Superscalar Sample Queue: Engineering a Distribution-Based Priority Queue

Master's Thesis of

Raphael von der Grün

at the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer: Prof. Dr. rer. nat. Peter Sanders
Advisor: Prof. Dr. rer. nat. Peter Sanders

1. November 2020 – 30. April 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Nuremberg, 30. April 2021

(Raphael von der Grün)

Abstract

To exploit the full performance potential of instruction-level parallelism in modern processors, the properties of their architecture must be considered. Although optimizing compilers do their best to produce machine code that uses the target's resources efficiently, they are constrained by the programs presented to them. While algorithmic design for efficient memory access is widespread, recent experimental results have also highlighted the importance of optimizations that reduce control hazards – thus further improving pipeline efficiency. We present a comparison-based priority queue that is cache-efficient, avoids branch mispredictions and supports operations *insert* and *deleteMin* in amortized expected time $O(\log N)$ and an amortized expected number of $O(1/B \log_{M/B} N/M)$ memory accesses on queues of size N, where M is the cache size and B the cache line size. In various experiments run on two x86-64 machines our implementation outperforms our strongest competitor – Sanders' Sequence Heap – by a factor of up to 2. We also show that transforming a single conditional branch into a data dependency improves the latter's performance by an average factor of 1.2.

Zusammenfassung

Um das Leistungspotential des Instruktionsebenenparallelismus moderner Prozessoren voll auszuschöpfen, müssen deren Architekturgegebenheiten berücksichtigt werden. Obwohl optimierende Compiler ihr Bestes geben um Maschinen-Code zu generieren, welcher die Ressourcen der Zielplattform effizient nutzt, sind sie in ihren Möglichkeiten doch beschränkt durch die Eingabeprogramme die sie übersetzen. Während Speichereffizienz beim Algorithmenentwurf meist berücksichtigt wird, haben neuere experimentelle Ergebnisse auch die Wichtigkeit von Optimierungen hervorgehoben, welche Steuerkonflikte reduzieren und dadurch die Pipeline-Effizienz noch weiter steigern. Wir präsentieren eine cache-effiziente vergleichsbasierte Vorrangwarteschlange die, wo möglich, falsch vorhergesagte Sprünge vermeidet und die Operationen *insert* und *deleteMin* bei einer Warteschlangengröße von N in amortisiert erwarteter Zeit $O(\log N)$, sowie amortisiert erwarteter Anzahl von $O(1/B \log_{M/B} N/M)$ Speicherzugriffen unterstützt – wobei M die Cache-Kapazität und B die Cache-Zeilengröße darstellen. In diversen Experimenten auf zwei x86-64 Computern konnte unsere Implementierung unseren stärksten Mitbewerber – Sanders' Sequence Heap – um einen Faktor von mehr als zwei schlagen.

Contents

List of Figures

1 Introduction

There are numerous applications for priority queues: the shortest path problem, discrete-event simulation, coding and compression, job scheduling, maximum flow computation and branch-and-bound [16, Section 6.6]. Since priority queues are often used as building blocks of more complex algorithms, their practical performance is of particular interest.

However, the performance characteristics of modern processors with their multi-tier memory hierarchies and instruction-level parallelism enabled by pipelining [8, 7] cannot be captured by counting instructions in the simplistic von Neumann model [12].

There is a considerable amount of theoretical work on priority queues that tries to give more realistic performance guarantees by providing theoretical bounds for the number of expensive memory accesses in the external-memory and cache-oblivious models [4, Section 10]. However, we could find no experimental results that suggest there exists a strong competitor to Sanders' Sequence Heap [14] when it comes to cached-memory performance. In fact, at least for the application in Dijkstra's Algorithm the contrary seems to be true [5].

To rectify this, we present the *Superscalar Sample Queue* (S^3Q), a comparison-based priority queue that is cache-efficient and avoids branch mispredictions wherever feasible. Our theoretical analysis shows that it supports operations *insert* and *deleteMin* in amortized expected time $O(\log N)$ and an amortized expected number of $O(1/B \log_{M/B} N/M)$ memory accesses on queues of size N, where M is the cache size and B the cache line size.

To verify that our results translate to actual performance on real hardware, we subjected our implementation to various benchmarks on different x86-64 machines. We compare our results with those of well-tuned versions of implicit binary heaps [19] and Sanders' Sequence Heap. Our results show that S^3Q is able to outperform all its competitors by a comfortable margin. As a side note, we also present a simple optimization that improves the performance of Sanders' Sequence Heap by an average factor of 1.2, thus narrowing the gap to our solution.

One way to look at S^3Q is as being the k-way distribution counterpart to Sanders' k-way merge based Sequence Heap. The advantage of the merge-based approach is that it is relatively indifferent to the distribution of the element keys. With our distribution-based approach we have to constantly make sure that the element partitions we maintain stay balanced, e.g. by subdividing key intervals that receive too many elements. However, our approach is implementable with few branch-mispredictions and furthermore offers several opportunities for parallelization.

S^3Q bears a certain similarity to Arge's cache-oblivious priority queue [1] from which we indeed drew some inspiration. Both share a multi-level distribution-based approach where elements are inserted into and removed from the finest-grained partitioning levels. They differ in the fact that our approach is cache-aware instead of cache-oblivious, the

size of our levels grows slower, and most importantly, we use sample-based splitting and k-way classification instead of sorting to distribute the elements.

To perform aforementioned operations efficiently, we borrowed concepts and implementation from *In-place Parallel Super Scalar Samplesort* (IPS^{4}o) [2], hence the name of our algorithm. Arguably the most important part we adopted is its extremely efficient k-way classification scheme and the implementation thereof.

Chapter 2 formally defines priority queues, presents the model we use to analyze our cache efficiency and gives a short overview of CPU pipelining and branch prediction. Chapter 3 proceeds with a detailed description of the algorithm we devised as well as a proof of its correctness. In chapter 4 we provide an analysis of the internal work that the algorithm performs and bound the number of expensive memory accesses. After that, chapter 5 gives an overview of our implementation and discusses interesting details as well as notable differences to the theoretical algorithm. Chapter 6 describes the methodology of our experiments, presents the choice of competitors and concludes by discussing our experimental results. Finally, in chapter 7 we explore possible ways to further improve the performance of S^{3}Q, close the remaining gap between theory and implementation and outline how to support additional operations.

2 Preliminaries

This chapter starts by formally defining priority queues, then presents the model we use to analyze our cache efficiency and concludes with a short overview of CPU pipelining and branch prediction.

2.1 Priority Queue

We define a *priority queue* (PQ) as a set of *elements* or *items*, each of which is identified by a unique *key*. The keys represent priorities and there exists a strict total order on them, which we also extend to their associated items. A priority queue provides at least the following operations on its elements:

insert insert an element into the queue
min get the element with the smallest key
deleteMin delete the element with the smallest key

We also use the concept of a *batched priority queue*. It provides the same semantics as a regular priority queue, but its operations always work in sets of elements. Given a batch size of S, *insert* inserts S elements at once, whereas *min* returns and *deleteMin* deletes the elements associated with the S smallest keys.

2.2 Memory Model

To analyze the cache-efficiency of our algorithms, we use the single-disk single-processor variant of the *parallel disk model* (PDM) [17, Chapter 2]. In this model, we count how often we need to transfer a block of size B between the fast memory of size M and the slow but unbounded external memory. In our case, the latter shall be the computer's main memory, while the former shall be the L1 data cache. Thus, M denotes the L1 data cache's capacity and B its cache line size. The model further prescribes that $1 \leq B \leq M/2$ and $M < N$, where N is the problem size. All parameters are given in units of items. We will refer to the transfer of cache lines as "memory accesses" or "I/Os".

Note that both placement as well as replacement policies of hardware caches are usually fixed, neither of which is modeled by the PDM [17, Section 2.3]. We will largely ignore this fact, with an exception being the consideration of cache associativity by reducing the effective size of M where applicable [13].

Moreover, the relative difference in speed between caches and main memory is orders of magnitude smaller than the relative difference in latencies between main memory and hard disks [17, Section 2.3]. Thus, we cannot simply disregard the work performed by the

CPU but instead analyze the number of processing instructions required by our algorithms as well. We will also refer to this metric as "time" or "internal work".

Finally, we will give bounds – both in terms of memory accesses and processor instructions – for two fundamental problems. The complexity of sequentially reading or writing (also *scanning* or *streaming*) N data items is bounded by [17, Chapter 3]

$$Scan(N) = \begin{cases} \Theta(N) & \text{time} \\ \Theta(N/B) & \text{I/Os,} \end{cases} \tag{2.1}$$

whereas the complexity of the comparison-based *sorting* of N elements is bounded by [17, Chapter 3, 16, Theorem 5.4]

$$Sort(N) = \begin{cases} \Theta(N \log N) & \text{time} \\ \Theta(N/B \log_{M/B} N/B) & \text{I/Os.} \end{cases} \tag{2.2}$$

Finally, note that the base of all logarithms in this work is 2 unless explicitly specified.

2.3 Pipelining

The following section is based on [7, Appendix C]. In it, Hennesey and Patterson give a concise synopsis of pipelining:

> *Pipelining* is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs.

While theoretically, the speedup provided by a pipeline is linear in the number of its stages, interdependencies between instructions can lessen that effect. Situations where these interdependencies prevent the execution of an instruction during its scheduled clock cycle are called *hazards*. For example, when an instruction wants to use the result of a preceding instruction but that result is not yet available, this would be called a *data hazard* (more specifically a *read-after-write hazard*). The reading instruction, and all others that come after it, might then have to wait until the result is ready. This is called a pipeline *stall*. The class of hazards which will be of prime interest to us are *control hazards*. They arise for example when the result on which a conditional branch depends is not yet ready and it is consequently unclear which instructions should be processed next.

There are many techniques that try to reduce the impact of the uncertainty introduced into programs by control logic. One such concept, of which at least a basic understanding is necessary to adequately gauge the impact of control flow optimizations, is the dynamic *branch prediction* that is performed by the CPU. While there are various schemes and associated parameters, the general idea is that the processor records some information about each conditional branch instruction in the program – e.g. if it was taken or not. Then, the CPU can refer to this information the next time that same conditional branch is encountered and can, for example, speculate that the outcome will be identical to before. If

the prediction is correct, the pipeline's efficiency is not affected by the conditional branch. If the prediction is incorrect, the speculatively executed instructions have to be removed from the pipeline and any of their results reverted.

This approach works well for conditional branches that mostly have the same outcome – which indeed applies to many common cases like loop control statements and error handling. However, the outcome of some conditional branches is not that easy to predict in general – like the decision whether to continue left or right during a binary search. This is where the need for clever algorithm engineering and careful evaluation arises.

3 Algorithm

We first give an informal overview of our data structure and then proceed to formally define its structure and operations theron.

3.1 Overview

The core of S^3Q is a batched priority queue that we call its *backend*. On top of that sits the *frontend*, a buffering adapter that provides the single-element priority queue interface.

The backend stores the contained items in multiple levels. The first level, through which all elements enter and leave the queue, holds the smallest items in the whole backend. Since the level size is limited, the biggest elements will have to move to the next level at some point. When that level runs full, the biggest items will move to a third level and so on. Each level partitions the contained items into subsets that we call buckets. These buckets are again ordered, such that all elements in a given bucket have smaller keys than any elements in any bucket following it. The elements inside the buckets on the other hand, are kept in no particular order. See Figure 3.1 for an illustration. If a bucket exceeds its size limit, its elements are partitioned into a number of smaller buckets. To limit the number of levels, the bucket capacity and thus the level capacity increases exponentially.

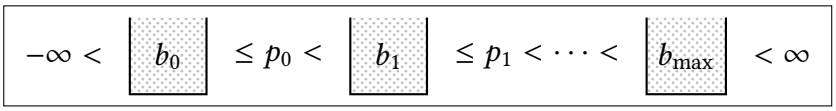

Figure 3.1: Structure of the ordered partitions managed by the queue levels.

The frontend contains all elements that are even smaller than those in the first level of the backend. To be able to provide efficient priority queue operations, it stores those elements as an implicit binary heap. When it runs out of elements it gets a new batch from the backend. When it grows too large to fit into fast memory it moves a fraction of its items to the backend. See Figure 3.2 for an overview of S^3Q's structure and data flow.

Before we can describe backend and frontend in more detail we need to define some parameters. First, let $c \in \Theta(M)$ be the maximum size of the binary heap in the frontend and the buckets in the first backend level. Moreover, let $k = 2^{2i} \in \Theta(M/B^{1+\varepsilon})$ where $i \in \mathbb{N} \setminus \{1\}$ and $0 \leq \varepsilon \leq 1$ be the maximum number of buckets on any level. Finally, let $\alpha = 2^i = \sqrt{k}$ be the split factor, i.e. the number of new buckets into which an overflowing bucket is split.

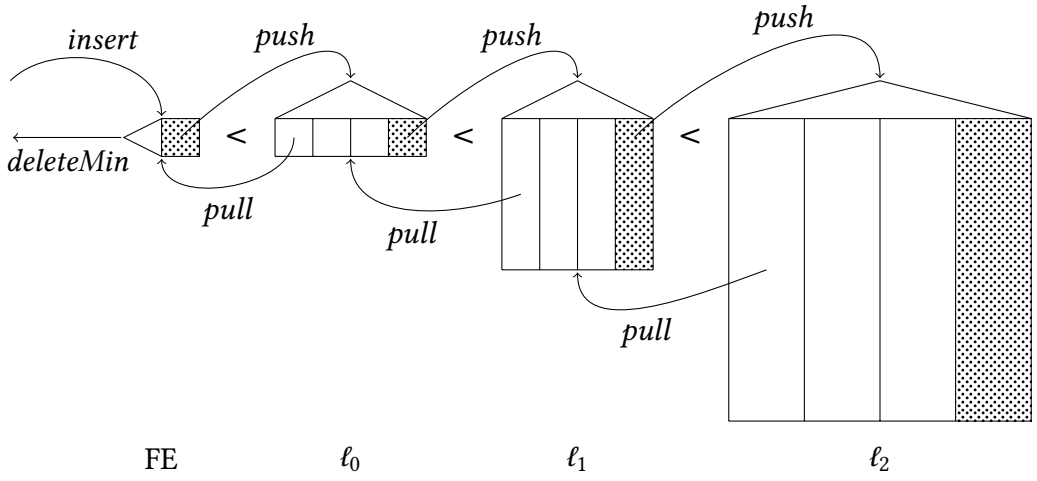

Figure 3.2: A S^3Q instance with frontend, three backend levels and a bucket growth factor of $k/2 = 4$ (not valid in practice). The arrows show how data flows through the levels. Except for the elements in the dotted max-buffers, all elements follow the order relation depicted by the < signs.

3.2 Backend

3.2.1 Structure

The backend of a PQ holding N elements consists of $\Theta(\log_k N/c)$ levels which are ordered from smallest (with regards to capacity) to largest and shall be denoted by $\ell_0, \ldots, \ell_{h-1}$. With each level, the capacity increases by a factor of $\Theta(k)$. Items enter and leave the PQ via the first level ℓ_0. In a multi-level PQ, ℓ_0 contains the $\Theta(ck)$ minimally keyed items as well as the $\Theta(c)$ most recently inserted items with larger keys. Items whose keys have rank beyond $\Theta(ck)$ will eventually be pushed to the next larger level. We call all levels except the last one *regular*.

Each level manages an ordered partition of the items it contains. That is, it consists of $\Theta(k)$ (or less, in case of the last level) *buckets* or *buffers*, referred to as $b_0, b_1, \ldots$, that partition the elements in that level such that the ordering of the buckets is consistent with the ordering of the elements. More formally, for any regular bucket b_i, $b_i \times b_{i+1}$ is a subset of the order relation over the elements in the PQ. The number of elements contained in any two buckets of a level can differ by a factor of up to $O(\alpha)$. This allows us to split an overflowing bucket into α smaller ones.

The last bucket of a level has a special role. It collects the elements that are too large for its level – until there are enough to efficiently push them to the next level. We also call that bucket *max-buffer* or $b_{\max}$ and the other buckets *regular*. The elements in the max-buffers are still in transit to their final location. Eventually, they could end up in any level past their current one.

For any level ℓ, we will refer to the number of buckets in that level as its *degree* or $deg(\ell)$. For any level or bucket x, we define its *size* as the number of elements it contains and its *capacity* as the maximum number of elements it is able to contain. For any regular

bucket b_j, let $p(b_j) = p_j$ refer to its *pivot* – the largest key of all elements contained in it. Furthermore, let $p(\{\}) = p(b_{\max}) = \infty$.

An empty backend is initialized with a single level that contains a single empty bucket. All operations we define below will maintain the following invariants:

1. Level degrees are limited
   a) each level has a degree of at most k
   b) each regular level has degree greater than $k/3 + 1$
2. Bucket sizes are limited
   a) a bucket at level i has size at most $c_i := c(k/2)^i$
   b) a bucket at level i with $deg(\ell_i) > 1$ has size at least $c_i/2\alpha$
3. Bucket and level ordering is consistent with element ordering
   a) items in a regular bucket b_i have smaller keys than those in bucket b_{i+1}
   b) items in regular buckets of a regular level ℓ_i have smaller keys than those in ℓ_{i+1}

3.2.2 Balancing rules

To maintain above invariants, we employ the following balancing rules:

A) Split overflowing buckets If a regular bucket or the last bucket of the last level exceeds its size limit then split it into a sorted α-partition. The resulting buckets replace the original bucket. May trigger rule *B*.

B) Retire buckets into max-buffer when degree overflows If the degree d of a level exceeds k, join its last $d - k + 1$ buckets. May trigger rule *C*.

C) Empty max-buffer into next level when it overflows If max-buffer's size on ℓ_i exceeds its capacity, push all but c_i/α items to the next level. If this happens on the last level, create a new level first. May trigger a prefix of rules *A, B, C* on the following level.

D) Handle degree underflow If the degree of regular level i is decreased to $k/3 + 1$, flush the max-buffer of ℓ_i into ℓ_{i+1} and then pull the first bucket from ℓ_{i+1} into ℓ_i. If that bucket's size exceeds c_i, split it into as many buckets as possible without violating invariants 1a and 2b. May trigger either a prefix of rules *A, B, C* or rule *D* on the following level.

3.2.3 Operations

We define the following operations on the backend.

push Given a level index i and a set of incoming elements S with $|S| = c$ if $i = 0$ else $c_{i-1}/2\alpha \leq |S| \leq 2c_i$. First classify all items from S into their respective bucket. That is, insert each item in S into the first bucket whose pivot compares greater or equal to the item's key. After that, apply balancing rules *A, B* and *C* as necessary.

pull Given a level index i, remove the first bucket from ℓ_i and return it. Subsequently, apply balancing rule D if necessary.

insert/deleteMin To insert a batch of elements into the PQ, call *push* on the first level. Similarly, to remove a batch of smallest elements, call *pull* on the first level.

The minimality of the returned elements is guaranteed, since we removed either a regular bucket, in which case the items are minimal by invariant 3. Or we removed a max-buffer, in which case by invariant 1 we can conclude that the first level must be the last and we removed the last elements from the queue, making them minimal as well.

3.2.4 Invariant Maintenance

This sections shows that all operations on the levels and the priority queue as a whole maintain each of the invariants listed above.

3.2.4.1 Invariant 1: Level degrees

There are two situations in which a level's degree grows. The first is by application of balancing rule A (split overflowing buckets). But if that causes a degree overflow, we apply balancing rule B right after it, reducing the degree to exactly k. The other situation is during application of balancing rule D (refill level from next one). However, in this case we choose the split degree in a way that the level's degree does not overflow afterwards. Thus part (a) of the invariant holds.

We guarantee part (b) by applying balancing rule D whenever a pull has reduced the degree of a regular level ℓ_i to $k/3 + 1$. We first push our max-buffer to the next level ℓ_{i+1}, reducing the degree further to $k/3$. Then we pull a bucket from ℓ_{i+1} and split it into as many buckets as possible. If the pulled bucket was the last bucket of ℓ_{i+1}, then ℓ_{i+1} will be removed as well, making ℓ_i the last level and thus exempt from invariant 1b. Otherwise, invariant 2b guarantees that the split will yield at least $c_{i+1}/2c_i = k/4 > 2$ new buckets. Therefore ℓ_i has degree greater than $k/3 + 1$ after applying balancing rule D and part (b) of the invariant is maintained.

3.2.4.2 Intermission: Sample splits

The easiest way to implement a d-way *Split* of N distinct elements is by sorting the input and then writing the first N/d elements into the first new bucket, the next N/d elements into the second bucket and so on.

Since we only need a sorted d-way partition of the input, our implementation of *Split* uses random sampling to estimate the d-quantiles of the input. We then classify and distribute the input just as we do when pushing items into a level. We will now quantify the probability of a good split.

Lemma 1. *Consider splitting a bucket of size N into d parts, using a sample of size dS with $S \in \Omega(\log N)$. Let r_i be the ratio between the actual size of the i-th resulting bucket and the target bucket size N/d. The probability that $\forall i : 1/2 \leq r_i \leq 4/3$ is greater than $1 - 2/N$.*

Proof. The upper bound for r_i can be proven similarly to [16, Lemma 5.11]. For our result, we actually use tighter bounds like those in [16, Exercise 5.50]. The lower bound of r_i can be proven analogously by reversing some inequalities and applying a different Chernoff bound in the end. Since, given an adequate sample size, the probability of violating either bound is at most $1/n$, the final result can be obtained by applying the union bound. □

If after a few tries we cannot produce a good split as defined above, we fall back to sorting. This should finally explain why invariant 2b allows buckets that are only half the size of a perfectly split max-size bucket.

3.2.4.3 Invariant 2: Bucket sizes are limited

Whenever a bucket overflows after an insertion, we apply balancing rule A to split it into a sorted α-partition. Since buckets resulting from a split are at most $\frac{4}{3\alpha} \leq \frac{1}{3}$ times the size of the bucket being split, we can guarantee the resulting buckets on ℓ_i to be at most c_i if an overflowing bucket can have at most $3c_i$ elements.

Lemma 2. *Let s_i be the maximum number of elements that a PQ level will receive during a push, with $s_0 \leq c$. Then $\forall i \in \mathbb{N}_0 : s_{i+1} < 2c_{i+1}$.*

Proof. Consider a *push* of s_i elements into ℓ_i. In the most extreme case, when all buckets are already full, only $k-1$ items are necessary to let them all overflow. However, at least the $(\sqrt{k}-1)c_i$ items contained in the first $\sqrt{k}-1$ buckets will remain in the level – since even splitting all of them will result in less than k buckets. Subtracting these remaining items from the kc_i items that originally resided in the level, as well as the s_i incoming items, gives the following recursive upper bound for the number of items pushed to ℓ_{i+1}:

$$s_{i+1} \leq s_i + kc_i - (\sqrt{k}-1)c_i = s_i + (k - \sqrt{k} + 1)c_i$$

which can easily be transformed to the closed form of

$$s_{i+1} \leq c + (k - \sqrt{k} + 1) \sum_{j=0}^{i} c_j.$$

To focus on the constant factor, we divide s_{i+1} by $c_{i+1} = c(k/2)^{i+1}$:

$$s_{i+1}/c_{i+1} \leq (2/k)^{i+1} + (k - \sqrt{k} + 1)(2/k)^{i+1} \sum_{j=0}^{i} (k/2)^j.$$

We use the geometric series with $q := k/2$ to provide a constant bound for the part of the last summand that is dependent on i:

$$\frac{\sum_{j=0}^{i} q^j}{q^{i+1}} = \frac{q^{i+1}-1}{(q-1)q^{i+1}} = \frac{1 - 1/q^{i+1}}{q-1} \leq \frac{1}{q-1} = \frac{2}{k-2}.$$

Together with $(2/k)^{i+1} \leq 2/k$, this gives us

$$s_{i+1}/c_{i+1} \leq \frac{2}{k} + \frac{2(k-\sqrt{k}+1)}{k-2} = 2\left(\frac{k-\sqrt{k}}{k-2} + \frac{1}{k-2} + \frac{1}{k}\right).$$

Since $k \geq 16$ this finally yields the desired upper bound

$$s_{i+1}/c_{i+1} \leq 2\left(\frac{12}{14} + \frac{1}{14} + \frac{1}{16}\right) < 2.$$

□

So even if an already full bucket on level i receives all the elements during a push, it will contain less than $3c_i$ elements after the push.

Another violation of invariant 2 might arise during application of balancing rule D, when we split a bucket from ℓ_{i+1} into as many buckets as possible. Note that for any level we can find some successor (still existing or not) which never has pulled any buckets. Since we have just established that invariant 2a is maintained locally, we can use that level for the beginning of an inductive argument. So suppose invariant 2a holds for ℓ_{i+1}, i.e. the incoming bucket's size is at most $c_{i+1} = \frac{k}{2}c_i$. Rule D is only applied if we have enough free bucket slots for a $2k/3$-way split. So even considering that we want to allow our sample split to produce buckets of up to $4/3$ the target size, we can guarantee that the resulting bucket size does not exceed $\frac{4}{3}\frac{3}{2k}\frac{k}{2}c_i = c_i$. Thus, invariant 2a holds for ℓ_i and inductively for the whole queue.

For part (b) we examine all situations in which bucket sizes are reduced. When we apply rule A, we split buckets containing at least c_i elements with unique keys into a sorted α-partition. This gives bucket sizes of at least $c_i/2\alpha$ as desired.

When we apply rule C because ℓ_i's max-buffer overflows, the latter contains more than c_i elements. By definition, we retain enough elements to satisfy the lower bound of the max-buffer size on level i. The items that we push will either go into existing buckets that already satisfy invariant 2b or into the first bucket of the last level which is exempt from the minimum-size requirement.

Finally, for rule D we must again consider two cases for the bucket we pull from the next level. If it is not the last bucket of the last level, it is guaranteed to have size at least $c_{i+1}/2\alpha = \alpha c_i/4$ which is at least $k/2$ times the receiving level's lower buffer size bound. If instead we pull the last level's last bucket, then invariant 2 gives no direct lower bound for its size. However, there are only two possibilities how that bucket came to be. Either it is the result of a split on the next level, in which case above bounds hold. Or it is the initial bucket of that level, in which case it contains at least the items from the initial push from level i. These in turn are guaranteed to be at least $(\alpha - 1)c_i/\alpha \geq c_i/\alpha$ items, which is enough to satisfy part (b) of the invariant.

3.2.4.4 Invariant 3: Consistent ordering

Initially a level contains no elements, so naturally the invariant holds.

During *push*, we classify every element x into a bucket b_i such that $x \leq p(b_i)$ and i is minimal. Since the pivots are the bucket maxima, this maintains the local part (a) of the

invariant. It should be easy to see that balancing during *push* – i.e. replacing a bucket with a sorted partition of itself, joining adjacent buckets and removing a bucket from the end of the level – also maintain part (a) of the invariant.

During *pull*, we remove a bucket from the front of the level. This affects neither part of the invariant. When we apply balancing rule D, we push the max-buffer to the next level, which does not affect the bucket ordering on the current level either.

However, it remains to be shown that the inter-level movement of items maintains the desired invariants as well. For that, first note that in a regular level ℓ_i, the pivot of the second to last bucket b never changes unless the max-buffer is completely emptied. Starting from the moment ℓ_i pushes the first batch of items to its new successor ℓ_{i+1}, we do not split the max-buffer of that level anymore. While a split of b results in a new bucket b' preceding the max-buffer, it holds that $p(b') = p(b)$. Finally, there will always be a preceding bucket, since invariant 2b guarantees that the degree will always stay above $k/3 + 1 > 2$. Thus, of all elements pushed to ℓ_i, exactly those with greater keys than $p(b)$ end up in the max-buffer. They either remain there until the level becomes irregular again or until they are pushed on to the next level. By part (a) of the invariant, all items in regular buckets of ℓ_i have keys of at most $p(b)$. Thus part (b) of the invariant is maintained when pushing items between levels.

Consequently, when we pull items into a regular level ℓ_i, we know that they are greater than those in any regular bucket in ℓ_i. Since we push our entire max-buffer before pulling, we even know they are greater than *all* items in ℓ_i, which maintains part (a) of the invariant. Moreover, since we always pull the entire first bucket from the successor, we know that all pulled items have smaller keys than the ones remaining in ℓ_{i+1}. Finally, if the pulled bucket is split, then this operation resets the pivot between regular buckets and the max-buffer of the pulling level to the key of one of the pulled elements. Since this pivot is less than any elements in ℓ_{i+1}, further pushes will continue to maintain part (b) of the invariant by the same argument as above.

3.3 Frontend

As mentioned above, the frontend provides a priority queue interface by acting as a buffering adapter on top of the batched priority queue described in the previous section. It does so by following the same principles as the backend. The elements which it contains are kept in an ordered partition $b_{\min} \dot\cup b_{\max}$ also called *min-buffer* and *max-buffer*. $b_{\min}$ is kept in min-heap order and always contains the smallest items in the whole PQ – frontend as well as backend. That also means it is only empty when the whole PQ is. Both buffers are guaranteed to contain at most c elements and are initially empty. The frontend's pivot p marks the upper bound of $b_{\min}$ and is initialized to ∞.

3.3.1 Operations

insert Given an incoming element x, we first classify it into the appropriate buffer based on the pivot. If that buffer has reached its maximum size c, we flush it into the backend. For $b_{\max}$ this simply means pushing all items contained in it to the first level of the backend.

For $b_{\min}$ we perform a ordered α-way split, keep the first of the resulting buckets and insert the others in front of all other buckets in the first backend level. Furthermore we set p to the maximum key of the new $b_{\min}$. Finally, if x was inserted into $b_{\min}$, we restore heap order in it.

min Return the first element in $b_{\min}$. The minimality of that element follows directly from the fact that $b_{\min}$ contains the smallest elements and is kept in min-heap order.

deleteMin We first remove the first item from $b_{\min}$. Then, if the queue still contains elements but $b_{\min}$ is empty, we have to refill it. If the backend is empty as well, all remaining items must be in $b_{\max}$ and we simply swap the two buffers and reset p to ∞ to do that. If however the backend still has elements left, we pull its first bucket and use that as the new $b_{\min}$ and the bucket's pivot as the new frontend pivot. To ensure that $b_{\min}$ contains all elements with keys less than or equal to p, we split $b_{\max}$ using the new pivot and add all items with smaller keys to $b_{\min}$. If that causes $b_{\min}$ to overflow, we flush all but a fraction of $1/\alpha$ of its elements to the backend, just as we do during *push*. In any case, we restore heap-order for the elements in $b_{\min}$.

4 Analysis

In this chapter we will show that the amortized expected complexity of our algorithm is optimal in the external memory as well as the RAM model.

4.1 Backend

We will first bound the number of levels in the PQ and analyze the complexity of the basic building blocks *Classify*, *Split* and *Retire*. We will then proceed to use these results to bound the complexity of recursive *push* and *pull* operations.

4.1.1 Height

We call the number of levels in the PQ its *height*. We furthermore will refer to it as h. Since, due to invariants 1b and 2b, a regular level i contains at least $\frac{k}{3}\frac{c_i}{2\alpha} = \frac{c}{3\alpha}(k/2)^{i+1}$ items, a PQ consisting of h levels has size at least

$$N \geq \frac{c}{3\alpha}\sum_{i=0}^{h-2}\left(\frac{k}{2}\right)^{i+1} > \frac{c}{3\alpha}\left(\frac{k}{2}\right)^{h-1}$$

which means we can equivalently specify the upper bound for the number of levels in terms of items as follows:

$$h < 1 + \log_{k/2} 3\alpha N/c \in O(\log_k N/M). \tag{4.1}$$

4.1.2 Classify

Whenever we push elements to a level or split an existing bucket into several new ones, we need to classify each of $N \in \Omega(M)$ items as belonging to exactly one of $d \in O(M/B)$ classes. The classes are defined by a sorted sequence of keys – the bucket pivots. We first use the pivots to build an implicit classification tree, if we not already have one from a prior classification that used the same pivots. This can be done in time $O(d)$. We can then use the implicit classification tree to determine the class of an item in time $O(\log d)$.

During this process, we require one block of memory as a read buffer, $O(d/B)$ blocks for the implicit classification tree as well as $O(d)$ blocks of memory that act as write buffers – one for each class. This adds up to a memory requirement of $O(M)$, which means that we can keep everything in fast memory if the constant factors are chosen carefully. Thus, the N input elements can be efficiently streamed through the classifier. This yields a worst case bound of $Scan(N)$ I/Os for the streaming of the elements plus another $O(d) = O(M/B)$

I/Os to write any incomplete write buffers. Thus we can classify $N \in \Omega(M)$ elements with worst-case costs of

$$Classify_d(N) = \begin{cases} \Theta(N \log d) & \text{time} \\ \Theta(N/B) & \text{I/Os.} \end{cases} \tag{4.2}$$

4.1.3 Split

To split a bucket of $N \in \Omega(M)$ items into d new buckets, we first draw a random sample of size dS with $S \in \Theta(\log N)$. This requires internal work and memory accesses linear in the sample size. After that, we sort the sample for costs $Sort(dS)$ to estimate the input's d-quantiles which we will use as the pivots for the new buckets. Finally, we classify the input elements according to those pivots and distribute them to their respective new buckets for a cost of $Classify_d(N)$ which also dominated the total cost of the sample split.

To ensure the bucket size invariants we declared in the previous chapter, we fall back to sorting the input if the resulting bucket sizes are not all within the bounds that are used in Lemma 1. Thanks to that event's low probability, this only adds an asymptotically insignificant $Sort(N)/N$ to the expected costs which thus remain at $Classify_d(N)$ while limiting the worst-case costs to $Sort(N)$. We will refer to this variant as $Split_d$ and use it henceforth.

4.1.4 Retire

When the level degree overflows we retire buckets, i.e. we join them with the max-buffer. Since we implement buffers as contiguous segments of memory, retiring a bucket simply means copying its entire contents into the max-buffer. Thus, retiring a bucket of size N incurs a cost of $Retire(N) = Scan(N)$. Since only items from regular buckets can be retired, and elements only enter regular buckets by means of classification, we can cover all retire costs by charging an extra $Retire(N)/N$ for each element classification.

4.1.5 push

When pushing a batch of $\hat{N} \in \Omega(M)$ items into some level i, each item is first classified into the appropriate bucket for a cost of at most $Classify_k(\hat{N})/\hat{N}$ per item. We then apply balancing rules A, B and C as necessary.

It follows directly from Lemma 2 that, when we apply rule A to split an overflowing bucket into α new buckets, the overflowing bucket has size at most $3c_i$. Consequently, to let all of the α new buckets overflow again, at least $\alpha c_i - 3c_i$ elements have to be inserted into them after the split. Since existing regular buckets only receive newly pushed items, we can conclude that one split occurs at most for every $c_i(\alpha - 3)/\alpha \geq c_i/4$ items that are pushed into a level. Thus, the amortized split costs during pushes to level i amounts to $O(Split(3c_i)/c_i)$ per element.

The costs for any retirements caused by rule B are already covered, as we discussed above. Thus, before application of balancing rule C, the amortized expected cost of pushing

an item into level i is

$$\mathbb{E}[push] = \begin{cases} O(\log k) & \text{time} \\ O(1/B) & \text{I/Os.} \end{cases} \tag{4.3}$$

Rule C simply pushes items on to the next level. Since any item is pushed at most once into each level, the backend has a total amortized expected push cost per item of

$$O(\log_k N/c) \cdot \mathbb{E}[push] = \begin{cases} O(\log N/c) & \text{time} \\ O(1/B \log_k N/c) & \text{I/Os.} \end{cases} \tag{4.4}$$

The amortized worst-case costs can be obtained in the same manner. Like the expected costs, they are dominated by the term $Split(3c_i)/c_i$ which has worst-case bounds of

$$push_i = Sort(3c_i)/c_i = \begin{cases} O(\log c_i) = O(i \log c) & \text{time} \\ O(\frac{1}{B} \log_{M/B} \frac{c_i}{B}) = O(i/B) & \text{I/Os.} \end{cases} \tag{4.5}$$

By the same argument as above, we arrive at a total amortized worst-case push cost per item of

$$\sum_{i=0}^{h-1} push_i = \begin{cases} O(\log c \cdot \log_k^2 N/c) & \text{time} \\ O(\frac{1}{B} \log_k^2 N/c) & \text{I/Os.} \end{cases} \tag{4.6}$$

Note that we the time bound is deliberately coarser than it could be to keep it simple.

4.1.6 pull

When we remove the first bucket from a level, we have to update the list of buckets for that level. Assuming we are using contiguous memory to store references to buffers, the cost per pull is $O(Scan(k))$. This is easily amortized for levels beyond the first, where at least $\Omega(\alpha c) = \Omega(k\sqrt{MB})$ items are pulled at a time. If we have $M \in O(B^3)$ then the costs are also amortized constant on the first level. Alternatively a linked list can be used instead to get worst-case constant costs for removal of a bucket reference. Consequently, everything we do during a pull from a given level (not considering a potential refill from the next level) can be done with constant costs per item if implemented appropriately.

Refills from the next level occur at most every $\Omega(2k/3)$ pulls. Let us first consider recursive pulls only. Any item that we pull from the first level of the queue has been pulled from at most h levels in total. In fact, as we have used before, $\sum_{i=0}^{h-1} c_i/c_h \leq 2/(k-2)$ for all h. So only a small fraction of all items is not contained in the last levels. Thus, charging all items to be pulled through all levels only introduces a negligible error. Since the dominating costs for pulling items from level i is splitting the pulled bucket into smaller ones, the costs per item are given by $pull_i = Split(c_i)/c_i = push_i$. Consequently, the cost for pulling an item through all levels is asymptotically the same as for pushing it through all levels. Adding the cost of another recursive push before every refill (i.e. every $\Omega(2k/3)$ pulls) does not change the asymptotic bound.

4.2 Frontend

The whole frontend has size $\Theta(M)$ and thus fits into memory if the parameters are chosen appropriately. Communication with the backend only happens at most every $\Omega(M)$ operations, so even if we evict the entire frontend for that, we only incur another $Scan(M)/M = 1/B$ per element. Thus, usage of the frontend does not change the amortized asymptotic I/O bound, even if it only barely fits into memory.

4.2.1 insert

The cost for classifying each incoming item into the min- or max-buffer is constant. If we insert into the min-buffer of size at most $c \in \Theta(M)$, we pay another $O(\log M)$ for pushing to the binary heap.

Flushing to the backend incurs the push costs per item given above. When flushing the max-buffer this is obvious. When flushing the min-buffer, we split it into a sorted α-partition and then prepend all but the first bucket to the first level. The cost for this is less than a regular push to the first level. Thus the total amortized expected time of inserting an element into a queue of size N is

$$O(\log N/M) + O(\log M) = O(\log N) \tag{4.7}$$

4.2.2 deleteMin

We pull each item from the backend, add constant costs for it being part of a *makeHeap* and finally $O(\log M)$ for being pulled from a heap of size at most $c \in \Theta(M)$. The costs for classifying the items in the max-buffer when we refill the min-buffer is also constant per item. Finally, the overflow handling for the min-buffer during deletion is identical to that during insertions and thus incurs at most the cost of a regular push to the backend. For a queue containing N elements, this gives a total amortized expected *deleteMin* time of

$$O(\log N/M) + O(\log M) = O(\log N) \tag{4.8}$$

5 Implementation

In this chapter we will give a rough overview of the most important aspects of our implementation. After that we will discuss deviations from the theoretical algorithm and their implications. And finally we will examine some key points of the implementation that are responsible for greatly reducing hard-to-predict conditional branches.

5.1 Overview

Our implementation consists of several portable C++17 class templates which allow for flexible configuration of key and value type as well as the various tuning parameters. Where at all possible, all parameters are made available at compile time to provide maximum information to the optimizing compiler.

Our implementation tries to avoid hard-to-predict conditional branches wherever feasible. To that end, it also makes use of sentinel values for keys in several places, so the smallest as well as the largest value of the chosen key type cannot be used as item keys. For example, items using IEEE floating point keys may have neither positive nor negative infinity as their key value.

The `Bucket` class is essentially a `std::vector` of elements accompanied by the bucket's pivot. All buckets on a single level are again kept in a `std::vector`. The list of levels is kept as a `std::deque`. All data structures use the default `std::allocator`. The frontend uses vectors for its buffers as well. The source code of our implementation and evaluation is available at `github.com/raphinesse/s3q`.

5.2 Differences to analyzed algorithm

In this section we describe various instances where our implementation deviates from the algorithm as described and analyzed in the previous chapters.

5.2.1 Relaxed unique-key requirement

In the previous proofs and analyses we strictly required all items to be uniquely keyed. This is convenient for the theoretical analysis but impractical for real-world usage. In fact we experienced this first-hand during our evaluation with monotonous queue workloads, where keys were drawn from an exponential distribution.

To address this practical issue, our implementation allows *some* duplicity among the keys of the items in a queue. More specifically, we only require overflowing buckets to contain enough unique keys to be able to properly split them without any special handling.

Of course, duplicate element keys are actually a blessing in disguise, since they are always sorted among themselves. See chapter 7 for a potential way of harnessing this.

5.2.2 Fixing failed sample splits

Our implementation does not fall back to sorting the input bucket after a failed sample split. Instead it makes a single pass over the split result to join any undersized bucket onto any of its neighbors. After that, it recursively splits any of the new buckets which might be overflowing.

This approach might be able to yield better constant factors than falling back to a sort, but we have no data to back that conjecture. Unfortunately, it is harder to reason about the number of buckets resulting from a split. It could be both less as well as more than α. Thus it would be desirable to implement the sort-fallback variant as well to compare both approaches in practice. The reason we did not do so is simply that – for a fair comparison – various other changes would have to be made to the current implementation.

5.2.3 Order of bucket splits & joins

In the algorithm described in the previous sections, we first split any regular bucket that overflows. Afterwards, if the level contains too many buckets, the last few are retired into the max-buffer. This approach is conceptually simple but it has downsides.

First of all, when we split a bucket b_i with $i > k - \alpha$ (we say i is in the *dead zone*) into α new buckets, we immediately have to retire some of the new buckets right after creating them since there are just not enough bucket slots in the level after i. This is obviously wasteful and should be avoided. Secondly, each level has to be able to hold not k but $k + \alpha$ buckets in memory – at least until the overflow has been fixed.

To mitigate these issues our implementation reverses the order of splits and joins. If it is known that performing a split and thus increasing the level degree by α will make the level overflow, then we retire as many buckets as necessary before actually performing the split. For overflowing buckets in the dead zone this means we do not split them at all but retire them and any following buckets into the max-buffer instead.

This way we neither need extra space to accommodate overflow buckets nor do we invest valuable resources in creating item partitions that will be discarded right after their creation. However, this approach also has a minor drawback that we need to consider. We might actually reduce our degree to below k when we retire overflowing buckets in the dead zone. Consequently, we have to flush the max-buffer more often and such a flush can contain more items than with the split-then-join variant. However, the analysis we performed in the previous chapter considers these facts and our results hold either way.

5.3 Classification

For fast and predictable classification of elements into their respective buckets we use the same approach as (In-Place Parallel) Super Scalar Samplesort [15, 2]. In fact, we actually use the `Classifier` class from the IPS^{4}oimplementation.

The basic idea is to build an implicit binary search tree from the sorted list of splitters. Much like an implicit binary heap, this tree resides in contiguous memory where index 1 is the root and the children of the node at index i are at indices $2i$ and $2i + 1$. When the tree is small enough to fit into fast memory – which it is in our case – its traversal is extremely performant. Furthermore, when determining the next child during a key lookup, we do not use any conditional branches but instead perform index arithmetic involving the result bit from the key comparison. Together with other optimizations like loop unrolling and interleaved classification of multiple elements to stagger data dependencies, this approach provides a very high classification throughput. For further details, we refer to the respective publications and implementations [15, 2].

The drawback is that we need extra memory for the search tree and that we have to rebuild it anytime buckets are added to or removed from a level. To avoid doing this more often than we have to, we mark the classificator instance as invalid whenever the pivots change and rebuild it only when it is used the next time. This way we keep the implementation simple and still avoid multiple rebuilds between classifications. The additional conditional branch before each classification is easily amortized since we always classify $\Omega(M)$ elements at once.

5.4 Frontend Priority Queue

Every element that is removed from a S^3Q instance passes through the priority queue buffer in the frontend. Since this is an inherently non-linear operation, it should be obvious that the implementation quality of this priority queue is paramount.

5.4.1 Predictability

Katajainen's experimental survey in search of the best priority queue [10] suggests that for small queue sizes, branch-optimized versions of Williams' implicit binary heap [19] provide the best performance on current hardware.

Sanders' Sequence Heap publication [14] includes a very detailed analysis of his implicit binary heap implementation which is used for solving a problem that is virtually identical to ours. This implementation makes use of the bottom-up heuristic [18] which – compared to a textbook implementation – already reduces the number of branches based on the outcome of key comparisons considerably. According to [6, 10], another worthwhile branch-reduction optimization is replacing the remaining branch in the body of the *siftDown* loop, which is part of *deleteMin*, by address arithmetic involving the comparison result – just as we do for traversing our classification tree. Our implementation largely borrows from Sanders' but also applies that additional optimization. This indeed eliminated the conditional jump instruction `jl` in the loop body and replaced it with the `setl` instruction – which loads the last comparison result into a regular register – followed by some address arithmetic. See Figure 5.1 for a comparison of both *siftDown* implementations. On top of that we also included the best performing `makeHeap` variant from [6] which helped us to further reduce branch mispredictions but did not have a significant impact on the algorithm's overall performance.

```
1  int hole = 1;
2  int succ = 2;
3  while (succ < size) {
4    Key key1 = data[succ].key;
5    Key key2 = data[succ+1].key;
6    if (key1 > key2) {
7      succ++;
8      data[hole].key = key2;
9      data[hole].val = data[succ].val;
10   } else {
11     data[hole].key = key1;
12     data[hole].val = data[succ].val;
13   }
14   hole = succ;
15   succ <<= 1;
16 }
```

Sanders' version

```
1  Idx hole = 1;
2  Idx succ = 2;
3  while (succ < size) {
4    // Set succ to the index of the
5    // successor with the smaller key
6    // without conditional jumps
7    succ += data[succ+1] < data[succ];
8
9    // Move the smaller of the
10   // successors up by one level
11   data[hole] = data[succ];
12
13   // Update hole & succ indices
14   hole = succ;
15   succ <<= 1;
16 }
```

Our version

Figure 5.1: Comparison of the *siftDown* loop from the binary heap used in Sanders' Sequence Heap on the left and our optimized and simplified version on the right. Since `data` is 1-indexed, `size` is also the index of the last element. The `<` operator used to compare heap elements compares the keys of those elements.

5.4.2 Details

Sanders reported about the inability of the compilers that were available at the time to properly optimize expressions involving member variables. Fortunately, in the last 20 years optimizing compilers apparently matured enough so that we do not have to worry about that anymore. This allowed us to simplify and shorten the code in some places. Nevertheless, careful examination of the generated assembly code for our programs can of course still reveal optimization opportunities. This is especially true for performance-critical parts as the following example demonstrates.

One very subtle yet interesting performance bug we uncovered this way was related to the integer type used for array indices – `int` in case of the original implementation. Being written before the advent of 64-bit mainstream machines, this probably was no issue then. On the x64 machines we used for benchmarking however, `int` is mapped to a signed 32-bit integer. To use this type as an offset in 64-bit address arithmetic a sign extension has to be performed first. It should be obvious that adding constant costs to every index calculation is undesirable when they are at the heart of the heap traversal that is an integral part of our hot path. Removing either the signedness or the size restriction of the index type is enough to resolve this issue. For our implementation this was an important step in leveraging the performance potential of the additional branch-reduction optimization described above.

Figure 5.2 shows the impact of the index type on the optimized assembly generated by our compiler for the loop body of *siftUp*. The additional sign extension necessary when using double-word signed integer semantics not only doubles the instructions per iteration used for address calculation, it also wastes an additional register and introduces an additional read-after-write hazard. Of course, the same principle holds for the other heap traversal loops.

```
1 Idx pred = hole >> 1;           // Idx is a type alias to an integer type
2 while (el < data[pred]) {       // must terminate due to sentinel at 0
3   data[hole] = data[pred];
4   hole = pred;
5   pred >>= 1;
6 }
```

C++ source code

```
1  .loop:
2  ; data[hole] = data[pred]
3  mov    rdx, qword ptr [rdx]
4  mov    qword ptr [rcx], rdx
5
6  ; update pointers & index
7  lea    rcx, [8*rdi + data]
8  sar    rdi
9  lea    rdx, [8*rdi + data]
10
11
12
13 ; if data[rdi] > el: repeat loop
14 cmp    dword ptr [8*rdi + data], eax
15 jg     .loop
```

Assembly generated for 64-bit signed `Idx`

```
1  .loop:
2  ; data[hole] = data[pred]
3  mov    rdx, qword ptr [rdx]
4  mov    qword ptr [rcx], rdx
5
6  ; update pointers & index
7  movsxd rcx, edi
8  sar    edi
9  movsxd rsi, edi
10 lea    rdx, [8*rsi + data]
11 lea    rcx, [8*rcx + data]
12
13 ; if data[rsi] > el: repeat loop
14 cmp    dword ptr [8*rsi + data], eax
15 jg     .loop
```

Assembly generated for 32-bit signed `Idx`

Figure 5.2: Optimized assembly code generated by `clang 11` for the loop of *siftUp* (shown at the top). Note the additional sign extension instructions (`movsxd`) and register (`rsi`) required by the 32-bit index version on lines 8–12. Registers contents: `rcx`: address of hole, `rdx`: address of predecessor, `{e,r}di`: index of predecessor.

6 Evaluation

This chapter discusses in detail the methodology employed in the evaluation of our implementation and the results thereof.

6.1 Benchmark Suite

We benchmarked our implementation along with several other competitors using a common benchmark suite of different workloads. The combination of competitor and workload is known at compile time to allow for adequate optimization by the compiler. We compile each combination to a separate binary, so that the benchmarks are completely isolated and memory leaks from one cannot influence another.

The resulting binary runs the benchmark for input sizes N close to the powers of two from 2^7 up to 2^{27} and prints the results to stdout. For each N, the results include execution duration (measured via `std::chrono::steady_clock`) of the benchmark, as well as various metrics collected via Linux' *perf*. The latter are obtained from the CPU's performance counters and include metrics such as number of cycles, instructions, branch-mispredictions or cache misses. It should be noted that these metrics are inherently imprecise since they are obtained via sampling. On the other hand, they represent what is happening on real hardware – in contrast to simulating profiling solutions like *Cachegrind*.

Our priority queue as well as our benchmarks are all single-threaded. Consequently, when running such a benchmark on an otherwise unloaded modern multi-core processor, the processing core can use all shared resources, most notably a potentially huge L3 cache, all by itself. This could mask the performance impact of different memory access strategies and does most likely not represent the usual state of a system that runs high-performance applications. Thus, when running a certain benchmark, we spawn as many copies as there are physical cores in the system. This way, all cores are similarly loaded most of the time and have to share their resources like they would in a real-world high-load scenario.

Each benchmark configuration (competitor, workload and N) is repeated until the total execution time exceeds one second, but at least once per physical CPU core. The results we report below are the arithmetic means over all runs of a configuration.

6.2 Benchmark Environment

All benchmark binaries were compiled using GCC 9.3.0-17ubuntu1~20.04 using `-O3` and run under Ubuntu 20.04.2 LTS using Linux 5.4.0-72-generic on otherwise unloaded machines. We decided against using `-march=native` for the sake of simplicity and because in early experiments we found that the 0.9-quantile of the speedup across all benchmark configurations was only 1.05.

We include benchmark results from two machines, both of which use a x86-64 CPU. The table in Figure 6.1 compares their most important hardware parameters. We will refer to the machines by their CPU vendor and microarchitecture from here on.

	Intel Xeon E5-2650 v2	AMD Athlon II X4 635
Microarchitecture	Ivy Bridge EP	K10
Launch Year	2013	2010
Clock Frequency	2.6 GHz–3.4 GHz	2.9 GHz
Cores (Threads)	8(16)	4(4)
L1D Cache	32 KiB, 8-way	64 KiB, 2-way
L2 Cache	256 KiB, 8-way	512 KiB, 16-way
L3 Cache	20 MiB, 20-way	–
Cache Line Size	64 B	64 B
Main Memory	128 GiB	12 GiB

Figure 6.1: Characteristics of the hardware used in our experiments. For the caches, the table shows size and associativity. The L3 cache is shared among all cores while the others are exclusive per core.

6.3 Workloads

The workloads we used for our benchmarks can be characterized in two dimensions. One is the key distribution of the elements inserted into the priority queue. The other is the access pattern, i.e. the sequence of operations performed on the priority queue. All configurations use 32-bit keys and 32-bit values for a total element size of 8 Byte.

Our most simple workload is *Random Sort*, where we first *insert* N elements and then call *deleteMin* N times to obtain the sorted input sequence. The element keys are 32-bit integer keys which are drawn uniformly at random from the entire available key space (except minimal and maximum values, because they are used as sentinels).

The other two workloads use the *Wiggle* access pattern: N sequences of *insert* (*deleteMin insert*)^s followed by N runs of *deleteMin* (*insert deleteMin*)^s. In early development we regularly benchmarked wiggle workloads for multiple values of s. However, we found that the effects which can be observed for e.g. $s = 4$ already occur for $s = 1$ albeit less pronounced. Thus we only include results for $s = 1$.

In logical succession of above sort workload, which can be seen as having a Wiggle access pattern with $s = 0$, we include *Random Wiggle* with $s = 1$ which uses the same key distribution as Random Sort. Note that the Random Wiggle workload has properties that make it arguably easier than the sort workload. For every step of the filling process, we insert two random keys into the queue and then remove the minimum key from the queue. As N increases, the distribution of the keys in the queue becomes is skewed towards larger keys. In turn, the probability of a *deleteMin* operation removing a recently inserted key becomes higher. This temporal locality property can be used to process the problem more efficiently. See [11] for more details.

The last workload, *Monotone Wiggle*, is one that could be processed by a monotonic priority queue. It has the same access pattern as Random Wiggle but its keys are single-precision floating-point numbers with a non-uniform distribution. When a new element is to be inserted, its key is determined by adding a random offset X to the largest key that has been removed from the queue so far. X is exponentially distributed with rate 1. The motivation for including this workload is to model access patterns that might occur in discrete-event simulation applications.

6.4 Competitors

We include results for a limited number of well-tuned algorithms only. The reason is simply that this work is not an experimental survey and finding or even producing well tuned implementations of published articles is a very time-intensive task.

S^3Q and S^3QBH For the parameters k and c of our implementation of S^3Q we experimented with multiple values in the neighborhood of their theoretical optima for the respective machines. Across both machines, the best results could be obtained with a maximum level degree of $k = 64$ and a base buffer size of $c = 2^{15}$ elements.

The tuned implicit binary heap implementation that we described in the previous chapter and also use in the frontend of S^3Q is included in the experiments as *S^3QBH*. It operates on a dynamically growing `std::vector`.

StdQueue Aside from our own tuned binary heap, the `std::priority_queue` that comes with our version of GCC was an obvious choice for inclusion. Like S^3QBH, it implements an implicit binary heap in a `std::vector` using the *bottom-up heuristic* [18]. Unlike our implementation it uses neither a sentinel at index 0 nor the branch reduction optimization in the body of the *siftDown* loop.

SeqHeap Another obvious choice is Sanders' original implementation of his Sequence Heap algorithm. We used the code from `algo2.iti.kit.edu/sanders/programs/spq` with the preset parameter settings: deletion buffer size $m' = 32$, insertion buffer size $m = 512$ and maximum merge arity $k = 64$. These parameter choices were confirmed to us as sensible choices by the code's author and the few parameter variations we tested did not lead to obvious improvements in our experiments.

SeqHeap+ Since the insertion buffer employed by the Sequence Heap greatly influences the latter's performance [14] we also include a Sequence Heap variant that has the optimizations from S^3QBH applied to its binary heap implementation.

Omissions Our initial experiment setup also included a 4-ary heap implementation. However, our experiments confirmed what Sanders' experiments already suggested: when the input is large enough for 4-ary heaps to beat binary heaps, they are both outclassed by Sequence Heaps. We also very briefly experimented on an implementation of B-Heaps [9], but we could not produce satisfactory results in the limited time we alloted.

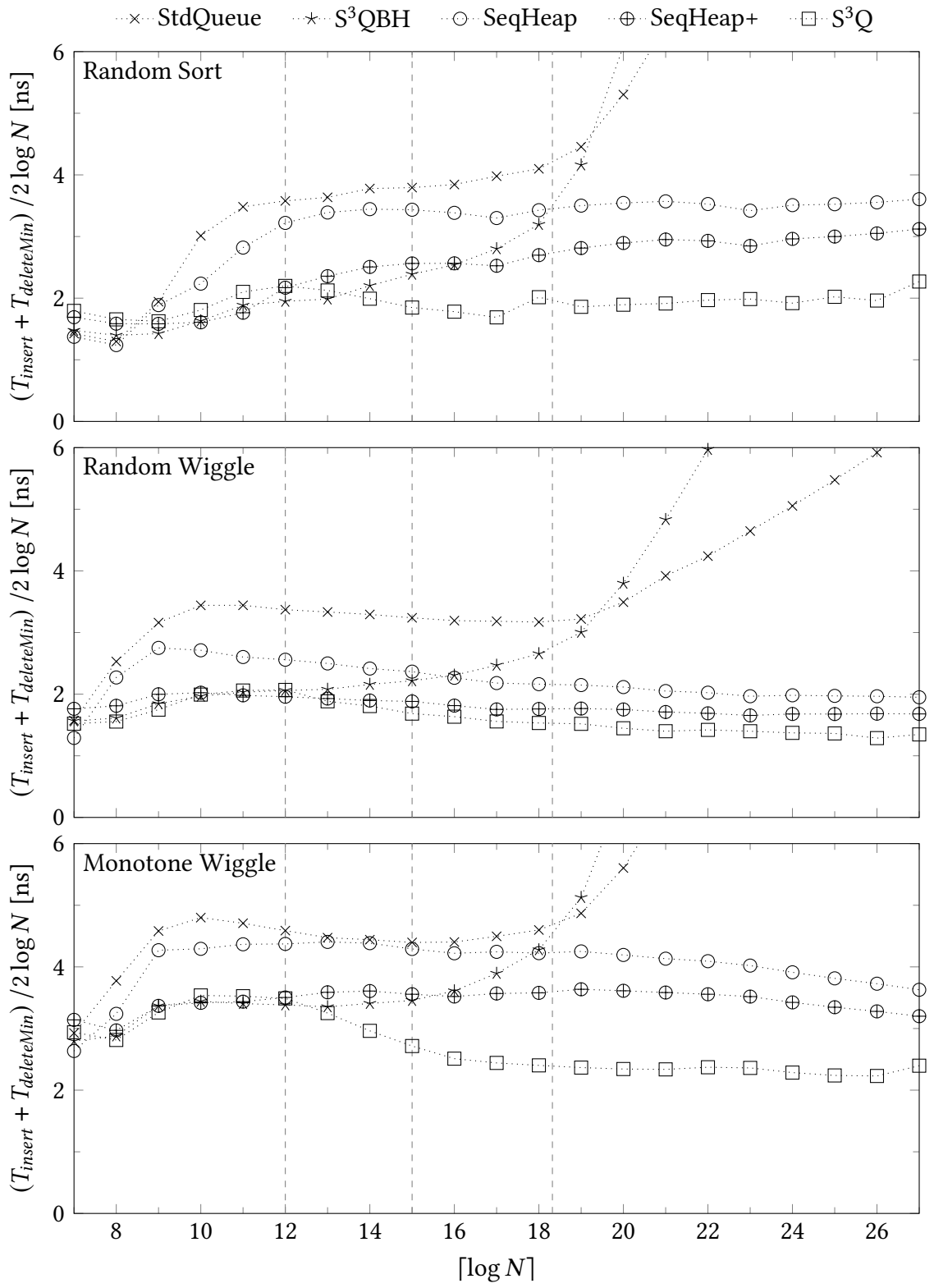

Figure 6.2: Performance on Intel Ivy Bridge EP. The dashed vertical lines signify the L1, L2 and L3 cache capacity in elements available to each core.

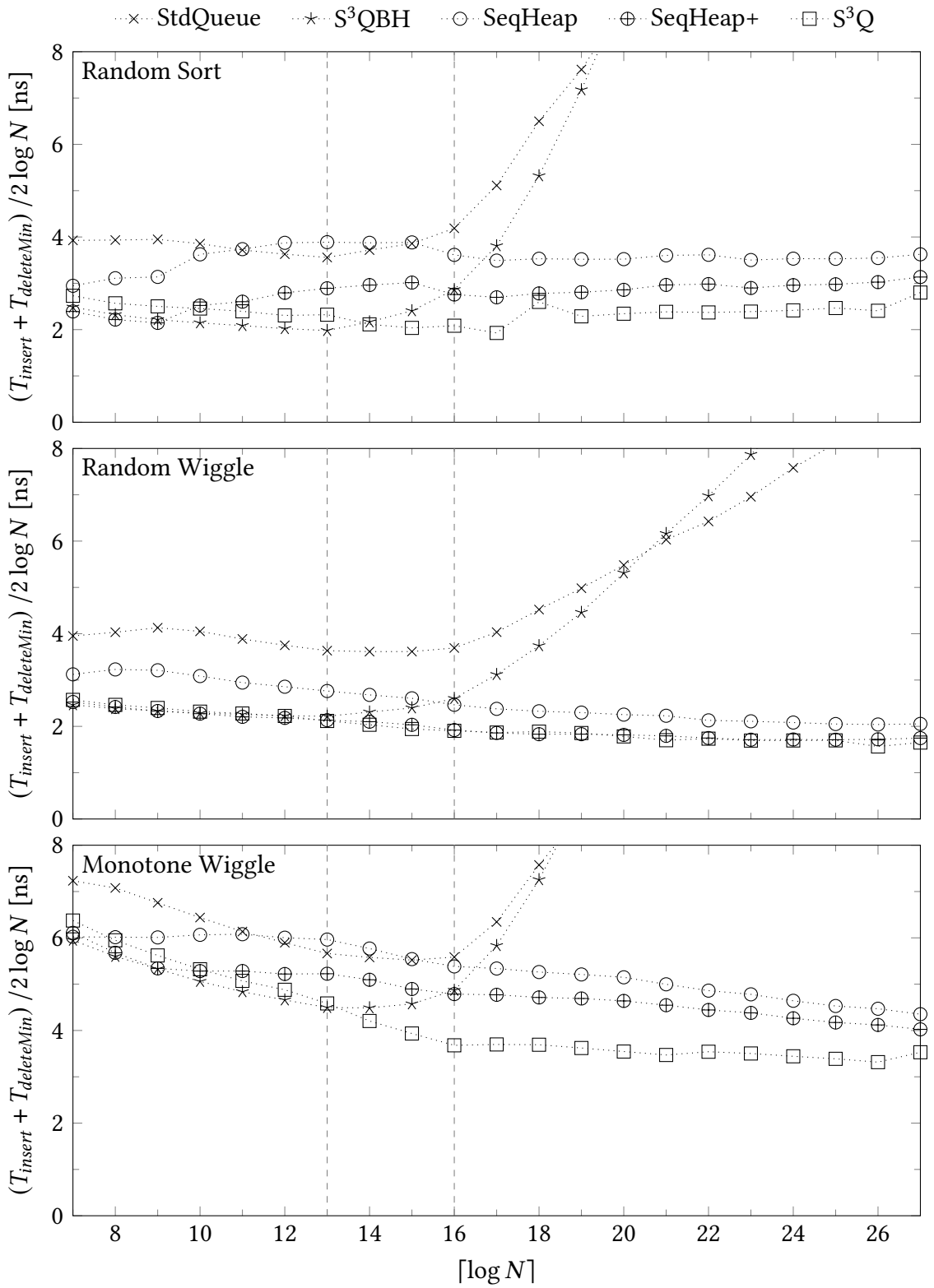

Figure 6.3: Performance on AMD K10. The dashed vertical lines signify the L1 and L2 cache capacity in elements available to each core.

6.5 Results

Figures 6.2 and 6.3 display the execution time for all of the workloads and competitors described above. The average time per operation $(T_{insert} + T_{deleteMin})/2$ is scaled by the expected number of comparisons needed per operation, which is $\log N$ for all included algorithms. Consequently, if an algorithm operates efficiently its curves should remain vaguely horizontal for the most part.

6.5.1 Superscalar Sample Queue vs Sequence Heap

It can be seen that our implementation of S^3Q – much like SeqHeap – shows exactly this behavior. Moreover, it is able to outperform SeqHeap for all input sizes beyond 256 elements across all workloads and machines. Compared to the original Sequence Heap implementation, S^3Q provides a speedup between 1.2 and 2 with a mean of 1.5 for input sizes greater than 2^{10} elements.

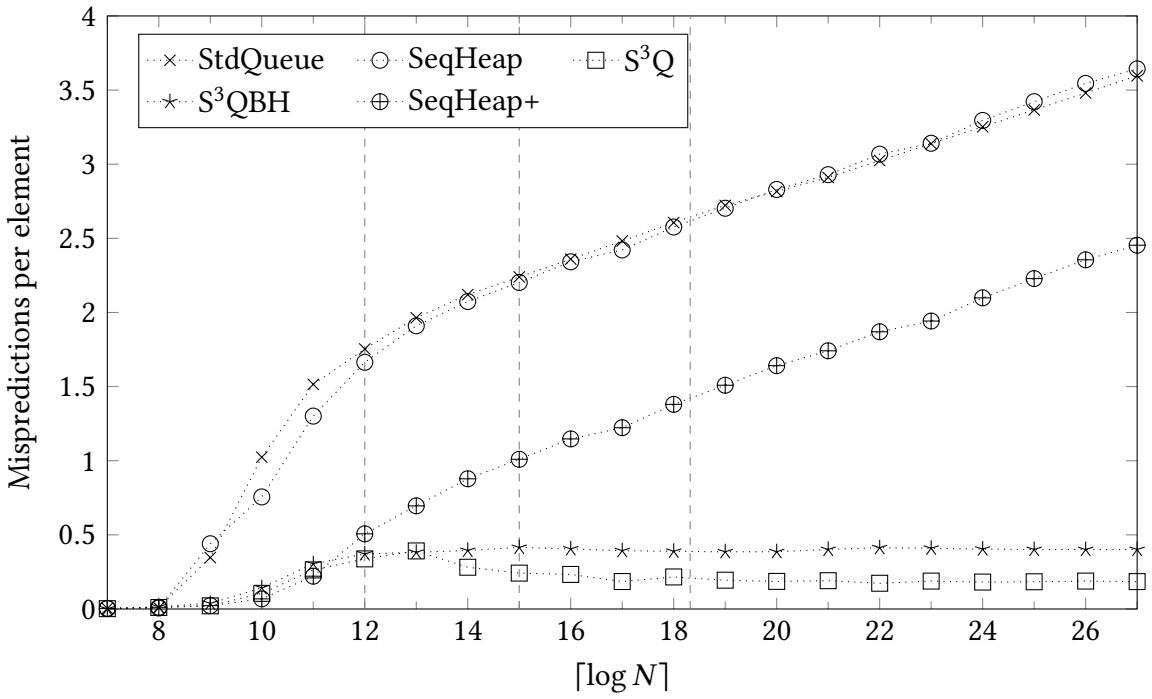

Figure 6.4: Branch mispredictions per element during sorting of N elements on an Intel Ivy Bridge EP. The dashed vertical lines signify the L1, L2 and L3 cache capacity in elements available to each core.

We attribute this largely to the reduced number of branch mispredictions – and related pipeline stalls – caused by S^3Q. This is backed by our measurements which are depicted in Figure 6.4. While on average, each operation of SeqHeap causes mispredictions logarithmic in the number of elements, S^3Q only causes a low constant number of mispredictions per operation.

What emphasizes the impact of reducing branch mispredictions even more is the fact that S^{3}Q exhibits less efficient cache usage than SeqHeap as can be seen in Figure 6.5. For inputs greater than or equal to 2^{18} elements, S^{3}Q causes between 0.9 and 2 times (average 1.4) as many cache faults as Sequence Heap implementations on the AMD processor. On the Intel CPU that value is even higher with values between 2.8 to 5 and a mean of 3.7.

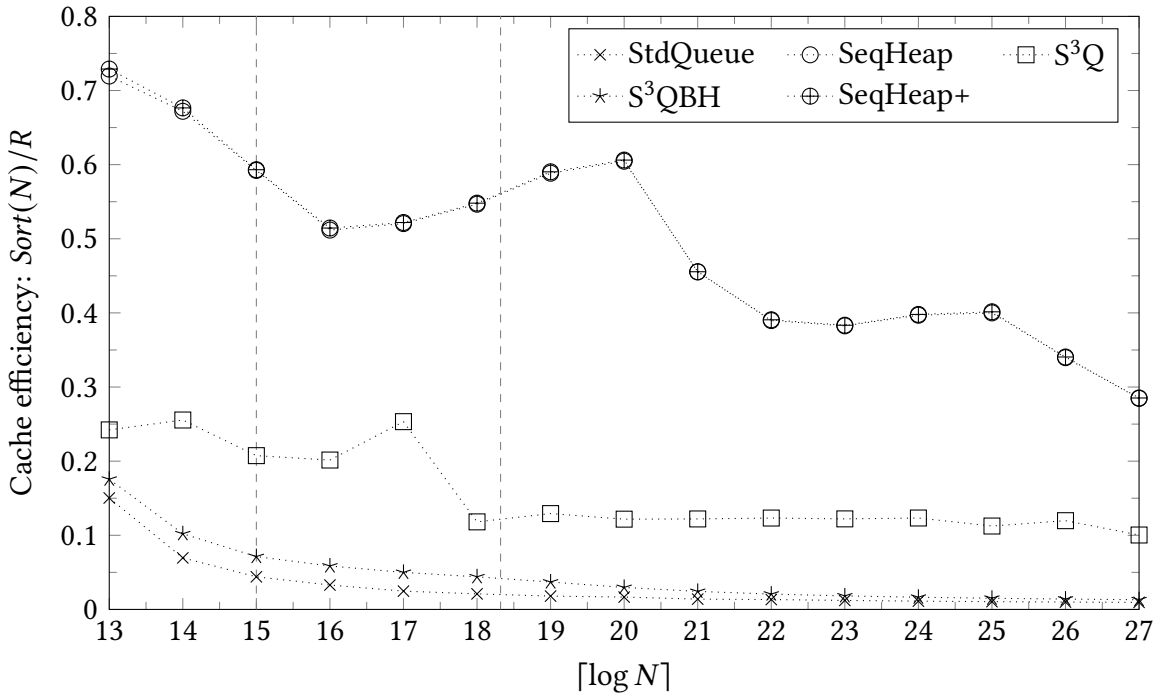

Figure 6.5: L1 data cache read efficiency during sorting of N elements on Intel Ivy Bridge EP expressed as the quotient of $Sort(N)$, the lower bound of I/Os required for sorting in the external memory model, and R, the number of L1 data cache *read* misses we observed. The dashed vertical lines signify the L2 and L3 cache capacity in elements available to each core. Note that only input sizes that are larger than the L1 data cache are depicted.

6.5.2 Binary Heaps and Branch Prediction

Compared to min $\{$S^{3}QBH, StdQueue$\}$ our S^{3}Q implementation obtains speedups that are roughly logarithmic in N for $N \in [2^{13}, 2^{27}]$, finally reaching values between 4.6 and 7 for the largest instances. For smaller instances that fit into the cache, we can see that S^{3}Q only incurs little overhead over S^{3}QBH since their performance is very similar.

For inputs larger than the first level cache, and even more after crossing the last level cache boundary, the performance of both binary heaps quickly deteriorates. In line with the observations from [10], the branch reduced binary heap is more efficient for small inputs but as its size grows beyond the cache capacity its performance drops even faster than that of its counterpart. As [10] states, this effect is most curious. Especially since –

according to our measurements – the optimized version uses less instructions, causes 4 to 8 times less branch mispredictions and also incurs less cache misses than StdQueue. We suppose that, while the number of cache faults is lower for S^3QBH, they have a higher penalty since the optimization we applied transformed a conditional branch – which even a static branch predictor would predict correctly half of the time in the average case – into a data dependency that causes a pipeline stall for every cache miss.

Nevertheless, applications which only use heaps that fit into fast memory can still profit from the speedup that is provided by applying the simple conditional-branch reduction in the *siftDown* loop. By doing just this (plus changing the index type; see subsection 5.4.2) we could achieve an average speedup of 1.2 for the Sequence Heap implementation.

7 Future Work

In this chapter we list potential ways to further improve and extend S^{3}Q.

7.1 Memory Management

As our evaluation has shown, S^{3}Q cannot yet match the I/O efficiency of Sanders' Sequence Heap implementation. One way to reduce the performed I/Os by a constant factor would be to implement buckets not as contiguous pieces of memory, but instead as linked lists of memory chunks of some uniform size in $\Theta(M)$. If we ensure that any bucket has at most one non-full chunk, this would reduce the I/O complexity of joining two buckets to $\Theta(M/B)$ while also improving the average space utilization. Furthermore, the copying of elements involved when resizing dynamically growing arrays can be avoided as well. Finally, this should reduce memory fragmentation and enables us to reduce our usage of relatively slow, general-purpose memory allocation routines provided by the operating system in favor of more specialized and efficient means like free-lists. While this change also somewhat increases the implementation complexity of various operations like sampling, classification and partitioning, we believe that a careful implementation would result in an overall performance gain.

7.2 Optimizations from IPS^{4}o

Due to the conceptual similarity between S^{3}Q and IPS^{4}o[2], many of the methods that are key to the latter's performance should be applicable to S^{3}Q as well. First and foremost, the parallel classification and partitioning schemes of IPS^{4}oare directly applicable to our use-case and should yield a considerable speedup for applications where the priority queue is the performance bottleneck.

By using a contiguous block of memory for the class write buffers during the partitioning of a bucket, we should be able to increase k to $\Theta(M/B)$ without falling victim to conflict misses when using caches with low associativity [13].

As mentioned before, a bucket consisting entirely of elements that share a single key is always sorted. This fact can be used to reduce the effective number of items that have to be processed. IPS^{4}ouses so-called *equality buckets* [3] for that. However, it is unclear if this concept can be efficiently transferred from the problem of sorting, where all elements are known in advance, to the more dynamic problem of maintaining a priority queue.

7.3 Handling of Invalid Splits

As already mentioned in chapter 5, the handling of invalid splits (ones where the resulting buckets violate the size constraints) in our current implementation does not match the theoretical algorithm analyzed by us. Thus, an efficient implementation of the fallback-to-sorting variant we analyzed should be evaluated to see how it compares to our current implementation. Such an implementation should be able to determine the validity of a split without having to distribute all elements to their new buckets first. This can be achieved with the oracle approach also employed by IPS^{4}o[2].

7.4 Extensions

Finally, we outline how S^{3}Q can be extended to support additional operations. We can efficiently *build* a S^{3}Q by performing $k/2$-way splits on the input and recursively on the leftmost resulting bucket until the base bucket size is reached. For an input size of $N \in \Omega(M)$ elements this operation requires expected time $\Theta(N \log k)$ with an expected number of $\Theta(N/B)$ memory accesses which is a factor of $\Theta(\log N)$ less than constructing an empty backend and performing N/B sequential pushes.

We also define a restricted *delete* operation for the case where a specific key is only deleted once and not re-inserted afterwards. To this end, we apply the same principle as Arge's Buffer Tree: when an element with key x shall be deleted, we insert a special marker element into our queue that also has key x. This element is treated like any other element, except by the *deleteMin* logic in the frontend. Here, we ensure that the special elements always compare smaller to their deletion targets. After any deletion from the min-buffer, we check if the next element is a marker element. If so, we apply *deleteMin* to the min-buffer two more times, removing the marker and the element it was supposed to delete. Under the given restrictions, this approach increases existing bounds only by a constant factor and the complexity of *delete* is the same as that of *insert*.

8 Conclusion

Our experiments suggest that Superscalar Sample Queues (S^3Q) are among the fastest comparison based priority queue implementations for cached memory. For input sizes greater than 2^{10} elements, we measured speedups between 1.2 and 2 compared to Sanders' Sequence Heap [14]. The main reason for this advantage is a better utilization of the instruction pipeline that is employed in modern superscalar processors. This is made possible by our careful avoidance of hard-to-predict conditional branches which reduces the average number of branch mispredictions caused by a call to *insert* or *deleteMin* in our experiments to less than 1/5. In fact, we could even speed-up Sanders' Sequence Heap implementation by an average factor of 1.2 just by transforming a conditional branch into a data dependency.

Still, there are quite a few opportunities to further improve upon the present version of S^3Q. Most notably, the current memory management leaves some room for improvement. While our experiments confirm the asymptotically optimal memory efficiency that our analysis predicted, the constant factors could probably be improved by employing more sophisticated data structures and allocation schemes. Furthermore, several techniques from IPS^{4}oshould be easily transferrable to S^3Q and further improve its performance. The most promising among those are parallel element classification and bucket partitioning as well as cache-aligned, contiguous write buffers to allow higher level degrees regardless of cache associativity. Finally, there is some disparity in how the theoretically analyzed algorithm and our implementation handle invalid sample splits. To be able to give definitive performance guarantees, this gap should be closed from either side.

If above items are addressed and our findings can be reproduced over a wider range of real-world load scenarios, we believe that S^3Q has the potential to be a good addition to performance-oriented general-purpose libraries – or maybe even standard libraries.

Bibliography

[1] Lars Arge et al. *An optimal cache-oblivious priority queue and its application to graph algorithms*. Tech. rep. 6. 2007, pp. 1672–1695. DOI: 10.1137/S0097539703428324.

[2] Michael Axtmann et al. "In-Place Parallel Super Scalar Samplesort (IPSSSSo)". In: *25th Annual European Symposium on Algorithms (ESA 2017)*. Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 9:1–9:14. ISBN: 978-3-95977-049-1. DOI: 10.4230/LIPIcs.ESA.2017.9.

[3] Timo Bingmann and Peter Sanders. "Parallel string sample sort". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8125 LNCS. Springer, Berlin, Heidelberg, 2013, pp. 169–180. ISBN: 9783642404498. DOI: 10.1007/978-3-642-40450-4_15.

[4] Gerth Stølting Brodal. *A survey on priority queues*. Tech. rep. 2013, pp. 150–163. DOI: 10.1007/978-3-642-40273-9_11.

[5] Mo Chen et al. *Priority Queues and Dijkstra's Algorithm*. Tech. rep. 2007, pp. 0–24.

[6] Amr Elmasry and Jyrki Katajainen. "Lean programs, branch mispredictions, and sorting". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7288 LNCS. Springer, Berlin, Heidelberg, 2012, pp. 119–130. ISBN: 9783642303463. DOI: 10.1007/978-3-642-30347-0_14.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[8] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2020. URL: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[9] Poul Henning Kamp. "You're doing It wrong". In: *Communications of the ACM* 53.7 (2010), pp. 55–59. ISSN: 00010782. DOI: 10.1145/1785414.1785434.

[10] Jyrki Katajainen. *Seeking for the best priority queue: Lessons learnt*. Tech. rep. 2013. URL: http://cphstl.dk/Report/Dagstuhl-13391/the-best.pdf.

[11] Daniel H. Larkin, Siddhartha Sen, and Robert E. Tarjan. "A back-to-basics empirical study of priority queues". In: *Proceedings of the Workshop on Algorithm Engineering and Experiments*. 2014, pp. 61–72. ISBN: 9781611973198. DOI: 10.1137/1.9781611973198.7. arXiv: 1403.0252.

[12] John von Neumann. "First Draft of a Report on the EDVAC". In: (1945).

[13] Peter Sanders. "Accessing multiple sequences through set associative caches". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1644 LNCS. Springer Verlag, 1999, pp. 655–664. ISBN: 3540662243. DOI: 10.1007/3-540-48523-6_62.

[14] Peter Sanders. "Fast Priority Queues for Cached Memory". In: *ACM Journal of Experimental Algorithmics* 5 (Dec. 2000), p. 7. ISSN: 10846654. DOI: 10.1145/351827.384249.

[15] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3221 (2004), pp. 784–796. ISSN: 16113349. DOI: 10.1007/978-3-540-30140-0_69.

[16] Peter Sanders et al. *Sequential and Parallel Algorithms and Data Structures*. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0.

[17] Jeffrey Scott Vitter. "Algorithms and Data Structures for External Memory". In: *Found. Trends Theor. Comput. Sci.* 2.4 (Jan. 2008), pp. 305–474. ISSN: 1551-305X. DOI: 10.1561/0400000014.

[18] Ingo Wegener. "Bottom-up-heap sort, a new variant of heap sort beating on average quick sort (if n is not very small)". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 452 LNCS (1990), pp. 516–522. ISSN: 16113349. DOI: 10.1007/BFb0029650.

[19] J. W. J. Williams. "Algorithm 232: Heapsort". In: *Communications of the ACM* 7.6 (1964), pp. 347–348.