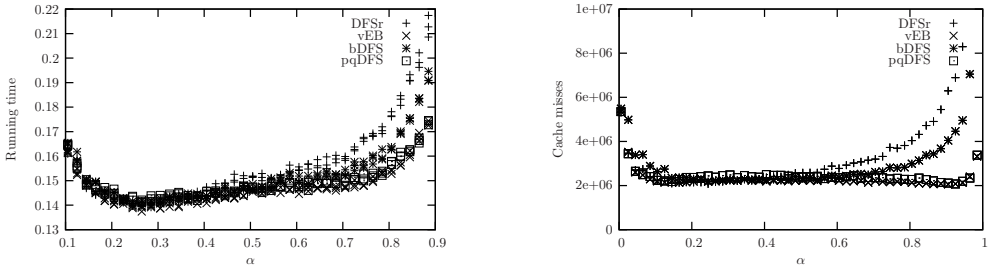

Figure 7.6: The running time (left) and the number of L1 data cache misses (right) performed by a skewed search tree of 25×10^3 nodes for 10^6 queries for the blocked layouts.

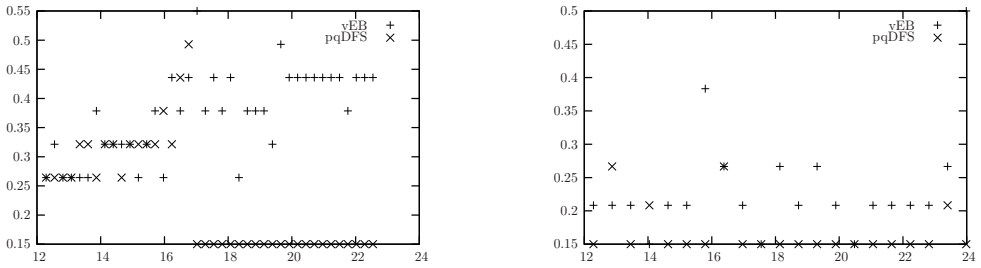

Figure 7.7: The skewness factors that achieved the minimum running times for different tree sizes.

Chapter 8

Adapting parallel algorithms to the W-Stream model

It is refreshing to see work on streaming moving from specific problems to a more established "theory" of general results.
— Anonymous reviewer

In this chapter we introduce the results in [36]. In Section 8.1 we show how to turn parallel algorithms into efficient algorithm in the W-Stream model. We first apply our simulations to give sorting algorithms achieving optimal (up to poly-log factors) trade-offs between passes and space in Section 8.2. In Section 8.3 we use our simulations to obtain similar results for several graph problems, such as connected components, minimum spanning tree, biconnected components, and maximal independent set. For some of these problems we also propose improved ad-hoc algorithms. Finally, in Section 8.4 we discuss the limitations of our approach.

8.1 Simulating parallel algorithms in W-Stream

In this section we show general techniques for simulating parallel algorithms in W-Stream. We show in the next sections that our techniques yield near-optimal algorithms for many classical combinatorial problems in the W-Stream model. In Theorem 8.1 we discuss how to simulate general CRCW PRAM algorithms.

Theorem 8.1 *Let A be a* PRAM *algorithm that uses N processors and runs in time T using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof (Sketch). In the PRAM model, at each parallel round, every processor may read $O(1)$ memory cells, perform $O(1)$ instructions to update its internal state, and write $O(1)$ memory cells. We assume that each memory address, cell value, and processor state takes $O(\log M)$ bits. A round of A can be simulated in W-Stream by performing $O((N \log M)/s)$ passes, where at each pass we simulate the execution of $\Theta(s/\log M)$ processors using s bits of working memory.

The content of the memory cells accessed by the algorithm and the state of each processor are maintained on the intermediate streams. We simulate the task of each processor in a constant number of passes as follows. We first read from the input stream its state and the content of the $O(1)$ memory cells used by A and then we execute the $O(1)$ instructions performed. Finally, we write to the output stream the new state and possibly the values of the $O(1)$ output cells. Memory cells that remain unchanged are simply propagated through the intermediate streams by just copying them from the input stream to the output stream at each pass. □

There are many examples of problems that can be solved near-optimally in W-Stream using Theorem 8.1. For instance, solving list ranking in PRAM takes $O(\log n)$ rounds and $O(n/\log n)$ processors [7], where n is the length of the list. By Theorem 8.1, we obtain a W-Stream algorithm that runs in $O((n \log n)/s)$ passes. An Euler tour of a tree with n vertices is computed in parallel in $O(1)$ rounds using $O(n)$ processors [68], which by Theorem 8.1 yields again a $p = O((n \log n)/s)$ bound in W-Stream. However, for other problems, the bounds obtained this way are far from being optimal. For instance, efficient PRAM algorithms for graph problems typically require $O(m + n)$ processors, where n is the number of vertices, and m is the number of edges. For these problems, Theorem 8.1 yields bounds of the form $p = O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds are known for many of them.

In Definition 8.1 we introduce RPRAM as an extension of the PRAM model. It allows every processor to handle in a parallel round not only $O(1)$ memory cells, but an arbitrary number of cells. Since in W-Stream a value in the working memory might be processed against all the data in the stream, we view RPRAM as a natural link between PRAM and W-Stream, even though it may be unrealistic in a practical setting. We first introduce a generic simulation that turns RPRAM algorithms into W-Stream algorithms. We then give RPRAM implementations that lead to efficient algorithms in W-Stream for a number of problems where the PRAM simulation in Theorem 8.1 does not yield good results.

Definition 8.1 *An* RPRAM *(Relaxed* PRAM*) is an extended CRCW* PRAM *machine with N processors and memory of size M where at each round a processor can execute $O(M)$ instructions that:*

- *can read an arbitrary number of memory cells. Each cell can only be read a constant number of times, and no assumptions can be made as to the order in which values are given to the processor;*
- *can write an arbitrary subset of the memory cells. The result of concurrent writes to the same cell by different processors in the same round is undefined. Writing can only be performed after all read operations have been done.*

Similarly to a PRAM*, each processor has a constant number of registers of size $O(\log M)$ bits.*

The jump in computational power provided by RPRAM allows substantial improvements for many classical PRAM algorithms such as decreasing the number of parallel rounds while preserving the number of processors or reducing the number of processors used while maintaining the same number of parallel rounds. We show in Theorem 8.2 that parallel algorithms implemented in this more powerful model can be simulated in W-Stream within the same bounds of Theorem 8.1.

Theorem 8.2 *Let A be an* RPRAM *algorithm that uses N processors and runs in time T using space $M = \text{poly}(N)$. Then A can be simulated in* W-Stream *in $p = O((T \cdot N \cdot \log M)/s)$ passes using s bits of working memory and intermediate streams of size $O(M + N)$.*

Proof (Sketch). We follow the proof of Theorem 8.1. The main difference is that a processor in the RPRAM model can read and write an arbitrary number of memory cells at each round, executing many instructions while still using $O(\log M)$ bits to maintain its internal state. Since the instructions of algorithm A performed by a processor during a round do not assume any particular order for reading the memory cells, reading memory values from the input stream can still be simulated in one pass. Replacing cell values read from the input stream with the new values written on the output stream can be performed in one additional pass. □

8.2 Sorting

As a first simple application of the simulation techniques introduced in Section 8.1, we show how to derive efficient sorting algorithms in W-Stream. We first recall that n items can be sorted on a PRAM with $O(n)$ processors in $O(\log n)$ parallel rounds and $O(n \log n)$ comparisons [68]. By Theorem 8.1, this yields a W-Stream sorting algorithm that runs in $p = O((n \log^2 n)/s)$ passes. In RPRAM, however, sorting can be solved by $O(n)$ processors in constant time as follows. Each processor is assigned to an input item; in one parallel round it scans the entire memory and counts the numbers i and j of items smaller than and equal to the item the processor is assigned to respectively. Then each processor writes its own item into all the cells with indices between $i + 1$ and $i + 1 + j$, and thus we obtain a sorted sequence.

Theorem 8.3 *Sorting n items in* RPRAM *can be done in $O(1)$ parallel rounds using $O(n)$ processors.*

Using the simulation in Theorem 8.2, we obtain the result stated below.

Corollary 8.1 *Sorting n items in* W-Stream *can be performed in $O(n \log n/s)$ passes.*

We obtain a W-Stream sorting algorithm that takes $p = O((n \log n)/s)$ passes, thus matching the performance of the best known algorithm for sorting in a streaming setting [87]. Since sorting requires $p = \Omega(n/s)$ passes in

W-Stream, this bound is essentially optimal. However, both our algorithm and the algorithm in [87] perform $O(n^2)$ comparisons. We reduce the number of comparisons to the optimal $O(n \log n)$ at the expense of increasing the number of passes to $O((n \log^2 n)/s)$ by simulating an optimal PRAM algorithm via Theorem 8.1, as stated before.

8.3 Graph problems

In this section we discuss how to derive efficient W-Stream algorithms for several graph problems using the RPRAM simulation in Theorem 8.2. Since efficient PRAM graph algorithms typically require $O(m+n)$ processors on graphs with n vertices and m edges [15], simulating such algorithms in W-Stream using Theorem 8.1 yields bounds of the form $p = O((m \cdot \text{polylog } n)/s)$, while $p = \Omega(n/s)$ almost-tight lower bounds in W-Stream are known for many of them. Graph connectivity is one prominent example [37]. Notice that, assigning each vertex to a processor, RPRAM gives enough power for each vertex to scan its entire neighborhood in a single parallel round. Since many parallel graph algorithms can be implemented using repeated neighborhood scanning, in many cases this allows us to reduce the number of processors from $O(m+n)$ to $O(n)$ while maintaining the same running time. By Theorem 8.2, this yields improved bounds of the form $p = O((n \cdot \text{polylog } n)/s)$.

8.3.1 Connected components (CC)

A classical PRAM random-mating algorithm for computing the connected components of a graph with n vertices and m edges uses $O(m+n)$ processors and runs in $O(\log n)$ time with high probability [15,100]. We first describe the algorithm and then we give a RPRAM implementation that uses only $O(n)$ processors which, by Theorem 8.2, leads to a nearly optimal algorithm in W-Stream.

PRAM algorithm. The algorithm is based on building a set of star subgraphs and contracting the stars. It each parallel round it performs the following sequence of steps.

1. Each vertex is assigned the status of parent or child independently with probability 1/2;

2. For each child vertex u, determine whether it is adjacent to a parent vertex. If so, choose one such a vertex to be the parent $f(u)$ of u, and replace each edge (u, v) by $(f(u), v)$ and each edge (v, u) by $(f(v), u)$;

3. For each vertex having parent u, set the parent to $f(u)$.

The algorithm performs $O(\log n)$ parallel rounds with high probability [15].

RPRAM implementation. We show how to implement each parallel round in RPRAM in $O(1)$ rounds using only $O(n)$ processors. We attach a processor to each vertex. We first assign each vertex the status of parent or child, and then for each vertex we scan its neighborhood to find a parent, if there exists one (in case of several parents, we break ties arbitrarily). Updating the parents according to the third step also takes one round in RPRAM. We obtain the result in Theorem 8.4.

Theorem 8.4 *Solving CC in* RPRAM *takes $O(n)$ processors and $O(\log n)$ rounds with high probability.*

By Theorem 8.2, this yields the following bound in W-Stream.

Corollary 8.2 *CC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [37], this upper bound is optimal up to a polylogarithmic factor. We note however that CC can be solved deterministically in W-Stream in $O((n \log n)/s)$ passes [37].

8.3.2 Minimum spanning tree (MST)

In this section, we first describe the PRAM algorithm in [15] for computing the MST of an undirected graph. We then give a RPRAM implementation that leads to an optimal algorithm (up to a polylog factor) in W-Stream by using the simulation in Theorem 8.2. Finally, we give an algorithm designed in W-Stream that outperforms the algorithm obtained via simulation.

PRAM algorithm. The randomized CC algorithm previously introduced can be extended to find a minimum spanning tree in a (connected) graph [15]. It also takes $O(\log n)$ rounds with high probability and uses $O(m+n)$ processors. The algorithm is based on the property that given a subset V' of vertices, a minimum weight edge having one and only one endpoint in V' is in some MST. We modify the second step of the CC algorithm as follows. Each child vertex u determines the minimum weight incident edge (u, v). If v is a parent vertex, then we set $f(u) = v$ and flag the edge (u, v) as belonging to the spanning tree. This algorithm computes a MST and performs $O(\log n)$ rounds with high probability.

RPRAM implementation. The updated second step runs in $O(1)$ rounds in RPRAM and uses $O(n)$ processors. Since the implementations of the other steps of the CC algorithm are unchanged and take $O(1)$ rounds and $O(n)$ processors, we obtain the result stated in Theorem 8.5.

Theorem 8.5 *MST is solvable in* RPRAM *using $O(n)$ processors and $O(\log n)$ rounds with high probability.*

Assuming edge weights can be encoded using $O(\log n)$ bits, we obtain the following bound in W-Stream by Theorem 8.2.

Corollary 8.3 *MST can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes.*

We now give a deterministic algorithm designed directly in W-Stream that improves the bounds achieved by using the simulation.

A faster *ad hoc* W-Stream algorithm. We again assume edge weights can be encoded using $O(\log n)$ bits. We build the MST by progressively adding edges as follows. We compute for each vertex the minimum weight edge incident to it. This set of edges E' is added to the MST. We then compute the connected components induced by E' and contract the graph by considering each connected component a single vertex. We repeat these steps until the graph contains a single vertex or there are no more edges to add. More precisely, we consider at each iteration a contracted graph where the vertices are the connected components of the partial MST so far computed. Denoting $G_i = (V_i, E_i)$ the graph before the i^{th} iteration, the $(i+1)^{th}$ iteration consists of the following steps.

1. for each vertex $u \in V_i$, we compute a minimum weight edge (u, v) incident to u, and flag (u, v) as belonging to the MST (cycles that might occur due to weight ties are avoided by using a tie-breaking rule). Denote $E'_i = \{(u, v), u \in V_i\}$ the set of flagged edges.

2. we run a CC algorithm on the graph (V_i, E'_i). The resulted connected components are the vertices of V_{i+1}.

3. we replace each edge (u, v) by $(c(u), c(v))$, where $c(u)$ and $c(v)$ denote the labels of the connected components previously computed.

We now analyze the number of passes required in W-Stream. Let $|V_i| = n_i$. The first and the third steps require $O((n_i \log n)/s)$ passes each, since we can process in one pass $O(s/\log n)$ vertices. Computing the connected components also takes $O((n_i \log n)/s)$ passes, and therefore the i^{th} iteration requires $O((n_i \log n)/s)$ passes. We note that at each iteration we add an edge for every vertex in V_i and thus $|V_{i+1}| \leq |V_i|/2$, i.e. the number of connected components is divided by at least two. We obtain that the total number of passes performed in the worst case is given by $T(n) = T(n/2) + O((n \log n)/s)$, which sums up to $O((n \log n)/s)$.

Theorem 8.6 *MST can be computed in* $O((n \log n)/s)$ *passes in* W-Stream.

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [37], this upper bound is optimal up to a polylog factor. To the best of our knowledge, no previous algorithm was known for MST in W-Stream.

8.3.3 Biconnected components (BCC)

Tarjan and Vishkin [112] gave a PRAM algorithm that computes the biconnected components (BCC) of an undirected graph in $O(\log n)$ time using $O(m+$

n) processors. We give an RPRAM implementation of their algorithm that uses only $O(n)$ processors while preserving the time bounds and thus can be turned using Theorem 8.2 in a W-Stream algorithm that runs in $O((n \log^2 n)/s)$ passes. We also give a direct implementation that uses only $O((n \log n)/s)$ passes.

PRAM algorithm. Given a graph G, the algorithm considers a graph G' such that vertices in G' correspond to edges in G and connected components in G' correspond to biconnected components in G. The algorithm first computes a rooted spanning tree T of G and then builds a subgraph G'' of G' having as vertices all the edges of T. The edges of G'' are chosen such that two vertices are in the same connected component of G'' if and only if the corresponding edges in G are in the same biconnected component. After computing the connected components of G'' the algorithm appends the remaining edges of G to their corresponding biconnected components. We now briefly sketch the five steps of the algorithm.

1. build a rooted spanning tree T of G and compute for each vertex its preorder and postorder numbers together with the number of descendants. Also, label the vertices by their preorder numbers.
2. for each vertex u, compute two values, $low(u)$ and $high(u)$, as follows.

$$\begin{aligned} low(u) &= \min(\{u\} \cup \{low(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\}) \\ high(u) &= \max(\{u\} \cup \{high(w) | p(w) = u\} \cup \{w | (u, w) \in G \setminus T\}), \end{aligned}$$

   where $p(u)$ denotes the parent of vertex u.
3. add edges to G'' according to the following two rules. For all edges $(w, v) \in G \setminus T$ with $v + desc(v) \leq w$, add $((p(v), v), (p(w), w))$ to G'', and for all $(v, w) \in T$ with $p(w) = v$, $v \neq 1$, add $((p(v), v), (v, w))$ to G'' if $low(w) < v$ or $high(w) \geq v + desc(v)$, where $desc(v)$ denotes the number of descendants of vertex v.
4. compute the connected components of G''.
5. add the remaining edges of G to their biconnected components. Each edge $(v, w) \in G \setminus T$, with $v < w$, is assigned to the biconnected component of $(p(w), w)$.

RPRAM implementation. We give RPRAM descriptions for all the five steps of the algorithm, each of them using $O(\log n)$ time and $O(n)$ processors. First, we compute a spanning tree of the graph using the RPRAM algorithm previously introduced. Rooting the tree and computing for each vertex the preorder and postorder numbers as well as the number of descendants are performed using list ranking and Euler tour [112], which take $O(\log n)$ time and $O(n)$ processors in PRAM, and thus in RPRAM. Since the second step takes $O(\log n)$ time using $O(n)$ processors in PRAM [112], the same bounds hold for RPRAM. We implement the third step in RPRAM in constant time and $O(n)$ processors, since it suffices a scan of the neighborhood for each vertex. For computing the connected components of G'' in the fourth step, we use the

RPRAM algorithm previously introduced that takes $O(\log n)$ time and $O(n)$ processors. Finally, we implement the last step of the algorithm in RPRAM in $O(1)$ time and $O(n)$ processors by scanning the neighborhood for all vertices v and assigning the edges to the proper biconnected components. Since we implement all the steps of the algorithm in RPRAM in $O(\log n)$ rounds and $O(n)$ processors, we obtain the following result.

Theorem 8.7 *BCC is solvable in* RPRAM *using $O(n)$ processors in $O(\log n)$ rounds with high probability.*

By Theorem 8.2, this yields the following bound in W-Stream.

Corollary 8.4 *BCC can be solved in* W-Stream *in $O((n \log^2 n)/s)$ passes with high probability.*

We now show that we can achieve better bounds with an implementation designed directly in W-Stream.

A faster *ad hoc* W-Stream algorithm. We describe how to implement directly in W-Stream all the steps of the parallel algorithm of Tarjan and Vishkin [112]. Notice that we have given constant time RPRAM descriptions for the third and the fifth step, thus by applying the simulation in Theorem 8.2 we obtain W-Stream algorithms that run in $O((n \log n)/s)$ passes. For computing the connected components in the fourth step, we use the algorithm in [37] that requires $O((n \log n)/s)$ passes. Therefore, to achieve a global bound of $O((n \log n)/s)$ passes, it suffices to give implementations that run in $O((n \log n)/s)$ passes for the first two steps. For the first step, we can compute a spanning tree within the bound of Theorem 8.6. Rooting the tree and computing the preorder and postorder numbers together with the number of descendants can be implemented in $O((n \log n)/s)$ passes using list ranking, Euler tour and sorting. Concerning the second step, we compute the *low* and *high* values by processing $\Theta(s/\log n)$ vertices at each pass, according to the postorder numbers.

Theorem 8.8 *BCC can be solved in* W-Stream *in $O((n \log n)/s)$ passes in the worst case.*

By the $p = \Omega(n/s)$ lower bound for CC in W-Stream [37], this upper bound is optimal up to a polylog factor. To the best of our knowledge, no previous algorithm was known for BCC in W-Stream.

8.3.4 Maximal independent set (MIS)

We give an efficient RPRAM algorithm for the maximal independent set problem (MIS), based on the PRAM algorithm proposed by Luby [81]. Using the simulation in Theorem 8.2, this leads to an efficient W-Stream implementation.

PRAM algorithm. A maximal independent set S of a graph G is incrementally built through a series of iterations, where each iteration consists of a sequence of three steps, as follows. In the first step, we compute a random subset I of the vertices in G, by including each vertex v with probability $1/(2 \cdot deg(v))$. Then, for each edge (u, v) in G, with $u, v \in I$, we remove from I the vertex with the smallest degree. Finally, in the third step, we add to S the vertices in I, and then we remove from G the vertices in I together with their neighbors. The above steps are iterated until G gets empty. The algorithm uses $O(m + n)$ processors and $O(\log n)$ parallel rounds.

RPRAM implementation. We implement the first step of each iteration in constant time and $O(n)$ processors in RPRAM, since it requires each vertex to compute its own degree. The second step can also be implemented in constant time, by having each vertex in I scan its neighborhood, and remove itself upon encountering a neighbor also in I with a larger degree. Finally, we implement the third step in constant time as well by scanning the neighborhood of each vertex that is not in I, and removing it from G if at least one of its neighbors is in I. Since the algorithm performs $O(\log n)$ iterations with high probability [81], we obtain the bound in Theorem 8.9.

Theorem 8.9 *MIS can be solved in* RPRAM *using* $O(n)$ *processors in* $O(\log n)$ *rounds with high probability.*

By Theorem 8.2, this yields the following bound in W-Stream.

Corollary 8.5 *MIS can be solved in* W-Stream *in* $O((n \log^2 n)/s)$ *passes with high probability.*

We now prove lower bounds which show that the bound in Corollary 8.5 is optimal up to a polylogarithmic factor.

Theorem 8.10 *MIS requires* $\Omega(n/s)$ *passes in* W-Stream.

Proof (Sketch). The proof is based on a reduction from the bit vector disjointness communication complexity problem. Alice has an n-bit vector A and Bob has an n-bit vector B; they wish to know whether A and B are disjoint, i.e. $A \cdot B > 0$. They build a graph on $4n$ vertices v_i^j, where $i = 1, \cdots, n$ and $j = 1, \cdots, 4$. If $A_i = 0$, then Alice adds edges (v_i^1, v_i^2) and (v_i^3, v_i^4), whereas if $B_i = 0$, then Bob adds edges (v_i^1, v_i^3) and (v_i^2, v_i^4). The size of any MIS is $2n$ if $A \cdot B = 0$ and strictly greater otherwise. □

8.4 Limits of the RPRAM approach

In this section we prove that the increased power that RPRAM provides does not always help in reducing the number of processors to $O(n)$ and thus in obtaining W-Stream algorithms that run in $O((n \cdot \text{polylog } n)/s)$ passes. As an example, in Theorem 8.11 we prove that detecting cycles of length two in a graph takes $\Omega(m/s)$ passes.

Theorem 8.11 *Testing whether a directed graph with m edges contains a cycle of length two requires $p = \Omega(m/s)$ passes in* W-Stream.

Proof (Sketch). We prove the lower bound by showing a reduction from the bit vector disjointness two-party communication complexity problem. Alice has an m-bit vector A and Bob has an m-bit vector B; they wish to know whether A and B are disjoint, i.e. $A \cdot B > 0$. Alice creates a stream containing an edge $e(i) = (x_i, y_i)$ for each i such that $A[i] = 1$ and Bob creates a stream containing an edge $e^r(i) = (y_i, x_i)$ for each i such that $B[i] = 1$, where $x_i = i \text{ div } \lceil \sqrt{m} \rceil$ and $y_i = i \bmod \lceil \sqrt{m} \rceil$. Let G be the directed graph induced by the union of the edges in the streams created by Alice and Bob. Clearly, there is a cycle of length two in G if and only if $A \cdot B > 0$. Since solving bit vector disjointness requires transmitting $\Omega(m)$ bits [75], and the distributed execution of any streaming algorithm requires the working memory image to be sent back and forth from Alice to Bob at each pass, we obtain $s = \Omega(m)$, which leads to $p = \Omega(m/s)$. □

Testing whether a directed graph has a cycle of length two can be performed easily in one round in RPRAM using $O(m)$ processors, by just checking in parallel whether there is any edge (x, y) that also appears as (y, x) in the graph. This leads to an algorithm in W-Stream that runs in $O((m \log n)/s)$ passes by Theorem 8.2.

Chapter 9

Resilient Priority Queues

Nice result, achieved through the expert use of known techniques.
— Anonymous reviewer

This chapter is dedicated to presenting the results published in [70]. In Section 9.1 we define the resilient priority queue and introduce some notation. We give a detailed description of the resilient priority queue in Section 9.2, while in Section 9.3 we prove its correctness and complexity bounds. Finally, in Section 9.4 we prove matching lower bounds for resilient priority queues.

9.1 Preliminaries

In this section we define the resilient priority queue and introduce some notation used throughout this chapter.

Given two sequences X and Y, we let XY denote the *concatenation* of X and Y. A sequence X is *faithfully ordered* if its uncorrupted keys appear in non-decreasing order. Finally, a *reliable value* is a value stored in unreliable memory which can be retrieved reliably in spite of possible corruptions. This is achieved by replicating the given value $2\delta + 1$ times. Retrieving a reliable value takes $O(\delta)$ time using the majority algorithm in [20], which scans the $2\delta + 1$ values keeping a single majority candidate and a counter in reliable memory.

Definition 9.1 *A resilient priority queue maintains a set of elements under the operations* Insert *and* Deletemin*. An* Insert *adds an element and a* Deletemin *deletes and returns the minimum uncorrupted element or a corrupted one.*

We note that our definition of a resilient priority queue is consistent with the resilient sorting algorithms introduced in [51]. Given a sequence of n elements, inserting all of them into a resilient priority queue followed by n Deletemin operations yields a faithfully ordered sequence.

9.2 Fault tolerant priority queue

In this section we introduce the resilient priority queue. It resembles the cache-oblivious priority queue by Arge *et al.* [9]. The elements are stored in faithfully

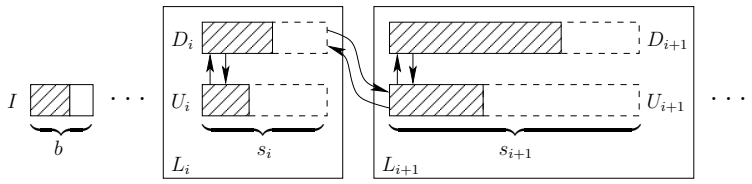

Figure 9.1: The structure of the priority queue. The buffers are stored in a doubly linked list using reliably stored pointers. Additionally, the size of each buffer is stored reliably.

ordered lists and are moved using two fundamental primitives, Push and Pull, based on faithful merging. We describe the structure of the priority queue in Section 9.2.1 and then introduce the Push and Pull primitives in Section 9.2.2. Finally, in Section 9.2.3, we describe the Insert and Deletemin operations.

9.2.1 Structure

The resilient priority queue consists of an insertion buffer I together with a number of layers $L_0, \ldots, L_k$, with $k = O(\log n)$. Each layer L_i contains an up-buffer U_i and a down-buffer D_i, represented as arrays. Intuitively, the up-buffers contain large elements that are on their way to the upper layers in the priority queue, whereas the down-buffers contain small elements, on their way to lower layers. The buffers in the priority queue are stored as a doubly linked list $U_0, D_0, \ldots, U_k, D_k$, see Figure 9.1. For each up and down buffer we reliably store the pointers to their adjacent buffers in the linked list and their size. In the reliable memory we store pointers to I, U_0 and D_0, together with $|I|$. Since the position of the first element in U_0 and D_0 is not always the first memory cell of the corresponding buffer, we also store the index of the first element in these buffers in reliable memory. The insertion buffer I contains up to $b = \delta + \log n + 1$ elements. For layer L_i we define the threshold s_i by $s_0 = 2 \cdot (\delta^2 + \log^2 n)$ and $s_i = 2s_{i-1} = 2^{i+1} \cdot (\delta^2 + \log^2 n)$, where n is the number of elements in the priority queue. We use these thresholds to decide whether an up buffer contains too many elements or whether a down buffer has too few. For the sake of simplicity, the up and down buffers are grown and shrunk as needed during the execution such that they don't use any extra space.

To structure the priority queue, we maintain the following invariants for the up and down buffers.

- *Order invariants:*
  1. All buffers are faithfully ordered.
  2. D_iD_{i+1} and D_iU_{i+1} are faithfully ordered, for $0 \leq i < k$.
- *Size invariants:*
  3. $s_i/2 \leq |D_i| \leq s_i$, for $0 \leq i < k$.

4. $|U_i| \leq s_i/2$, for $0 \leq i < k$.

By maintaining all the up and down buffers faithfully ordered, it is possible to move elements between neighboring layers efficiently, using faithful merging. By invariant 2, all uncorrupted elements in D_i are smaller than all uncorrupted elements in both D_{i+1} and U_{i+1}. This ensures that small elements belong to the lower layers of the priority queue. We note that there is no assumed relationship between the elements in the up and down buffers in the same layer. Finally, the size invariants allow the sizes of the buffers to vary within a large range. This way, $\Omega(s_i)$ INSERT or DELETEMIN operations occur between two operations on the same buffer in L_i, yielding the desired amortized bounds.

Since the s_i values depend on n, whenever the size of the priority queue increases or decreases by $\Theta(n)$, we perform a global rebuilding. This rebuilding is done by collecting all elements, sorting them with an optimal resilient sorting algorithm [49], and redistributing the output into the down buffers of all the layers starting with L_0. After the global rebuilding, the up buffers are empty and the down buffers full, except possibly the last down buffer.

9.2.2 Push and pull primitives

We now introduce the two fundamental primitives used by the priority queue. The PUSH primitive is invoked when an up buffer contains too many elements, breaking invariant 4. It "pushes" elements upwards, repairing the size invariants locally. The PULL operation is invoked when a down buffer contains too few elements, breaking invariant 3. It fills this down buffer by "pulling" elements from the layer above, again locally repairing the size invariants. Both operations faithfully merge consecutive buffers in the priority queue and redistribute the resulting sequence among the participating buffers. After merging, we deallocate the old buffers and allocate new arrays for the new buffers.

Push. The PUSH primitive is invoked when an up buffer U_i breaks invariant 4, *i.e.* when it contains more than $s_i/2$ elements. In this case we merge U_i, D_i and U_{i+1} into a sequence M using the resilient merging algorithm in [49]. We then distribute the elements in M by placing the first $|D_i| - \delta$ elements in a new buffer D'_i, and the remaining $|U_{i+1}| + |U_i| + \delta$ elements in a new buffer U'_{i+1}. After the merge, we create an empty buffer, U'_i, and deallocate the old buffers. If U'_{i+1} contains too many elements, breaking invariant 4, the PUSH primitive is invoked on U'_{i+1}. When L_i is the last layer, we fill D'_i with the first elements of M and create a new layer L_{i+1} placing the remaining elements of M into D'_{i+1} instead of U'_{i+1}. Since $|D'_i|$ is smaller than $|D_i|$, it could violate invariant 3. This situation is handled by using the PULL operation and is described after introducing PULL.

Unlike the priority queue in [9], the PUSH operation decreases the size of a down buffer. This is required to preserve invariant 2, in spite of corruptions. After a PUSH call, D'_i can contain elements from $U_i \cup U_{i+1}$. Since there is no assumed relationship between elements in $U_i \cup U_{i+1}$ and those in $D_{i+1} \cup U_{i+2}$, we need to ensure that each element in D'_i originating from $U_i \cup U_{i+1}$ is faithfully

smaller than the elements in $D_{i+1} \cup U_{i+2}$. Assume the size of D_i is preserved, *i.e.* $|D'_i| = |D_i|$. Consider a corruption that alters an element in D_i to some large value before the PUSH. This corrupted value could be placed in U'_{i+1} and, since $|D'_i| = |D_i|$, an element from $U_i \cup U_{i+1}$ must be placed in D'_i. This new element in D'_i potentially violates invariant 2.

Pull. The PULL operation is called on a down buffer D_i when it contains less than $s_i/2$ elements, breaking invariant 3. In this case, the buffers D_i, U_{i+1}, and D_{i+1} are merged into a sequence M using the resilient merging algorithm in [49]. The first s_i elements from M are written to a new buffer D'_i, and the next $|D_{i+1}| - (s_i - |D_i|) - \delta$ elements are written to D'_{i+1}. The remaining elements of M are written to U'_{i+1}. A PULL is invoked on D'_{i+1}, if it is too small.

Similar to the PUSH operation, the extra δ elements lost by D_{i+1} ensure that the order invariants hold in spite of possible corruptions. That is, a corruption of an element in $D_i \cup D_{i+1}$ to a very large value may cause an element from U_{i+1} to take the place of the corrupted element in D'_{i+1} and this element is possibly larger than some uncorrupted element in $D_{i+2} \cup U_{i+2}$.

After the merge, U'_{i+1} contains δ more elements than U_{i+1} had before the merge, and thus it is possible that it has too many elements, breaking invariant 4. We handle this situation as follows. Consider a maximal series of subsequent PULL invocations on down buffers $D_i, D_{i+1}, \ldots, D_j$, $0 \leq i < j < k$. After the first PULL call on D_i and before the call on D_{i+1} we store a pointer to D_i in the reliable memory. After all the PULL calls we investigate all the affected up buffers, by simply following the pointers between the buffers starting from D_i, and invoke the PUSH primitive wherever necessary. The case when PUSH operations cause down buffers to underflow is handled similarly.

9.2.3 Insert and deletemin

An element is inserted in the priority queue by simply appending it to the insertion buffer I. If I gets full, its elements are added to U_0 by first faithfully sorting I and then faithfully merging I and U_0. If U_0 breaks invariant 4, we invoke the PUSH primitive. If L_0 is the only layer of the priority queue and D_0 violates the size constraint, we faithfully merge the elements in I with D_0 instead.

To delete the minimum element in the priority queue, we first find the minimum of the first $\delta + 1$ values in D_0, the minimum of the first $\delta + 1$ values in U_0, and the minimum element in I. We then take the minimum of these three elements, delete it from the appropriate buffer and return it. After deleting the minimum, we right-shift all the elements in the affected buffer from the beginning up to the position of the minimum. This way we ensure that elements in any buffer are stored consecutively. If D_0 underflows, we invoke the PULL primitive on D_0, unless L_0 is the only layer in the priority queue. If U_0 or D_0 contains $\Theta(\log n + \delta)$ empty cells, we create a new buffer and copy the elements from the old buffer to the new one.

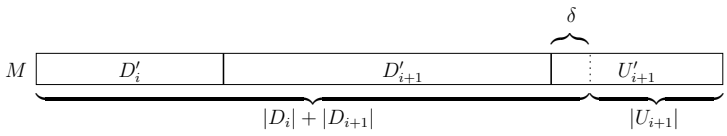

Figure 9.2: The distribution of M into buffers.

9.3 Analysis

In this section we analyze the resilient priority queue. We prove the correctness in Section 9.3.1 and analyze the time and space complexity in Section 9.3.2.

9.3.1 Correctness

To prove correctness of the resilient priority queue, we need to show that the DELETEMIN operation returns the minimum uncorrupted value or a corrupted value. We first prove that the PULL and PUSH operations preserve the order invariants.

Lemma 9.1 *The* PULL *and* PUSH *primitives preserve the order invariants.*

Proof. Recall that in a PULL invocation on buffer D_i, the buffers D_i, U_{i+1}, and D_{i+1} are faithfully merged into a sequence M. The elements in M are then distributed into three new buffers D'_i, U'_{i+1}, and D'_{i+1}, see Figure 9.2. To argue that the order invariants are satisfied we need to show that the elements of the down buffer on layer L_j, for $0 \leq j < k$, are faithfully smaller than the elements of the buffers on layer L_{j+1}, where k is the index of the last layer. The invariants hold trivially for unaffected buffers. The faithful merge guarantees that $D'_iD'_{i+1}$ as well as $D'_iU'_{i+1}$ are faithfully ordered, and thus the individual buffers are also faithfully ordered. Since invariant 2 holds for the original buffers all uncorrupted elements in D_{i+1} and U_{i+1} are larger than the uncorrupted elements in D_i, guaranteeing that $D_{i-1}D'_i$ is faithfully ordered. Finally, we now show that $D_{i+1}D_{i+2}$ and $D_{i+1}U_{i+2}$ are faithfully ordered.

Let m be the minimum uncorrupted element in $D_{i+2} \cup U_{i+2}$. We need to show that all uncorrupted elements in D'_{i+1} are smaller than m. If no uncorrupted element from U_{i+1} is placed in D'_{i+1}, the invariant holds by the order invariants before the operation. Otherwise, assume that an uncorrupted element $y \in U_{i+1}$ is moved to D'_{i+1}. Since $|U'_{i+1}| = |U_{i+1}| + \delta$ and y is moved to D'_{i+1}, at least $\delta + 1$ elements originating from $D_i \cup D_{i+1}$ are contained in U'_{i+1}. Since there can be at most δ corruptions, there exists at least one uncorrupted element, x, among these. By faithful merging, all uncorrupted elements in D'_{i+1} are smaller than x, which means that $y \leq x$. Since x originates from $D_i \cup D_{i+1}$, it is smaller than m. We obtain $y \leq m$.

A similar argument proves correctness of the PUSH operation. We conclude that both order invariants are preserved by PULL and PUSH operations.

□

Having proved that the order invariants are maintained at all times, we now prove the correctness of the resilient priority queue.

Lemma 9.2 *The* DELETEMIN *operation returns the minimum uncorrupted element in the priority queue or a corrupted element.*

Proof. We recall that the DELETEMIN operation computes the minimum of the first $\delta + 1$ elements of U_0 and D_0. It compares these values with the minimum of I, found in a scan, and returns the smallest of these elements. Since U_0 and D_0 are faithfully ordered, the minimum of their first $\delta + 1$ elements is either the minimum uncorrupted value in these buffers, or a corrupted value even smaller. Furthermore, according to the order invariants, all the values in layers $L_1, \ldots, L_k$ are faithfully larger than the minimum in D_0. Therefore, the element reported by DELETEMIN is the minimum uncorrupted value or a corrupted value. □

9.3.2 Complexity

In this section we show that our resilient priority queue uses $O(n)$ space and that INSERT and DELETEMIN take $O(\log n + \delta)$ amortized time. We first prove that the PULL and PUSH primitives restore the size invariants.

Lemma 9.3 *If a size invariant is broken for a buffer in L_0, invoking* PULL *or* PUSH *on that buffer restores the invariants. Furthermore, during this operation* PULL *and* PUSH *are invoked on the same buffer at most once. No other invariants are broken before or after this operation.*

Proof. Assume that PUSH is invoked on U_0, and that it is called iteratively up to some layer L_l. By construction of PUSH, the size invariants for all the up buffers now hold. Since a PUSH steals δ elements from the down buffers, the layers $L_0, \ldots, L_l$ are traversed again and PULL is invoked on these as needed. The last of these PULL operations might proceed past layer L_l. Similarly, a PULL may cause an up buffer to overflow. However, since the cascading PUSH operations left $|U_i| = 0$ for $i \leq l$, any new PUSH are invoked on up buffers only on layer L_{l+1} or higher, thus PUSH is invoked on each buffer at most once. A similar argument works for the PULL operation.

□

Lemma 9.4 *The resilient priority queue uses $O(n + \delta)$ space to store n elements.*

Proof. The insertion buffer always uses $O(\log n + \delta)$ space. We prove that the remaining layers use $O(n)$ space. For each layer we use $O(\delta)$ space for storing structural information reliably. In all layers, except the last one, the down buffer contains $\Omega(\delta^2)$ elements by invariant 3. This means that for each of these layers the elements stored in the down buffer dominate the space complexity. The structural information of the last layer requires additional $O(\delta)$ space. □

The space complexity of the priority queue can be reduced to $O(n)$ without affecting the time complexity, by storing the structural information of L_0 in safe memory, and by doubling or halving the insertion buffer during the lifetime of the algorithm such that it always uses $O(|I|)$ space.

Lemma 9.5 *Each* INSERT *and* DELETEMIN *takes* $O(\log n + \delta)$ *amortized time.*

Proof. We define the following potential function:

$$\Phi = \sum_{i=1}^{k} (c_1 \cdot (\log n - i) \cdot |U_i| + c_2 \cdot i \cdot |D_i|) \ .$$

We use Φ to analyze the amortized cost of a PUSH operation. In a PUSH operation on U_i, buffers U_i, D_i, and U_{i+1} are merged. The elements are then distributed into new buffers U_i', D_i', and U_{i+1}', such that $|U_i'| = 0$, $|D_i'| = |D_i| - \delta$, and $|U_{i+1}'| = |U_{i+1}| + |U_i| + \delta$. This gives the following change in potential $\Delta\Phi$:

$$\begin{aligned} \Delta\Phi &= -|U_i| \cdot c_1 \cdot (\log n - i) - \delta \cdot c_2 \cdot i + (|U_i| + \delta) \cdot c_1(\log n - (i+1)) \\ &= -c_1 \cdot |U_i| + \delta(-c_2 \cdot i + c_1 \cdot \log n - c_1 \cdot i - c_1) \ . \end{aligned}$$

Since the PUSH is invoked on U_i, invariant 4 is not valid for U_i and therefore $|U_i| \geq \frac{s_i}{2} = 2^i \ (\log^2 \ n \ + \ \delta^2)$. Thus:

$$\Delta\Phi \leq -c_1 \cdot |U_i| + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot 2^i \cdot (\log^2 n + \delta^2) + c_1 \cdot \delta \cdot \log n \leq -c_1 \cdot c' \cdot |U_i| \ , \tag{9.1}$$

for some constant $c' > 0$.

Since faithfully merging two sequences of size n takes $O(n + \delta^2)$ time [49], the time used for a PUSH on U_i is upper bounded by $c_m \cdot (|U_i| + |D_i| + |U_{i+1}| + \delta^2)$, where c_m depends on the resilient merge. This includes the time required for retrieving reliably stored variables. Adding the time and the change in potential we are able to get the amortized cost less than zero by tweaking c_1 based on equation (9.1). This is because $|U_i|$ is $\Omega(\delta^2)$ and at most a constant fraction smaller than the participants in the merge.

A similar analysis works for the PULL primitive. We now calculate the amortized cost of INSERT and DELETEMIN. We ignore any PUSH or PULL operations since their amortized costs are negative. The amortized time for inserting an element in I, sorting I, and merging it with U_0 is $O(\log n + \delta)$ per operation. The change in potential when adding elements to L_0 is $O(\log n)$ per element. The time needed to find the smallest element in a DELETEMIN is $O(\log n + \delta)$, and the change in potential when an element is deleted from L_0 is negative.

The cost of the global rebuilding is dominated by the cost of sorting, which is $O(n \log n + \delta^2)$. There are $\Omega(n)$ operations between each rebuild, which leads to $O(\log n + \delta)$ time per operation, since $\delta \leq n$. We conclude that each INSERT and DELETEMIN takes $O(\log n + \delta)$ amortized time. □

Theorem 9.1 *The resilient priority queue takes* $O(n)$ *space and uses amortized* $O(\log n + \delta)$ *time per operation.*

9.4 Lower bound

In this section we prove that any resilient priority queue takes $\Omega(\log n + \delta)$ time for either INSERT or DELETEMIN in the comparison model, under the assumption that no elements are stored in reliable memory between operations. This implies optimality of our resilient priority queue under these assumptions. We note that the reliable memory may contain any structural information, e.g. pointers, sizes, indices.

Theorem 9.2 *A resilient priority queue containing n elements, with $n > \delta$, uses $\Omega(\log n + \delta)$ comparisons to perform* INSERT *followed by* DELETEMIN.

Proof. Consider a priority queue Q with n elements, with $n > \delta$, that uses less than δ comparisons for an INSERT followed by a DELETEMIN. Also, Q does not store elements in reliable memory between operations. Assume that no corruptions have occurred so far. Without loss of generality we assume that all the elements in Q are distinct. We prove there exists a series of corruptions C, $|C| \leq \delta$, such that the result of an INSERT of an element e followed by a DELETEMIN returns the same element regardless of the choice of e.

Let $k < \delta$ be the number of comparisons performed by Q during the two operations. We force the result of each comparison to be the same regardless of e by suitable corruptions. In all the comparisons involving e, we ensure that e is the smallest. We do so by corrupting the value which e is compared against if necessary, by adding some positive constant $c \geq e$ to the other value. If two elements different than e are compared, we make sure the outcome is the same as if no corruptions had happened. If one of them was corrupted, adding c to the other one reestablishes their previous ordering. If both of them were corrupted by adding c, their ordering is unchanged and no corruptions are needed. Forcing any comparison to give the desired outcome requires at most one corruption, and therefore $|C| \leq k < \delta$.

We now consider the value e' returned by DELETEMIN on Q. If $e = e'$ then we choose e to be larger than some element $x \in Q$ not affected by a corruption in C. Such a value exists because the size of the priority queue is larger than δ. Since $e = e' > x$, Q returned an uncorrupted element that was not the minimum uncorrupted element in Q. If $e \neq e'$ we choose e to be smaller than any element in Q. With such a choice of e, no corruptions are required and the value returned by Q was not corrupted, but still larger than e. This proves Q is not resilient.

Adding the classical $\Omega(\log n)$ bound for priority queues in the comparison model the result follows. □

Chapter 10

Optimal Resilient Dynamic Dictionaries

The paper is very well written, contains a good motivation, good area overview, all proof are sound, complete and easy to understand.
— Anonymous reviewer

This chapter is devoted to presenting the results published as [27], which has been accepted for publication as a merged paper [24]. The remainder of the chapter is structured as follows. First we introduce the randomized static dictionary in Section 10.1, and then the deterministic static dictionary in Section 10.2. Finally, in Section 10.3 we present the dynamic dictionary.

10.1 Optimal randomized static dictionary

In this section we introduce a simple randomized resilient search algorithm. It searches for a given element in a sorted array using worst case $O(\log \delta)$ random bits and expected time $O(\log n + \delta)$, assuming that corruptions are performed by a non-adaptive adversary. The running time matches the algorithm by Finocchi et al. [49], which, however, uses expected $O(\log n \cdot \log \delta)$ random bits. The main idea of our algorithm is to implicitly divide the sorted input array in 2δ disjoint sorted sequences $S_0, \ldots, S_{2\delta-1}$, each of size at most $\lceil n/2\delta \rceil$. The j'th element of S_i, $S_i[j]$, is the element at position $\text{pos}_i(j) = 2\delta j + i$ in the input array. Intuitively, this divides the input array into $\lceil n/2\delta \rceil$ consecutive *blocks* of size 2δ, where $S_i[j]$ is the i'th element of the j'th block. Note that, since 2δ disjoint sequences are defined from the input array and at most δ corruptions are possible, at least half of the sorted sequences $S_0, \ldots, S_{2\delta-1}$ do not contain any corrupted elements.

The algorithm generates a random number $k \in \{0, \ldots, 2\delta - 1\}$ and performs an iterative binary search on S_k. We store in safe memory k, the search key e, and the left and right indices, l and r, used by the binary search. The binary search terminates when l and r are adjacent in S_k, and therefore 2δ elements apart in the input array, since $\text{pos}_k(r) - \text{pos}_k(l) = 2\delta$ when $r = l + 1$. If the binary search was not mislead by corruptions, then the location of e is between $\text{pos}_k(l)$ and $\text{pos}_k(r)$ in the input array. To check whether the search was mislead, we perform the following verification procedure. Consider the neighborhoods N_l and N_r, containing the $2\delta + 1$ elements in the input array situated to the

left of $\text{pos}_k(l)$ and to the right of $\text{pos}_k(r)$ respectively. We compute the number $s_l = |\{z \in N_l \mid z \leq e\}|$ of elements in N_l that are smaller than e in $O(\delta)$ time by scanning N_l. Similarly, we compute the number s_r of elements in N_r that are larger than e. If $s_l \geq \delta + 1$ and $s_r \geq \delta + 1$, and the search key is not encountered in N_l or N_r, we decide whether it lies in the array or not by scanning the $2\delta - 1$ elements between $\text{pos}_k(l)$ and $\text{pos}_k(r)$. If s_l or s_r is smaller than $\delta + 1$, a corruption has misguided the search. In this case, a new k is randomly selected and the binary search is restarted.

Theorem 10.1 *The randomized dictionary supports searches in $O(\log n + \delta)$ expected time and uses $O(\log \delta)$ expected random bits.*

Proof. We first prove the correctness of the algorithm. Assume that $s_l \geq \delta + 1$ and $e \notin N_l$. Since only δ corruptions are possible, there exists an uncorrupted element in N_l strictly smaller than e. Because the input array is sorted, no uncorrupted elements to the left of $\text{pos}_k(l)$ in the input array are equal to e. By a similar argument, if $s_r \geq \delta + 1$ and $e \notin N_r$, then no uncorrupted elements to the right of $\text{pos}_k(r)$ in the input array are equal to e. If no corrupted elements are encountered during the binary search, all the uncorrupted elements of N_l are smaller than e, and therefore $s_l \geq \delta + 1$. Similarly, we have $s_r \geq \delta + 1$, and the algorithm terminates after scanning the elements between l and r.

We now analyze the running time. Each iteration generates a random number $k \in \{0, \ldots, 2\delta - 1\}$, using $O(\log \delta)$ random bits. The sorted sequences induced by different k's are disjoint, thus at most δ of them may contain corruptions. Since there are 2δ sorted sequences, the probability of selecting a value k that leads to a corruption-free sequence is at least $1/2$, and therefore the expected number of iterations is at most two. Each iteration uses $O(\log n)$ time for the binary search and $O(\delta)$ time for the verification. We conclude that a search uses expected $O(\log \delta)$ random bits and $O(\log n + \delta)$ expected time. □ □

We note that for each iteration an adaptive adversary can learn about the subsequence S_k on which we perform the binary search by investigating the elements accessed. Subsequently a single corruption suffices to force the search path to end far enough from its correct position such that the verification fails. In this situation, the algorithm performs $O(\delta)$ iterations and therefore $O(\delta(\log n + \delta))$ time regardless of the random choices of subsequences on which to perform the binary search.

We obtain a worst case bound of $O(\log \delta)$ random bits by using a standard derandomization technique. In the i'th iteration we perform the binary search on sequence $S_{h(i)}$, for $h(i) = (r_0 + ir_1 + i^2 r_2 + i^3 r_3) \bmod k$, where k is a prime number with $2\delta \leq k < 4\delta$, and r_i are chosen uniformly at random in $\{0, \ldots, k-1\}$. By construction $h(i)$ is a 4-wise independent hash function [69], which suffices to obtain an expected constant number of iterations for our algorithm [95].

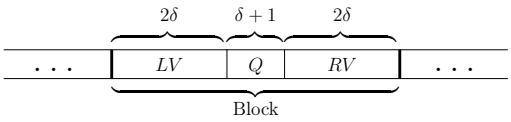

Figure 10.1: The structure of a block. The left and right verification segments, LV and RV, contain 2δ elements each, and the query segment Q contains $\delta+1$ elements.

10.2 Optimal static dictionary

In this section we close the gap between lower and upper bounds for deterministic resilient searching algorithms. We present a resilient algorithm that searches for an element in a sorted array in $O(\log n + \delta)$ time in the worst case, which is optimal [51]. It is an improvement of the previously published best deterministic dictionary, which supports searches in $O(\log n + \delta^{1+\varepsilon})$ time [49]. We reuse the idea presented in the design of the randomized algorithm and define disjoint sorted sequences to be used by a binary search algorithm. Similarly to the randomized algorithm, we design a verification procedure to check the result of the binary search. We design the adapted binary search and the verification procedure such that we are guaranteed to advance only one level in the binary search for each corrupted element misleading the search. We count the number of detected corruptions and adjust our algorithm accordingly to ensure that no element is used more than once, excepting a final scan performed only once on two adjacent blocks. The total time used for verification is $O(\delta)$.

We divide the input array into implicit blocks. Each block consists of $5\delta+1$ consecutive elements of the input and is structured in three segments: the *left verification segment*, LV, consists of the first 2δ elements, the next $\delta+1$ elements form the *query segment*, Q, and the *right verification segment*, RV, consists of the last 2δ elements of the block, see Figure 10.1. The left and right verification segments, LV and RV, are used only by the verification procedure. The elements in the query segment are used to define the sorted sequences $S_0, \ldots, S_\delta$, similarly to the randomized dictionary previously introduced. The j'th element of sequence S_i, $S_i[j]$, is the i'th element of the query segment of the j'th block, and is located at position $\text{pos}_i(j) = (5\delta+1)j + 2\delta + i$ in the input array.

We store a value $k \in \{0, \ldots, \delta\}$ in safe memory identifying the sequence S_k on which we currently perform the binary search. Also, k identifies the number of corruptions detected. Whenever we detect a corruption, we change the sequence on which we perform the search by incrementing k. Since there are $\delta+1$ disjoint sequences, there exists at least one sequence without any corruptions.

Binary search. The binary search is performed on the elements of S_k. Similarly to the randomized algorithm, we store in safe memory the search key, e, and the left and right sequence indices, l and r, used by the binary search. Initially, $l = -1$ is the position of an implicit $-\infty$ element. Similarly, r is the

position of an implicit ∞ to the right of the last element. Since each element in S_k belongs to a distinct block, l and r also identify two blocks, B_l and B_r.

Each step in the binary search compares the search key e against the element at position $i = \lfloor (l + r)/2 \rfloor$ in S_k. Assume without loss of generality that this element is smaller than e. We set l to i and decrement r by one. We then compare e with $S_k[r]$. If this element is larger than e, the search continues. Otherwise, if no corruptions have occurred, the position of the search element is in block B_r or B_{r+1} in the input array. When two adjacent elements are identified as in the case just described, or when l and r become adjacent, we invoke a verification procedure on the corresponding blocks. The pseudo-code description of the binary search is given in Algorithm 1, and a working example is shown in Figure 10.2.

Algorithm 1: Pseudo-code for the binary search procedure.

```
l ← −1
r ←last-block+1
while r − l > 1 do
    i ← ⌈(l+r)/2⌉
    if rep_k(block(i)) < e then
        l ← i
        r ← r − 1
        if rep_k(block(r)) < e then
            if verify(r,r+1) is successful then
                return success
            else
                Backtrack
    else if rep_k(block(i)) > e then
        Similar to previous case.
    else
        return success
if verify(l,r) is successful then
    return success
else
    Backtrack
```

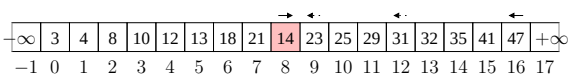

Figure 10.2: Example of binary search on a sequence S_k, for the search key 21. The arrows show the direction of the search. The emphasized element is corrupted.

The verification procedure determines whether the two adjacent blocks, denoted B_i and B_{i+1}, are correctly identified. If the verification succeeds, the binary search is completed, and all the elements in the two corresponding ad-

jacent blocks, B_i and B_{i+1} are scanned. The search returns true if e is found during the scan, and false otherwise. If the verification fails, the search may have been mislead by corruptions and we backtrack it two steps. To facilitate backtracking, we store two word-sized bit-vectors, d and f in safe memory. The i'th bit of d indicates the direction of the search and the i'th bit of f indicates whether there was a rounding in computing the middle element in the i'th step of the binary search respectively. We can easily compute the values of l and r in the previous step of the binary search by retrieving the relevant bits of d and f. If the verification fails, it detects at least one corruption and therefore k is incremented, thus the search continues on a different sequence S_k.

Verification phase. Verification is performed on two adjacent blocks, B_i and B_{i+1}. It either determines that e lies in B_i or B_{i+1} or detects corruptions. The verification is an iterative algorithm maintaining a value which expresses the confidence that the search key resides in B_i or B_{i+1}. We compute the *left confidence*, c_l, which is a value that quantifies the confidence that e is in B_i or to the right of it. Intuitively, an element in LV_i smaller than e is consistent with the thesis that e is in B_i or to the right of it. However an element in LV_i larger than e is inconsistent. Similarly, we compute the *right confidence*, c_r, to express the confidence that e is in B_{i+1} or to the left of it.

To compute c_l we scan a sub-interval of the left verification segment, LV_i, of B_i. Similarly, the right confidence is computed by scanning the right verification segment, RV_{i+1}, of B_{i+1}. Initially, we set $c_l = 1$ and $c_r = 1$. We scan LV_i from right to left starting at the element at index $v_l = 2\delta - 2k$ in LV_i. Intuitively, by the choice of v_l we ensure that no element in LV_i is accessed more than once. Similarly, we scan RV_{i+1} from left to right beginning with the element at position $v_r = 2k$. In an iteration we compare $LV_i[v_l]$ and $RV_{i+1}[v_r]$ against e. If $LV_i[v_l] \leq e$, c_l is increased by one, otherwise it is decreased by one and k is increased by one. Similarly, if $RV_{i+1}[v_r] \geq e$, c_r is increased; otherwise, we decrease c_r and increase k. The verification procedure stops when $\min(c_r, c_l)$ equals $\delta - k + 1$ or 0. The verification succeeds in the former case, and fails in the latter. The pseudo-code for the verification procedure is introduced in Algorithm 2, and a working example is shown in Figure 10.3.

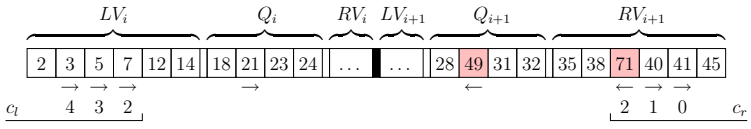

Figure 10.3: A verification step for $\delta = 3$, with $k = 1$ initially. The search key is 45. The verification algorithms stops with $c_r = 0$, reporting failure. The emphasized elements are corrupted.

Theorem 10.2 *The resilient algorithm searches for an element in a sorted array in $O(\log n + \delta)$ time.*

Algorithm 2: Pseudo-code for the verification procedure.

```
input : k: Number of errors identified so far
        δ: maximum number of errors
        l:index of the left block
        r:index of the right block
LV ← index of first element in LV_l
RV ← index of first element in RV_r
i_l ← LV + 2δ − 2k
i_r ← RV + 2k
c_r, c_l ← 1
while 0 < min(c_l, c_r) < δ − k + 1 do
    if A[i_l] < e then
        c_l ← c_l + 1
    else
        c_l ← c_l − 1
        k ← k + 1
    if A[i_r] > e then
        c_r ← c_r + 1
    else
        c_r ← c_r − 1
        k ← k + 1
    i_l ← i_l − 1
    i_r ← i_r + 1
if min(c_l, c_r) = 0 then
    return failure
else
    Scan left and right block and return result
```

Proof. We first prove that when c_l or c_r decrease during verification, a corruption has been detected. We increase c_l when an element smaller than e is encountered in LV_i, and decrease it otherwise. Intuitively, c_l can been seen as the size of a stack S. When we encounter an element smaller than e, we treat it as if it was pushed, and as if a pop occurred otherwise. Initially, the element g from the query segment of B_i used by the binary search is pushed in S. Since g was used to define the left boundary in the binary search, $g < e$ at that time. Each time an element $LV_i[v] < e$ is popped from the stack, it is *matched* with the current element $LV_i[v_l]$. Since $LV_i[v] < e < LV_i[v_l]$ and $v_l < v$, at least one of $LV_i[v_l]$ and $LV_i[v]$ is corrupted, and therefore each match corresponds to detecting at least one corruption. It follows that if $2t-1$ elements are scanned on either side during a failed verification, then at least t corruptions are detected.

We now argue that no single corrupted cell is counted twice. A corruption is detected if and only if two elements are matched during verification. Thus it suffices to argue that no element participates in more than one matching. We first analyze corruptions occurring in the left and right verification segments. Since the verification starts at index $2(\delta - k)$ in the left verification segment and k is increased when a corruption is detected, no element is accessed twice,

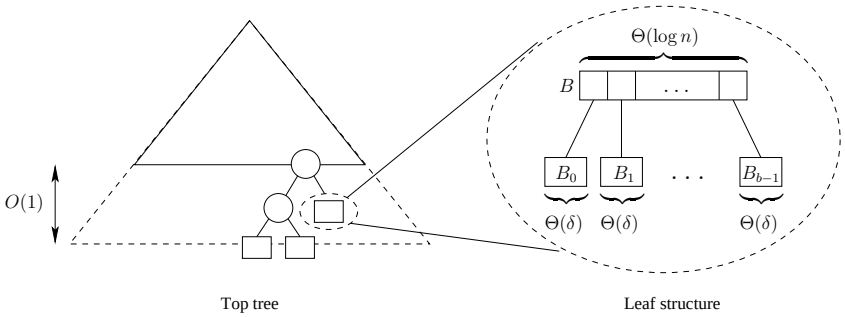

Figure 10.4: The structure of the dynamic dictionary.

and therefore not matched twice either. A similar argument holds for the right verification segment. Each failed verification increments k, thus no element from a query segment is read more than once. In each step of the binary search both the left and the right indices are updated. Whenever we backtrack the binary search, the last two updates of l and r are reverted. Therefore, if the same block is used in a subsequent verification, a new element from the query segment is read, and this new element is the one initially on the stack. We conclude that elements in the query segments, which are initially placed on the stack, are never matched twice either.

To argue correctness we prove that if a verification is successful, and e is not found in the scan of the two blocks, then no uncorrupted element equal to e exists in the input. If a verification succeeds and e is not found in either block, then $c_l \geq \delta - k + 1$. Since only $\delta - k$ more corruptions are possible, there is at least one uncorrupted element in LV_i smaller than e and thus there can be no uncorrupted elements equal to e to the left of B_i in the input array. By a similar argument, if $c_r \geq \delta - k + 1$, then all uncorrupted elements to the right of B_{i+1} in the input array are larger than e.

We now analyze the running time. We charge each backtracking of the binary search to the verification procedure that triggered it. Therefore, the total time of the algorithm is $O(\log n)$ plus the time required by verifications. To bound the time used for all verification steps we use the fact that if $O(f)$ time is used for a verification step, then $\Omega(f)$ corruptions are detected or the algorithm ends. At most $O(\delta)$ time is used in the last verification for scanning the two blocks. □ □

10.3 Dynamic dictionary

In this section we describe a linear space resilient deterministic dynamic dictionary supporting searches in optimal $O(\log n + \delta)$ worst case time and range queries in optimal $O(\log n + \delta + k)$ worst case time, where k is the size of the output. The amortized update cost is $O(\log n + \delta)$.

Structure. The sorted sequence of elements is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log n)$ elements. For each leaf structure we select a guiding element, and we place these $O(n/(\delta \log n))$ guiding elements in the leaves of a reliably stored binary search tree. Each guiding element is chosen such that it is larger than all uncorrupted elements in the corresponding leaf structure.

For this reliable *top tree* T, we use the (non-resilient) binary search tree in [25], which consists of $h = \log |T| + O(1)$ levels when containing $|T|$ elements. In the full version [26] it is shown that the tree can be maintained such that the first $h - 2$ levels are complete. We lay the tree in memory in left-to-right breadth first order, as specified in [25]. It uses linear space, and an update costs amortized $O(\log^2 |T|)$ time. A global rebuilding is performed when $|T|$ changes by a constant factor.

All the elements and pointers in the top tree are stored reliably, using replication. Since a reliable value takes $O(\delta)$ space, $O(\delta|T|)$ space is used for the entire structure. The time used for storing and retrieving a reliable value is $O(\delta)$, and therefore the additional work required to handle the reliably stored values increases the amortized update cost to $O(\delta \log^2 |T|)$ time.

The leaf structure consists of a top bucket B and b buckets, $B_0, \ldots, B_{b-1}$, where $\log n \leq b \leq 4 \log n$. Each bucket B_i contains between δ and 6δ input elements, stored consecutively in an array of size 6δ, and uncorrupted elements in B_i are smaller than uncorrupted elements in B_{i+1}. For each bucket B_i, the top bucket B associates a guiding element larger than all elements in B_i, a pointer to B_i, and the size of B_i, all stored reliably. Since storing a value reliably uses $O(\delta)$ space, the total space used by the top bucket is $O(\delta \log n)$. The guiding elements of B are stored as a sorted array to enable fast searches using the deterministic resilient search algorithm from Section 10.2.

Lemma 10.1 *The dynamic dictionary uses $O(n)$ space to store n elements.*

Proof. Since a leaf structure stores $\Theta(\delta \log n)$ input elements, the top tree contains $O(n/(\delta \log n))$ nodes, using $O(\delta|T|) = O(\delta n/(\delta \log n)) = o(n)$ space. Each of the $O(n/(\delta \log n))$ leaf structures uses $O(\delta \log n)$ space and therefore the total space used for leaf structures is $O(n)$. □ □

Searching. The search operation consists of two steps. It first locates a leaf in the top tree T, and then searches the corresponding leaf structure. Let h denote the height of T. If $h \leq 3$, we perform a standard tree search from the root of T using the reliably stored guiding elements and pointers. Otherwise, we locate two internal nodes, v_1 and v_2, with guiding elements g_1 and g_2, such that $g_1 < e \leq g_2$, where e is the search key. Since $h - 2$ is the last complete level of T, level $\ell = h - 3$ is complete and contains only internal nodes. The breadth first layout of T ensures that elements of level ℓ are stored consecutively in memory. The search operation locates v_1 and v_2 using the deterministic resilient search algorithm from Section 10.2 on the array defined by level ℓ. The search only considers the $2\delta + 1$ cells in each node containing

guiding elements and ignores memory used for auxiliary information, e.g. sizes and pointers. Although they are stored using replication, the guiding elements are considered as $2\delta + 1$ regular elements in the search. Since the space used by the auxiliary information is the same for all nodes, these gaps in the memory layout of level ℓ are easily excluded from the search. We modify the resilient searching algorithm previously introduced such that it reports two consecutive blocks with the property that if the search key is in the structure, it is contained in one of them. The reported two blocks, each of size $5\delta + 1$, span $O(1)$ nodes of level ℓ and the guiding elements of these are queried reliably to locate v_1 and v_2. The appropriate leaf can be in either of the subtrees rooted at v_1 and v_2, and we perform a standard tree search in both using the reliably stored guiding elements and pointers. Searching for an element in a leaf structure is performed by using the resilient search algorithm from Section 10.2 on the top bucket, B, similar to the way v_1 and v_2 were found in T. The corresponding reliably stored pointer is then followed to a bucket B_i, which is scanned.

Range queries can be performed by scanning the level ℓ, starting at v, and reporting relevant elements in the leaves below it.

Lemma 10.2 *The search operation of the dynamic dictionary uses $O(\log n+\delta)$ worst case time. A range query reporting k elements is performed in worst case $O(\log n + \delta + k)$ time.*

Proof. The initial search in the top tree takes $O(\log n + \delta)$ worst case time by Theorem 10.2. Traversing the $O(1)$ levels to a leaf takes time $O(\delta)$. Searching in the top bucket of the leaf structures uses $O(\log\log n + \delta)$ time, again using Theorem 10.2. The final scan of a bucket takes time $O(\delta)$.

In a range query, the elements reported in any leaf completely contained in the query range pay for the $O(\delta \log n)$ time used for going through the bottom part of the top tree and scanning the top bucket. The search pays for the rightmost traversed leaf. □ □

Updates. Updating the structure is performed using standard bucketing techniques. To insert an element into the dictionary, we first perform a search to locate the appropriate bucket B_i in a leaf structure, and then the element is appended to B_i and the size of B_i in the top bucket is updated. When the size of B_i increases to 6δ, we split it into two buckets, B_s and B_g, of almost equal sizes. We compute a guiding element that splits B_i in $O(\delta^2)$ time by repeatedly scanning B_i and extracting the minimum element. The element m returned by the last iteration is kept in safe memory. In each iteration, we select a new m which is the minimum element in B_i larger than the current m. Since at most δ corruptions can occur, B_i contains at least 2δ uncorrupted elements smaller than m and 2δ uncorrupted elements larger, after $|B_i|/2 = 3\delta$ iterations. The elements from B_i smaller than m are stored in B_s, and the remaining ones are stored in B_g. The guiding element for B_s is m, while B_g preserves the guiding element of B_i. The new split element is reliably inserted in the top bucket using an insertion sort step, by scanning and shifting the elements in B from right to left, and placing the new element at its appropriate position. Similarly, when

the size of the top bucket becomes $4 \log n$, it is split in two new leaf structures. The first leaf structure consists of the first $2 \log n$ bottom buckets, and the second leaf structure contains the rest. The second leaf structure is associated with the original guiding element, and the guiding element of the new leaf structure is the the last guiding element in its top bucket. This new guiding element is inserted into the top tree.

Deletions are handled similarly by first searching for the element and then removing it from the appropriate bucket. When an element is deleted from a bucket, we ensure that the elements in the affected bucket are stored consecutively by swapping the deleted element with the last element. If the affected bucket holds fewer than δ elements after the deletion, it is merged with a neighboring bucket. If the resulting bucket contains more than 6δ elements, it is split as described above. If the top bucket contains less than $\log n$ guiding elements, it is merged with a neighboring leaf structure which is found using a search. Following this, the original leaf is deleted from the top tree.

Lemma 10.3 *The insert and delete operations of the dynamic dictionary take* $O(\log n + \delta)$ *amortized time each.*

Proof. An update in the top tree takes $O(\delta \log^2 n)$ time and requires $\Omega(\delta \log n)$ updates in the leaf structures. Thus each update costs amortized $O(\log n)$ time for operations in the top tree. Splitting and merging a bucket of a leaf structure takes time $O(\delta \log n)$ for updates to the top bucket and $O(\delta^2)$ time for computing a split element for a bucket. A bucket is split or merged every $\Omega(\delta)$ operations resulting in an amortized update cost of $O(\log n + \delta)$. Appending or removing a single element to a bucket takes worst case time $O(\delta)$ for updating the size. Adding the $O(\log n + \delta)$ cost of the initial search concludes the proof. □ □

Theorem 10.3 *The resilient dynamic dictionary structure uses* $O(n)$ *space while supporting searches in* $O(\log n + \delta)$ *time worst case with an amortized update cost of* $O(\log n + \delta)$. *Range queries with an output size of* k *is performed in worst case* $O(\log n + \delta + k)$ *time.*

Bibliography

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proc. 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 540–549, 2004.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[5] R. Anderson and M. Kuhn. Tamper resistance - a cautionary note. In *Proc. 2nd Usenix Workshop on Electronic Commerce*, pages 1–11, 1996.

[6] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, pages 125–136, 1997.

[7] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33(5):269–273, 1990.

[8] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. 2002.

[9] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[10] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, page 27. CRC Press, 2004.

[11] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the i/o-complexity of comparison-based algorithms. In *Proc. 3rd Workshop on Algorithms and Data Structures*, pages 83–94. Springer-Verlag, 1993.

[12] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 580–589, 1996.

[13] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[14] Z. Bar-Yosseff, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

[15] G. E. Blelloch and B. M. Maggs. Parallel algorithms. In *The Computer Science and Engineering Handbook*, pages 277–315. 1997.

[16] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[17] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the Intel pentium 4 processor on 90nm thechnology. *Intel Technology Journal*, 08(01):1–18, 2004.

[18] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT*, pages 37–51, 1997.

[19] R. S. Borgstrom and S. R. Kosaraju. Comparison-based search in the presence of errors. In *Proc. 25th Annual ACM symposium on Theory of Computing*, pages 130–136, 1993.

[20] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[21] G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 3–13. Springer Verlag, Berlin, 2004.

[22] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 426–438. Springer Verlag, Berlin, 2002.

[23] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. 35th Annual ACM Symposium on Theory of Computing*, pages 307–315, 2003.

[24] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. In *Proc. 14th Annual European Symposium on Algorithms*, 2007. To appear.

[25] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[26] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT, May 2002.

[27] G. S. Brodal, R. Fagerberg, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. Technical Report RS-07-12, BRICS, 2007.

[28] G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 576–588. Springer Verlag, Berlin, 2005.

[29] G. S. Brodal, R. Fagerberg, and G. Moruz. On the adaptiveness of quicksort. In *Proc. 7th Workshop on Algorithm Engineering and Experiments*, pages 130–140, 2005.

[30] G. S. Brodal and G. Moruz. Tradeoffs between branch mispredictions and comparisons for sorting algorithms. In *Proc. 9th International Workshop on Algorithms and Data Structures*, volume 3608 of *Lecture Notes in Computer Science*, pages 385–395. Springer Verlag, Berlin, 2005.

[31] G. S. Brodal and G. Moruz. Skewed binary search trees. In *Proc. 14th Annual European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 708–719. Springer Verlag, Berlin, 2006.

[32] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[33] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer Systems Science*, 7(4):354–375, 1973.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition.* MIT Press, 2001.

[35] E. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 2002.

[36] C. Demetrescu, B. Escoffier, G. Moruz, and A. Ribichini. Adapting parallel algorithms to the w-stream model, with applications to graph problems. In *Proc. 32nd International Symposium on Mathematical Foundations of Computer Science*, 2007. To appear.

[37] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proc. 17th Annual ACM-SIAM Symposium of Discrete Algorithms*, pages 714–723, 2006.

[38] K. Diks and A. Pelc. Optimal adaptive broadcasting with a bounded fraction of faulty nodes (extended abstract). In *Proc. 5th Annual European Symposium on Algorithms*, pages 118–129, 1997.

[39] A. Elmasry. Priority queues, pairing, and adaptive sorting. In *29th Annual International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 183–194. Springer Verlag, Berlin, 2002.

[40] A. Elmasry. Adaptive sorting with avl trees. In *3rd IFIP International Conference on Theoretical Computer Science*, pages 307–316, 2004.

[41] A. Elmasry and M. L. Fredman. Adaptive sorting and the information theoretic lower bound. In *20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *Lecture Notes in Computer Science*, pages 654–662. Springer Verlag, Berlin, 2003.

[42] A. Elmasry and A. Hammad. An empirical study for inversions-sensitive sorting algorithms. In *4th International Workshop on Experimental and Efficient Algorithms*, pages 597–601, 2005.

[43] V. Estivill-Castro and D. Wood. A new measure of presortedness. *Information and Computation*, 83(1):111–119, 1989.

[44] V. Estivill-Castro and D. Wood. Practical adaptive sorting. In *Advances in Computing and Information - Proc. International Conference on Computing and Information*, pages 47–54. Springer-Verlag, 1991.

[45] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surverys*, 24(4):441–475, 1992.

[46] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the streaming model: the value of space. In *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, 2005.

[47] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

[48] J. Feigenbaum, S. Kannan, M. J. Strauss, and M. Viswanathan. An approximate L^1 difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2003.

[49] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. In *Proc. 33rd International Colloquium on Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2006.

[50] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 547–554, 2007.

[51] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proc. 36th Annual ACM Symposium on Theory of Computing*, pages 101–110, 2004.

[52] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache oblivious algorithms. In *40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[53] L. Gasieniec and A. Pelc. Broadcasting with a bounded fraction of faulty nodes. *Journal of Parallel and Distributed Computing*, 42(1):11–20, 1997.

[54] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 389–398, 2002.

[55] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Quicksand: Quick summary and analysis of network data. Technical report, DIMACS Technical Report 2001-43, 2001.

[56] L. Golab and M. T. Özsu. Data stream management issues – a survey. Technical report, School of Computer Science, University of Waterloo, TR CS-2003-08, 2003.

[57] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[58] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation of linear lists. In *Proc. 9th Annual ACM Symposium on Theory of Computing*, pages 49–60, 1977.

[59] J. Hastad and T. Leighton. Fast computation using faulty hypercubes. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 251–263, 1989.

[60] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 274–284, 1987.

[61] J. L. Hennesy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 2006.

[62] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. pages 107–118, 1999.

[63] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[64] C. A. R. Hoare. Algorithm 63: Partition. *Commun. ACM*, 4(7):321, 1961.

[65] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, 1961.

[66] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, April 1962.

[67] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33:518–528, 1984.

[68] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[69] A. Joffe. On a set of almost deterministic k-independent random variables. *Annals of Probability*, 2(1):161–162, 1974.

[70] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *Proc. 10th International Workshop on Algorithms and Data Structures*, 2007. To appear.

[71] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. D. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors (extended abstract). In *Proc. 31st Annual Symposium on Foundations of Computer Science*, pages 285–296, 1990.

[72] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *Proc. 14th Annual European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 780–791. Springer, 2006.

[73] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

[74] D. E. Knuth. *The Art of Computer Programming. Vol 3, Sorting and Searching*. Addison-Wesley, 1973.

[75] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[76] S. Kutten and D. Peleg. Fault-local distributed mending. *Journal of Algorithms*, 30(1):144–165, 1999.

[77] S. Kutten and D. Peleg. Tight fault locality. *SIAM Journal on Computing*, 30(1):247–268, 2000.

[78] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Transactions on Computers*, 40(9):1081–1084, 1991.

[79] F. T. Leighton and B. M. Maggs. Expanders might be practical: Fast algorithms for routing around faults on multibutterflies. In *Proc. 30th Annual Symposium on Foundations of Computer Science*, pages 384–389, 1989.

[80] C. Levcopoulos and O. Petersson. Splitsort – an adaptive sorting algorithm. *Information Processing Letters*, 39(1):205–211, 1991.

[81] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal of Computing*, 15(4):1036–1053, 1986.

[82] H. Manilla. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 34:318–325, 1985.

[83] S. McFarling. Combining branch predictors. Technical report, Western Research Laboratory, 1993.

[84] A. McGregor. Finding matchings in the streaming model. In *Proc. 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 170–181, 2005.

[85] K. Mehlhorn. *Data Structures and Algorithms. Vol. 1, Sorting and Searching*. Springer Verlag, 1984.

[86] A. Moffat, O. Petersson, and N. C. Wormald. Sorting and/by merging finger trees. In *Algorithms and Computation: Third International Symposium, ISAAC '92*, volume 650 of *Lecture Notes in Computer Science*, pages 499–508. Springer Verlag, Berlin, 1992.

[87] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[88] S. Muthukrishnan. Data streams: algorithms and applications, 2003. Available at `http://www.cs.rutgers.edu/ muthu`.

[89] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. 4th Annual ACM Symposium on Theory of Computing*, pages 137–142, 1972.

[90] A. Pagh, R. Pagh, and M. Thorup. On adaptive integer sorting. In *Proc. 12th Annual European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 556–567. Springer, 2004.

[91] PAPI (Performance Application Programming Interface). Software library found at `http://icl.cs.utk.edu/papi/`, 2004.

[92] S. Park and B. Bose. All-to-all broadcasting in faulty hypercubes. *IEEE Transactions on Computers*, 46(7):749–755, 1997.

[93] O. Petersson and A. Moffat. A framework for adaptive sorting. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 59:152–179, 1995.

[94] U. F. Petrillo, I. Finocchi, and G. F. Italiano. The price of resiliency: a case study on sorting with memory faults. In *Proc. 14th Annual European Symposium on Algorithms*, pages 768–779, 2006.

[95] S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 713–722, 2002.

[96] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., 1996.

[97] R. M. Ramanathan. Extending the world's most popular processor architecture. Available at: `http://www.intel.com/technology/magazine/computing/new-instructions-1006.htm`, 2006.

[98] R. M. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. Available at: `http://www.intel.com/technology/magazine/computing/quad-core-1206.htm`, 2006.

[99] B. Ravikumar. A fault-tolerant merge sorting algorithm. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, pages 440–447, 2002.

[100] J. H. Reif. Optimal parallel algorithms for integer sorting and graph connectivity. Technical Report TR 08-85, Aiken Computation Laboratory, Harvard University, Cambridge, 1985.

[101] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *Proc. 26th Annual International Symposium on Fault-Tolerant Computing*, pages 394–403, 1996.

[102] G. K. Saha. Software based fault tolerance: a survey. *Ubiquity*, 7(25), 2006.

[103] P. Sanders and S. Winkel. Super scalar sample sort. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes in Computer Science*, pages 784–796. Springer Verlag, Berlin, 2004.

[104] R. Sedgewick. Analysis of shellsort and related algorithms. In *Proc. 4th European Symposium on Algorithms*, pages 1–11, 1996.

[105] R. Seidel. Backwards analysis of randomized geometric algorithms. Technical Report TR-92-014, International Computer Science Institute, Univeristy of Calfornia at Berkeley, February 1992.

[106] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[107] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. International Conference on Dependable Systems and Networks*, pages 389–398. IEEE Computer Society, 2002.

[108] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12, 2002.

[109] G. R. Srinivasan. Modeling the cosmic-ray-induced soft-error rate in integrated circuits: an overview. *IBM Journal of Research and Development*, 40(1):77–89, 1996.

[110] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conference*, pages 13–24, 1998.

[111] A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 5th edition edition, 2006.

[112] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 12–20, 1984.

[113] Tezzaron Semiconductor. Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html, 2004.

[114] A. J. van de Goor. *Testing Semiconductor Memories: Theory and Practice*. ComTex Publishing, Gouda, The Netherlands, 1998.

[115] R. van der Pas. Memory hierarchy in cache-based systems. Sun Microsystems Blueprints, November 2002.

[116] P. van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. 1990.

[117] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[118] O. Wechsler. Inside Intel core microarchitecture: Setting new standards for energy-efficient performance. Available at: `http://www.intel.com/technology/magazine/computing/core-architecture-0306.htm`, 2006.

[119] www.wikipedia.org. Moore's law. Available at: `http://en.wikipedia.org/wiki/Moore's_law`, 2007.

[120] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proc. International Conference on Dependable Systems and Networks*, pages 421–430, 2001.

[121] S. S. Yau and F.-C. Chen. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, 1980.

[122] Y.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM International Symposium on Computer Architecture*, pages 124–134, 1992.

[123] W. Zhang. Replication cache: A small fully associative cache to improve data cache reliability. *IEEE Transactions on Computers*, 54(12):1547–1555, 2005.

[124] W. Zhang, S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam. Icr: In-cache replication for enhancing data cache reliability. In *International Conference on Dependable Systems and Networks*, pages 291–300, 2003.