Analytical Query Processing Using Heterogeneous SIMD Instruction Sets

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Medien-Inf. Annett Ungethüm
geboren am 26. September 1985 in Rochlitz

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr. Viktor Leis
Friedrich Schiller Universität Jena
Fakultät für Mathematik und Informatik
Institut für Informatik
Praktische Informatik
Professur Datenbanken und Informationssysteme
Ernst-Abbe-Platz 2
07743 Jena

Tag der Verteidigung: 05. Oktober 2020

Dresden, im August 2020

ABSTRACT

Numerous applications gather increasing amounts of data, which have to be managed and queried. Different hardware developments help to meet this challenge. The growing capacity of main memory enables database systems to keep all their data in memory. Additionally, the hardware landscape is becoming more diverse. A plethora of homogeneous and heterogeneous co-processors is available, where heterogeneity refers not only to a different computing power, but also to different instruction set architectures. For instance, modern Intel® CPUs offer different instruction sets supporting the Single Instruction Multiple Data (SIMD) paradigm, e.g. SSE, AVX, and AVX512.

Database systems have started to exploit SIMD to increase performance. However, this is still a challenging task, because existing algorithms were mainly developed for scalar processing and because there is a huge variety of different instruction sets, which were never standardized and have no unified interface. This requires to completely rewrite the source code for porting a system to another hardware architecture, even if those architectures are not fundamentally different and designed by the same company. Moreover, operations on large registers, which are the core principle of SIMD processing, behave counter-intuitively in several cases. This is especially true for analytical query processing, where different memory access patterns and data dependencies caused by the compression of data, challenge the limits of the SIMD principle. Finally, there are physical constraints to the use of such instructions affecting the CPU frequency scaling, which is further influenced by the use of multiple cores. This is because the supply power of a CPU is limited, such that not all transistors can be powered at the same time. Hence, there is a complex relationship between performance and power, and therefore also between performance and energy consumption.

This thesis addresses the specific challenges, which are introduced by the application of SIMD in general, and the heterogeneity of SIMD ISAs in particular. Hence, the goal of this thesis is to exploit the potential of heterogeneous SIMD ISAs for increasing the performance as well as the energy-efficiency.

CONTENTS

AKNOWLEDGMENTS

When I started working at the chair as a research assistant, it never crossed my mind, that I would stay longer than a year and even write a thesis. However, the encouraging work atmosphere and the unexpectedly wide area of research topics convinced me to take my chance in research. First, I would like to thank Wolfgang Lehner for giving me this opportunity to write a thesis, and for trusting in my ability to finish this, although I have not attended any of his lectures past my elementary studies. Second, I owe my gratitude to Dirk Habich, who helped me not only with his wealth of knowledge, but also found the time to discuss new ideas, and the occasional deadlock some of these ideas were leading to. I would also like to thank Viktor Leis for agreeing to be the external reviewer of this thesis.

Further, I would like to thank all my colleagues for the discussions in the office and in the coffee kitchen, the countless KuKs, for sharing joy and sorrow, and for just being a great team. I especially thank the core MorphStore team, the colleagues from other chairs, who I had the pleasure to work with, and the current and former coffee kitchen regulars (in alphabetical order): Alex K., Bene, Claudio, Johannes L., Juliana, Kai, Lisa, Lucas, Maik, Nils, Martin, Patrick, Sebastian, Tobi, Tomas, and everybody I forgot to mention. From time to time, there are also a few motivated students, who help to test new ideas and breathe new life into side projects, which would otherwise not have been realized: André, Eric, Felicita, Johannes P., and Lennart - I wish you all the best for your own master- and phd-theses.

Finally, I thank my family and friends, who always supported me. Elisa, Anne, Romy, and Alex M. taught me, that friendships can last, even if you cannot see each other for months or even years. My parents always believed in me, and the week-ends at my father's cozy home office thirty years ago were the time when I learned, that research must be fun. Most importantly, I want to thank my husband Stefan, who encouraged me to take this path, spent countless hours to explain the principles of electrical engineering to me, and had my back when times were stressful.

Annett Ungethüm
Dresden, October 6, 2020

INTRODUCTION

1.1 MOTIVATION

In our increasingly digitalized world, applications and servers have to manage rapidly growing amounts of data. Applications analyzing this data use Online Analytical Processing (OLAP) for this task. They have to process these large amounts of data while maintaining performance and latency constraints. This includes but is not limited to data gathered for and from digital health, digital assistants, e-commerce, smart homes and cities, and industry 4.0. The recent trend for home office and limitations in social life due to the spread of Covid-19 has fueled the use of cloud services[1], online shops[2], telehealth [3], and PCs in general[4] even further. The analysis of this data is retrieved by Online Analytical Processing (OLAP), which typically includes the access and evaluation of only a small selection of attributes.

To satisfy the demand for performance and low latency, hardware development has established a few trends. One of these trends is the growing amount of main memory, which is available at relatively constant prices. This enables applications to keep all of their operational data in main memory, which is faster than accessing data on disk. Another trend is the introduction of Co-Processors. While GPUs have been common for more than 20 years now, other accelerators have only become available and affordable to the general public during the past few years, e.g. dedicated vector processors, FPGAs, and multi-socket systems. This growing heterogeneity also causes a diversity of instruction sets, because each system offers one or more instruction sets tailored to the hardware. A third development is the increasing density of transistors, which enables multiple cores and different specialized instructions on the same CPU. However, not all of these transistors can be powered at the same time. Hence, only a part of the available chip can be used at a time. This phenomenon is called Dark Silicon [EBA+11]. Thus, it is necessary to use the right transistors, i.e. the right instructions in order to achieve the requested performance.

> *"Unfortunately, improving performance of applications has now become much more difficult than in the good old days of frequency scaling. This is also affecting databases and data processing applications in general[..]"*
> - Jens Teubner and Louis Woods [Teu17]

At the same time, the importance of energy-efficiency as an optimization goal is increasing [HSMR09]. This has different reasons. The cost of powering large-scale data centers is just one of them. There are also physical constraints like the already mentioned power limits. Another reason is that energy dissipation is released as heat, which requires cooling, which requires even more energy and limits the ongoing miniaturization of circuits. It is not hard to guess that this cycle cannot be upheld endlessly. Hence, energy consumption is not only an optimization goal, but also a limiting factor. There are numerous

[1] https://www.handelsblatt.com/25813516.html, accessed 23/07/2020

[2] https://de.statista.com/statistik/daten/studie/579708/umfrage/monatliche-umsatzentwicklung-im-versand-und-internet-einzelhandel/, accessed 24/07/2020
https://www.nytimes.com/interactive/2020/05/13/technology/online-shopping-buying-sales-coronavirus.html, accessed 24/07/2020

[3] https://www.gruenderszene.de/health/coronakrise-telemedizin-durchbruch, accessed 24/07/2020
https://www.hessenschau.de/gesellschaft/wie-die-corona-krise-die-telemedizin-voranbringt,digitale-sprechstunde-100.html, accessed 24/07/2020

[4] https://www.cnbc.com/2020/05/04/pc-sales-usage-rise-during-coronavirus-lockdown.html, accessed 23/07/2020

approaches for energy-efficient data processing, which aim to also keep the performance at an acceptable level, like ERIS [Kis17], E²DBMS [TWZX14], an extension of HyPer for heterogeneous ARM® systems [MRS+14], and even methods to optimize DRAM power consumption [AOA15]. These approaches rely on the setting of hardware knobs, e.g. the CPU or DRAM frequency, and the number and affinity of active threads.

A method, which is promising to boost performance and energy-efficiency likewise, is the use of data-parallelism additionally to the parallelism of instructions. While instruction-level parallelism is reached by multi-threading, data-level parallelism follows the Single Instruction (stream) Multiple Data (stream) paradigm, abbreviated as SIMD, where multiple data elements are processed with one instruction. Hardware support for SIMD is provided by different instruction sets, e.g. ARM® NEON, ARM® SVE, Intel® SSE, and Intel® AVX512. These instruction sets offer so-called vector registers, which can store multiple elements, and instructions to process the data in these vector registers. Therefore, the use of SIMD is also referred to as vectorized processing. However, a meaningful use of SIMD is not always trivial, especially when there are dependencies between the elements in the same vector register. A prominent example, where this situation occurs is compression. To keep the memory footprint small and to reduce memory I/O, it is common to compress data. This can be done in different ways, e.g. by saving only the differences between values instead of the actual value. Thus, values cannot be treated as independent objects of a vector. Branching, typically realized as masked operations, is another aspect, which is not supported by all instruction sets. Moreover, technically it can break the SIMD paradigm, e.g. if only one bit of the mask is set, such that only a single value is processed. Some of these difficulties have already been identified in the 70s [Fly72] and the growing size of vector registers today increases their significance. In some cases, it might even be more useful to use scalar processing over vectorized processing, which breaks down to the already mentioned choice of the right instructions, i.e. scalar or vectorized. Additionally, there is no standard for SIMD instruction sets. Therefore, different manufacturers develop different instruction sets and different versions of these instruction sets, which do not necessarily show a consistent naming scheme or function range. For instance, the degree to which intra-register dependencies can be processed is different. This degree ranges from not possible, e.g. with SSE, to the detection of conflicts between all vector elements with AVX512. Wherever an instruction set is not offering the appropriate operations, workarounds are necessary, which typically include a scalar processing part affecting the performance negatively. Moreover, different vector register sizes and instruction sets can be present on the same system. Especially Intel® CPUs are equipped with different SIMD instruction sets. Hence, these CPUs provide a heterogeneous Instruction Set Architecture (ISA). Previous approaches to provide a unified interface for different instruction sets focus mainly on sequential memory access and element-wise arithmetic computation, e.g. Sierra [L+14], VC [KL12a], boost.SIMD [EFGL14], and UME::SIMD [KM17].

"A number of difficulties can be anticipated for the SIMD organization."
- Michael J. Flynn [Fly72]

However, if used appropriately and with a powerful instruction set, substantial performance gains over scalar execution can be achieved in query processing [P+19b]. For this reason, it is worth to analyze and overcome the specific challenges, which are introduced by the application of SIMD in general, and the heterogeneity of SIMD ISAs in particular. The goal of this thesis is to exploit the potential of heterogeneous SIMD ISAs for increasing the performance as well as the energy-efficiency.

1.2 CONTRIBUTIONS

Tweaking the use of different SIMD instruction sets requires a deep understanding of the specific challenges created by large registers and the growing heterogeneity of instruction sets. In this thesis, we provide an insight into these challenges and present according solutions, which are a prerequisite for our optimization approach. The main contributions of this thesis can be summarized as follows:

1. We provide a thorough analysis of the challenges created by the introduction of wide vector registers. The growing size of vector registers creates inherent and algorithmic challenges, which must be overcome to be of use for data intensive workloads. Additionally, the diversity of instruction sets on different hardware platforms complicates the port of existing solutions and a fine-grained selection of vector register sizes.

2. We present a *Template Vector Library*, which abstracts from heterogeneous instruction sets while enforcing explicit vectorization. This enables a hardware-oblivious implementation of portable solutions, which tackle the specific challenges of large vector registers. The range of functions, which our library provides, so-called primitives, was chosen to work for memory intensive workloads with different memory access patterns as opposed to existing solutions, which are mainly made for computational workloads, e.g. for scientific simulations.

3. Our *Template Vector Library* enables a fine-grained choice of the register size and the instruction set without changing the underlying source code. This offers the possibility to explore and to use the optimization potential of combining different instruction sets in the same query. We develop a benchmark-based model, which we use for the optimization of performance as well as for the optimization of energy-efficiency. We call this model *Work-Energy-Profile*.

4. To minimize the number of required benchmarks, we present a method, which combines several primitive *Work-Energy-Profiles* into profiles for more complex use-cases, e.g. for operators.

5. We introduce approaches of applying *Work-Energy-Profiles* in different scenarios, e.g. for continuous workloads or for individual queries.

6. An end-to-end evaluation using the popular Star Schema Benchmark [OOC07] shows the applicability of our optimization approach for analytical queries. To apply our approach, we use MorphStore [DUP+20], an in-memory query execution engine, which implements all its operators and compression algorithms using our template vector library. We also compare our results to MonetDB [IGN+12], a state-of-the-art column-store.

Additionally to our main contributions, we provide an overview of related work for each individual topic and a discussion of selected results. Most of our contributions have already been published in international conferences and large parts of the source code are available online, which we will reference in the appropriate sections of this thesis. Although our approach is software-based, we also give a short insight into the hardware-based approaches for performance and energy-efficiency optimization using instruction set extensions. As we will show, hardware design does not have to be a one-sided process, but in a collaborative environment, software requirements can serve input for the development of hardware, which fits the applications they are made for.

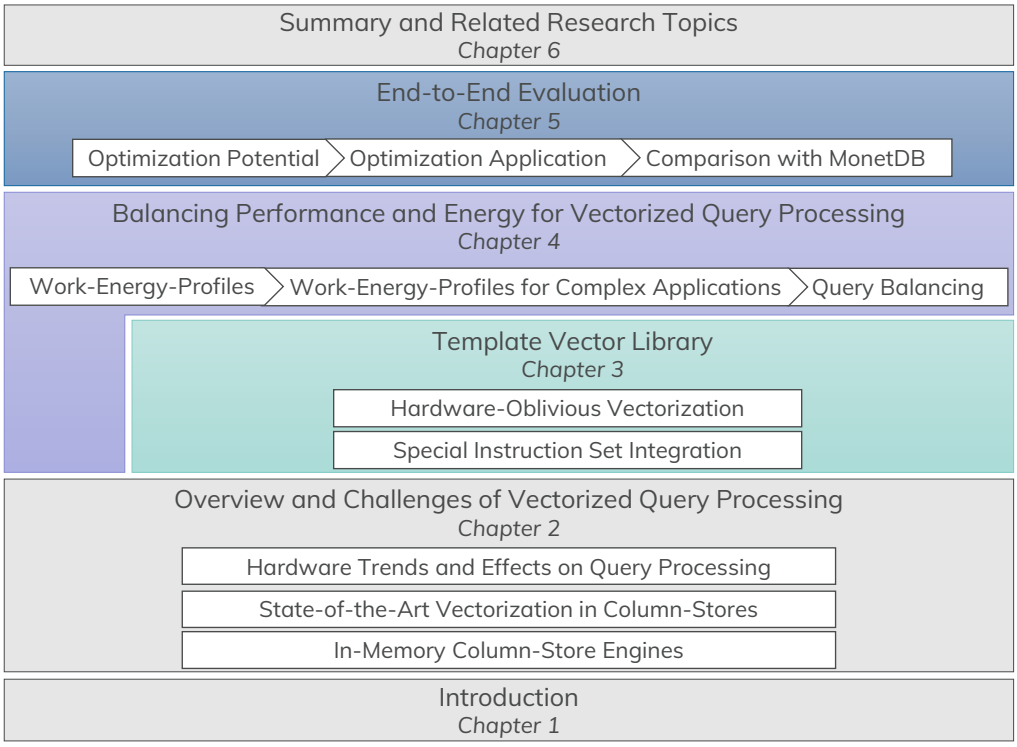


Figure 1.1: Overview of the structure of this thesis

1.3 OUTLINE

Figure 1.1 provides an overview of the topics of this thesis, which roughly reflects our main contributions. In Chapter 2, we recapitulate the state-of-the-art in column-store engines and their application of vectorization. We also discuss the effects of vectorization on query performance and its physical constraints, which come with instruction set extensions. Chapter 3 shows why the integration of vector instruction sets into query execution engines is not trivial. There are reasons following directly from the size of the vector registers, and there are reasons following from the fact that many existing algorithms were designed for scalar processing. Moreover, the variety of instruction sets between different hardware systems requires different implementations, which can vary heavily in their complexity and performance. Therefore, we present a *Template Vector Library*, which enables hardware-oblivious explicit vectorization. In Chapter 4, we present our *Work-Energy-Profiles* as a model used to optimize for performance and for energy-efficiency. By using our library, different instruction sets integrate seamlessly into this approach. We also provide a benchmark concept to create such *Work-Energy-Profiles* and a method to create *Work-Energy-Profiles* for use-cases which were not individually benchmarked. Then, we apply our optimization approach to complex analytical queries in Chapter 5. Finally, we provide a summary in Chapter 6. Additionally, we show how our optimization goals are tackled from the hardware side and which potential future research topics arise from our work.

OVERVIEW AND CHALLENGES OF VECTORIZED QUERY PROCESSING

The state-of-the-art in analytical query processing are in-memory column-store engines [Pla09]. In this chapter, we will give an outline of the characteristics of such systems in Section 2.1 and present two widely adopted optimizations: vectorization and compression in Section 2.2. These optimizations are often used in a combination, i.e. vectorization is used to enhance the compression speed ([D+19, L+15, SGL10]), which improves the overall performance. However, the use of vectorization in an ever-changing hardware landscape presents the query execution engine with some challenges, which we will discuss in section 2.3.

2.1 IN-MEMORY COLUMN-STORE ENGINES

Efficient analytical query processing heavily relies on fast data access. In the memory hierarchy, disk access has a higher latency and lower bandwidth than the volatile memory in the lower layers. For this reason, it makes sense to keep data in other layers of the memory hierarchy, e.g. in main memory. The increasing density of main memory at relatively low prices allows for the development of in-memory database systems, which reduce the use of disk to a minimum or completely eliminate it, e.g. HyPer [KN11] or MonetDB [IGN+12].

Further, analytical queries are characterized by the access of few columns but many rows, e.g. a simple scan reads all rows of a column. Thus, memory access is not only optimized by moving the data into main memory, but also by reorganizing it in a so-called column-store. This means, that data is partitioned vertically, such that columns are stored sequentially, where each element has an ID identifying the tuple it belongs to. To reconstruct a tuple, a projection on the other columns is done with this ID. A project operator reads the IDs of a result set and gathers the values of another column, which share the same ID. For an efficient memory access and evaluation, values and IDs are typically represented by a numerical data type with a fixed size, i.e. as integer data types. In MonetDB such a collection of (ID, value)-tuples is called Binary Association Table (BAT), where the value-part is a memory mapped simple array. A column-store enables linear memory access during scans and aggregations of columns, because only the column, which is evaluated has to be read. There are no other attributes of the tuple, which have to be skipped or read without being used. This increases the performance compared to the random access required for accessing selected columns in horizontally fragmented relations. A further optimization is to omit the ID and just use the position in an array instead. This is done in the read-optimized store of the system C-Store, which shows significant speed-ups compared to row-stores, which use horizontal fragmentation [SAB+18]. Later, the idea was also adopted by MonetDB, such that only a base value per column is stored now and the other IDs can be reconstructed from this base value[1].

The execution engine of such column-stores requires special operators working on columnar data instead of the traditional rows, e.g. [BMK+99, BHC12]. Figure 2.1(a) illustrates this using a simple warehouse example query. This query assumes a relation *items* with at least 2 attributes *product* and *price*. The requested result is the number of stocked items, which cost less than 150 cents. The data is stored column-wise. To evaluate the example query, at first a scan on the *price* column is executed. Each value is read and checked for the condition *less than 150*. The result of this check can be represented in two different ways. One way is a bitmask, where each set bit indicates that the corresponding element satisfies the condition. The second way is to store the position of the matching elements in an intermediate column. Both ways can be used to do a projection on the *quantity* column in the second step. With the bitmask, each bit of the mask is read.

[1] https://www.monetdb.org/blog/monetdb-goes-headless, accessed 08/04/2020

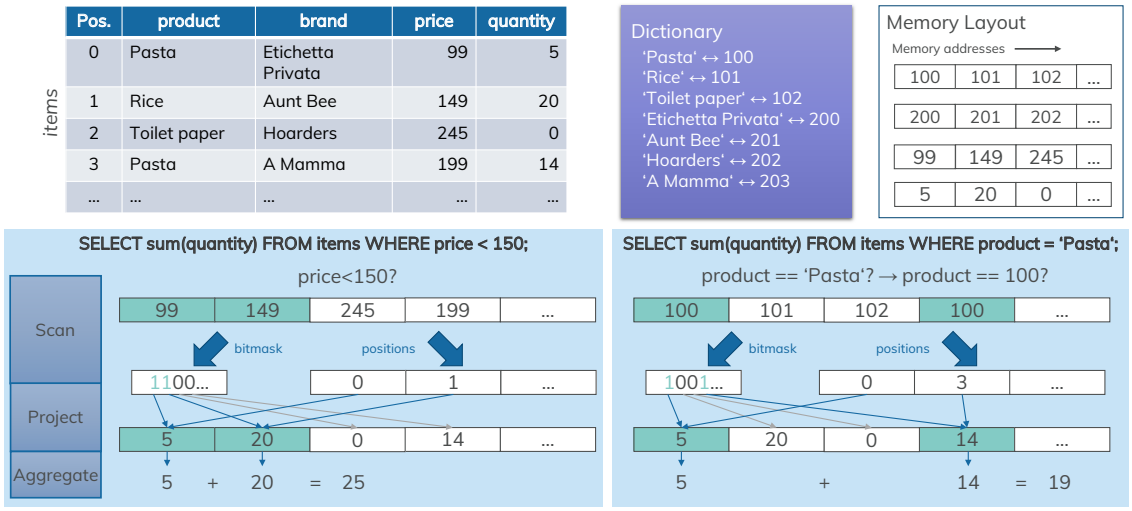


Figure 2.1: A relation (*items*) and examples illustrating the query execution on data in columnar layout. (a) A query reading columns with integer data. (b) Data is dictionary encoded before being processed.

If the current bit is set, the position of this bit is the position of a result element in the *quantity* column. The values at these positions are then written sequentially into an intermediate column. All values of this intermediate column are then summed up by the aggregation operator. In this scenario, the columnar layout enables a maximum of linear memory access. The scan in the first step reads the *price* column sequentially and writes the intermediate result sequentially, i.e. the bitmask or the positions. The projection in the second step sequentially reads the intermediate result. Only the *quantity* column is accessed randomly during the projection to gather its values. The result of the projection is again stored in an intermediate column, which resides sequentially in memory. Finally, the aggregation reads this second intermediate result sequentially. Some systems fuse operators whenever possible, e.g. they do the projection and the aggregation in one step without writing an intermediate result, but the sequential memory access is preserved. Such in-memory column-stores have become the most common approach for analytical query processing because they perform better on modern CPUs than the traditional row-store [Pla09]. Additionally, they are perfectly suited for further optimizations like vectorization or compression, which we will discuss in the following sections.

2.2 STATE-OF-THE-ART VECTORIZATION IN COLUMN-STORES

Vectorization allows to process a number of values with a single instruction and in a single register, the so-called *vector register*. Hence, vectorization is also called SIMD (single instruction multiple data). Recent CPUs are equipped with instruction sets for vectorized processing. A SIMD instruction set offers two extensions to the basic instruction set: (1) Vector registers, which are larger than the traditional scalar register, and (2) Special instructions working on the vector registers, usually wrapped by a handy library for a higher level programming language, e.g. for C/C++. From server-grade Intel CPUs (SSE, AVX, AVX512) to mobile devices (e.g. NEON on Arm CPUs), vectorization is ubiquitously available. Hence, SIMD is also naturally applied in query processing. For more than a decade, using SIMD has been state-of-the-art when it comes to optimizing column-store query engines. In this context, vectorization is a tool for single-query optimization, and the design of the operators is a key aspect to reach this goal.

Column-stores provide their column data as a sequence of values with the same data type, which enables fast sequential access by the operators. By vectorizing operators, data parallelism is added, which can speed up query performance. Moreover, sequential data access can be vectorized trivially from a programmer's perspective, because loading a vector register from sequential data in memory is a basic functionality found in all SIMD instruction sets. The memory access is then still linear or strided, i.e. the memory address, from which the data is read, moves only forward and in well-defined steps. However, for an efficient evaluation in vector registers, all values need to have a fixed size. For this reason, data with a variable length, e.g. strings, are dictionary encoded. This means that each value is encoded with an integer. These integers are then used during query processing instead of the actual values. The mapping between the values and integers is stored in a separate dictionary. For example, the *product* and *brand* columns of the warehouse example from Section 2.1 are not given with IDs but with names, which are strings. Figure 2.1(b) shows a dictionary, which maps each value of the *product* and *brand* columns to an integer. These integer values are stored sequentially as shown in the Figure. A query can now be evaluated on integers instead of strings. This is shown with an example, which queries the amount of stocked pasta. The scan on the *product* column in the first step does only integer comparisons and stores the intermediate result. Although the dictionary can be used to reconstruct the original values, this is not necessary in this query.

Finally, operators and compression do not make a working system. A processing technique is required to actually evaluate queries with vectorized operators on compressed data. In the following, we will explain the mentioned aspects in more detail.

2.2.1 Vectorized Operators

Since in-memory column-store engines assume that all of their data resides in main memory, fast memory access is the key to performance optimization. From this perspective, physical operators can roughly be categorized into two groups: (1) Operators with sequential memory access, e.g. scan, and (2) Operators with random memory access, e.g. lookup.

Linear Memory Access

For column stores, this first group can often be vectorized with a very basic SIMD instruction set, which offers only generic load and store functions. Hence, first approaches for vectorized operators using only linear memory access have already been proposed almost two decades ago. For instance, in 2002, Zhou et al. present SIMD algorithms for scans, aggregations, nested loop joins, and for building selected index structures [ZR02]. These operators require the data to be stored column-wise, such that multiple values of a column can be loaded into a vector register with one instruction and then be evaluated with another instruction. An example operator proposed in [ZR02] is shown in Figure 2.2 on the left side. Since the SSE instruction set with 128 bit registers on 32-bit systems was common when the work was proposed, we decided to show a corresponding example implementation using SSE and 32-bit data types on the right side of the figure. The example sums up all values in a column y, where column x equals 5. For simplicity, this implementation hard-codes the element count of the vector register (S in line 1) and the constant 5 (x_5 in line 2). The columns are given as raw data pointers x_ptr and y_ptr. The first part, i.e. the for-loop, is used in all operators, which are linearly reading data.

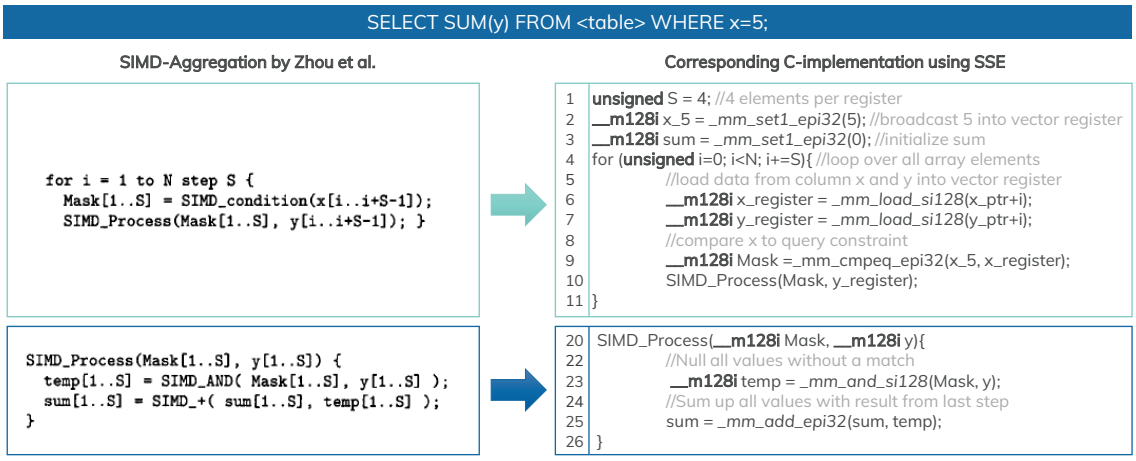


Figure 2.2: A SIMD aggregation as proposed by Zhou et al. [ZR02] and a corresponding implementation for C/C++ using the Intel SSE instruction set. Note that the final aggregation of the four elements in the sum register are not pictured.

In this loop, the function $SIMD\_condition$ checks whether a condition is met by the elements of a vector and returns the result as a bitmask. In our example, this is a check for equality with the number 5. The second part, i.e. the function $SIMD\_Process$, is specific to the aggregation-operator. Note that the authors use a 1-based index, while our implementation uses a 0-based index. Apart from this difference, we exchanged the generic functions and data types with the according SSE intrinsics and types, and added some initializations and explicit vector load operations. All in all, the translation into a real programming language is straight forward. After some initializations in line 1-3, the for-loop starts. In this loop, two vector registers are loaded from linear data in the two columns x and y (line 6-7). The comparison for equality with 5 is done in line 9. The result is another vector $Mask$, which stores either 32 set bits for a match or 32 unset bits if there was no match for each vector element. This result and the register loaded in line 7 are the parameters for the function $SIMD\_Process$ (line 10). Finally, $SIMD\_Process$ (line 20-26) performs the actual aggregation. At first all elements without a match are nulled using $Mask$ and a logical AND (line 23). Then, the values are added to the values of the preceding iteration (line 25). The final aggregation of the elements in the resulting register after the last iteration is done linearly, introduces a minor overhead, and is not shown in the code.

Random Memory Access

The operators from the second group require at least one of the following vector instructions: gather, scatter, masked load, masked store, or compress store. Figure 2.3 illustrates these instructions. A gather loads a vector register with values from various memory addresses. A scatter operation is the store equivalent of the gather. It stores values from a vector register to various, non-linear, memory addresses. A selective load, also called a masked load, uses a bitmask to load only selected values from memory into a register (B). The elements at positions, which are not loaded, are either zeroed or filled from another vector register (A). The masked store works the other way round. The bitmask is not used for loading, but for storing selected elements from a vector register into memory. The existing values at addresses, which are not written, do not change,i.e. they are not

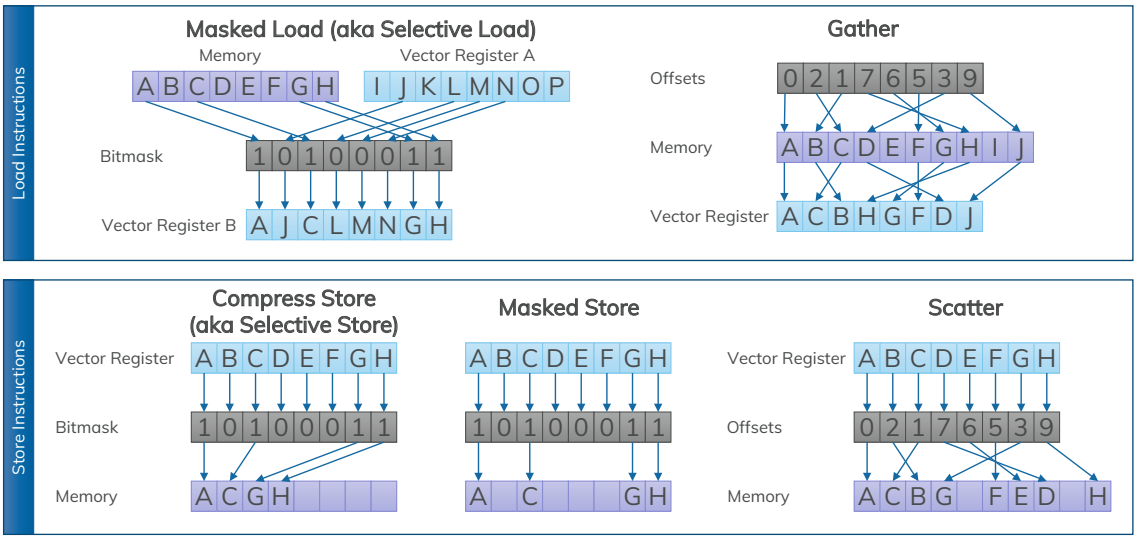


Figure 2.3: Different instructions for random memory access. Often, this also involves selective vector lane access.

zeroed. In contrast to this, a compress store does store all selected values linearly into memory without any gaps for the not selected elements. If the compress store is called in a loop, the new memory address has to be computed after each iteration, i.e. the new address is only known at runtime. For this reason, memory access cannot be scheduled as efficiently as with a common store instruction, where the address offset equals a whole vector width in every iteration. In literature, the compress store is also called selective store. However, it is not the store equivalent for a selective load. For a clear distinction of the instructions, we will call it compress store in the remainder of this thesis. There is no dedicated compress load, because it is just a special case of the gather instruction, but there are combinations of the presented instructions, e.g. masked gather. If these instructions are not supported by the CPU, data has to be moved before it can be loaded into vector registers, which is another expensive operation. Hence, operators requiring random memory access have only been vectorized, when newer instruction sets, e.g. AVX2 on intel CPUs, were widely available.

Polychroniou et al. present a collection of vectorized operators, which use these instructions [P+15]. Their work includes not only the building of index structures, but also using them, e.g. the build and probe phase of Cuckoo hashing. This can then be used to realize more sophisticated physical operators, e.g. hash joins. However, this is only beneficial for the performance, because the required random memory access instructions are natively supported by the CPU used by the authors.

Despite this hardware support for random memory access, the best performance is still shown when linear memory access is used. In some cases, the advantages of linear access outweighs the cost of moving data to enable this access, especially if the data is read multiple times. Schlegel et al. exemplarily show the impact of linear memory access compared to random memory access for a vectorized k-ary search [SGL09]. In one scenario, they calculate the addresses of the separators in each step. Hence, the separators are not loaded from sequential addresses. In the other scenario, the separators are written linearly to a dedicated memory region before executing the search. Compared to a linear binary search, the k-ary search with random memory access reached a speed-up of up to 1.8, while with linear memory access, it reached a speed-up of up to 2.5 on an Intel i7 CPU.

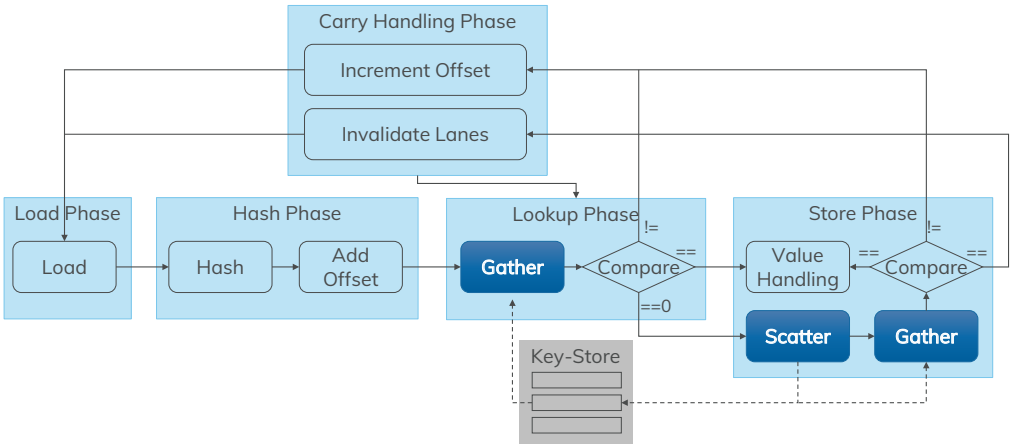


Figure 2.4: The vectorized linear probing as proposed in [P+15] introduces the highlighted random memory access instructions.

Kim et al. [KKL+09] consider a sort-merge join to exploit data-parallelism and to avoid random memory access as far as possible. They report a comparable performance with a hash join when using 256-bit registers and a higher performance when using 512-bit registers. Unlike a hash join, a sort-merge join using a bitonic sort avoids gather and scatter instructions by using linear load and store instructions as well as shuffle instructions.

The case of random memory access in hash joins, specifically during linear probing, has been investigated in [P+19a]. Generally, linear probing is a way to find an empty slot in a hash table to insert a key, when a hash collision occurs, i.e. when a hash function maps two different keys to the same bucket. In these cases, the next best free slot is used to insert a new bucket with the second key. Vectorizing this approach introduces the challenge to find such collisions between the keys or hash values, which are loaded into the same register. Figure 2.4 shows the control flow of the vectorized linear probing as proposed in [P+15]. In the load phase, keys are loaded into a register. In the hash phase, a bucket is computed for each key and an offset is added, which is initially zero. The lookup phase loads the keys currently stored in the computed buckets and compares them to the keys loaded in the register. If a bucket is empty ($== 0$), the corresponding keys in the register are stored with a scatter operation in the store phase. However, to avoid storing potential equal hash values, which might exist in the same register, to the same bucket, another gather operation is used to load the active keys again and compare them. If the comparison is successful, the lanes (bits holding a vector element) of these keys in the register are invalidated in the *carry handling phase*. For all keys, where the comparison is not successful, the lanes are not invalidated, and they are not stored. If the keys are equal, there is nothing to be stored and the vector lane containing the key is also invalidated. If the bucket is not empty but the keys are not equal, a new bucket has to be found. In this case, the lane is not invalidated and the key cannot be stored in the current iteration. All invalidated lanes can be reloaded with new keys in the next iteration, while all other lanes stay in the vector register and an offset is added for the next hash phase. Thus, there are several random memory access instructions in vectorized linear probing. Specifically, the lookup phase contains a gather, and the store phase even contains a scatter and a gather.

2.2.2 Lightweight Integer Compression

The use of vectorization in operators introduces data parallelism, which increases the performance. This performance can even further be increased, when more elements fit into one vector register. This cannot only be achieved by wider vector registers, but also by smaller data elements. For instance, if only 16 or 8 bit are used per value instead of 32, a 128 bit register can fit not only 4, but 8 resp. 16 values. Hence, the data parallelism is increased because more values can be loaded and evaluated at once. Moreover, bandwidth is saved if the data required for query evaluation is smaller. To reach this goal, compression is used. Consequently, compression itself is also vectorized to enable the compression of multiple values at once [SGL10, L$^+$15]. Since a user is usually interested in correct results, lossless compression is used in database systems, e.g. the aggregated payroll costs of a company should be calculated exactly instead of approximately, not only for the tax declaration. A challenge for compression in in-memory database systems is that the bandwidth savings must outweigh the costs of compression and decompression. This is a major difference to the compression used for archives on disks or on tape, where the compression rate is the most important optimization goal. That is why in-memory database systems typically use lightweight compression as opposed to heavyweight compression. In lightweight compression, the compression and decompression performance is higher than in heavyweight compression, potentially on the cost of the compression rate.

Abstraction Levels

As described in [D$^+$19], there are different lightweight compression techniques, algorithms, and implementations, which address different levels of abstraction.

Techniques describe the basic idea for a compression. Techniques can work on two different levels: The physical level or the logical level. On the physical level, the number of used bits per value is reduced, while the logical level reduces the number of values. The physical level compression technique works on either bit-, byte-, or word-granularity and reduces the number of existing but unused bits. This is why it is called Null Suppression (NS). On the logical level, there are different techniques. The already mentioned Dictionary Encoding (DICT) encodes values as integers. While not necessarily reducing the required space, it can be used to encode strings or random data types as integers, which can then be further compressed with a different technique. Frame-of-Reference (FOR), Delta Coding (DELTA), and Run-Length-Encoding (RLE) work on integers. FOR encodes values as the difference to a given reference value, while DELTA stores the difference to a predecessor. FOR or DELTA can cause a later Null Suppression to reach a higher compression ratio. RLE detects sequences of the same value and stores this value only once, together with the length of the sequence.

Algorithms are concrete descriptions of the binary data layout of one or more techniques. Thus, an algorithm explains how to realize a technique or a cascade of techniques. For example, there are NS-algorithms, which try to store as many values as possible within one processor word, while others try to store each value with a minimum number of bits or bytes [ZZL$^+$15]. These three groups are referred to as bit-, byte-, or word-aligned. A fourth group of NS-algorithms clusters sequences of integers with the same bitwidth into frames. These frame-aligned codes can be regarded as a special case of bit-aligned codes. An example for an algorithm using a cascade of techniques is PFOR,

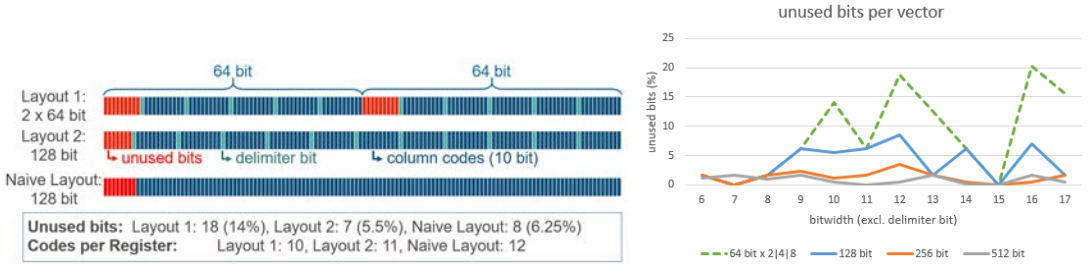


Figure 2.5: Different layouts for storing codes into vector registers and the number of unused bits depending on the register size and the code size

which combines FOR with a bit-aligned NS algorithm. PDICT works similarly but with Dictionary Encoding instead of FOR. PFOR-DELTA additionally does DELTA Coding before applying PFOR [ZHNB06]. As already mentioned, if data is not already present in an integer format, DICT must even be applied before being able to use any other algorithm, which also results in a cascade of techniques. While a compression technique only describes an abstract idea, compression algorithms describe concrete data layouts, which can be optimized for vectorized processing. For instance, Bitpacking (BP) is a bit-aligned NS compression algorithm, which exists in different variants, e.g. SIMD-BP128 is optimized for 128-bit wide vector registers. SIMD-FastPFOR is a vector-optimized variant of an advanced PFOR algorithm [L+15]. There are also vectorized algorithms for DELTA, FOR, and RLE [D+19].

Implementations are the hardware-specific code of algorithms. This code can differ depending on the instruction set offered by the CPU, especially when vectorization is used.

This shows, that there is a huge variety in lightweight compression. There are various algorithms for each compression technique and each algorithm can be realized by different hardware-specific implementations. Moreover, there is no algorithm, which shows the best performance for every use-case [D+19]. Nevertheless, specialized operators, which can directly work on compressed data, can increase the query performance, even if the chosen algorithm is not the most optimal one, because a higher degree of data parallelism is possible.

Specialized Operators

Especially for data, which has been null suppressed, there is a huge variety of operators, mostly scans, e.g. [WPB+09, LP13, FLKX15]. These operators can be vectorized to gain more performance. They either decompress data in-register or not at all.

An example for a vectorized scan, which does an in-register decompression, is SIMD-Scan by Willhalm et al. [WPB+09]. SIMD-Scan directly loads a sequence of bit-aligned codes into a vector register (R_1). A subset of these codes is then copied into a second register (R_2), which is used for the next steps. Then, the codes in R_2 are shifted within the register, such that a code starts at the least significant bit of each vector lane, i.e. at the beginning of each vector element. This allows predicate handling using instructions, which evaluate vector registers on element granularity. The result can be transformed

into a bit vector and written to memory. After writing back the result, the next subset of codes is copied from R_1 into R_2 and the steps are repeated until there are no more elements left in R_1, which have not been evaluated.

BitWeaving [LP13] works without decompression and was originally designed for scalar processor words, but a variant of BitWeaving, the so-called Vertical BitWeaving, can be vectorized in a straight-forward way. Like SIMD-Scan, BitWeaving works on null suppressed bit-aligned codes. The vertical variant expects the codes to be stored in a vertical layout, i.e. a code is spread over multiple processor words. For example, assuming 64-bit processor words, the first bit of the first 64 codes are stored sequentially in one processor word. The second bit of the first 64 words is stored in the second processor word and so on. Hence, there are as many processor words needed as there are bits per code. The predicate is stored in the same way, i.e. a processor word is always filled with the corresponding bit of the predicate. Each of the processor words containing the codes is then bitwise compared to the corresponding processor word containing the predicate. The results after each step are combined bit-wise using a logical AND. In the end, the result is a bitmask indicating which of the 64 elements matched the predicate. This way, the codes do not even have to be decompressed in register. Furthermore, the larger the processor word, the more codes can be scanned at once. For this reason, the vectorization of Vertical BitWeaving is straight forward and shows a significant performance gain over the scalar implementation [LUH+18b]. There is also a vectorized variant of horizontal BitWeaving, where codes are aligned to be loaded sequentially into a vector register with a separating delimiter bit. This delimiter bit contains the result after the predicate evaluation. However, this does not only introduce additional computation overhead but also wastes some space in the vector register depending on the code size. Figure 2.5 exemplarily shows two different layouts for storing codes horizontally into a vector register. It illustrates how to store 10-bit codes into a 128-bit register. Layout 1 allows working within word boundaries. This is important because horizontal BitWeaving requires an arithmetic addition, which does not exist for whole vector registers. Layout 2 uses the register width more efficiently. For comparison, a naive layout without any delimiter bits is also shown. Next to the layouts, the number of unused bits is shown for different vector sizes.

2.2.3 Processing Techniques

Operators and an option for compression alone do not make a query engine. To run vectorized queries, a processing technique is required. Generally, there are three different ways to process a query: (1) *tuple-at-a-time*, (2) *operator-at-a-time*, and (3) *vector-* or *block-at-a-time*. *Tuple-at-a-time* processing is also known as the Volcano iterator model [Gra94]. In this technique, a *next()*-function is implemented by each operator, which returns the next tuple. A query plan is realized as a tree and each node of this tree calls the *next()*-function of its children until no tuples are returned anymore. This allows to fuse several operators into one loop, which computes individual tuples, effectively eliminating the need for materialization of the intermediate results. As much as the fusion of operators decreases unnecessary memory access, as much optimization potential does it take away in other aspects. For instance, capabilities of modern CPUs, e.g. useful prefetching, are heavily impeded by operator fusion as shown in [M+17]. For this reason, the authors argue for a relaxed operator fusion, where execution pipelines are split in more places than only pipeline-breakers. They could show that this has a beneficial effect on query performance, even though there is an additional cost for writing and reading the materialized intermediates.

The opposite of *tuple-at-a-time* is *operator-at-a-time*. With the *operator-at-a-time* technique, no operators are fused. Instead, each operator processes a whole column producing an

intermediate result, which is the input for the next operator. This eliminates the issues mentioned in [M+17] most widely on the cost of additional main memory access.

A system using this technique is MonetDB [IGN+12], which adopts the column-store concept of C-Store. However, compression of the intermediates reduces memory access and therefore helps to compensate for this disadvantage. This allows for the combination of the advantages of both worlds. Moreover, when using SIMD, values from more than one tuple are processed. For this reason, *tuple-at-a-time* processing is not applicable when using SIMD, while *operator-at-a-time* is.

Vertica is another column-store system, which is a direct successor of C-Store [LFV+12]. One of the improvements over the original C-Store is the heavy and varied use of light-weight compression. Delta compression and Run Length Encoding can be used to minimize the memory footprint. Additionally, Vertica offers a variant of Dictionary Encoding. This is a necessary condition for all engines to make use of SIMD, because the overwhelming majority of SIMD instructions process only numeric values. For being able to process string values, they have to be encoded as numeric values first, which is done by Dictionary Encoding.

Apart from compression, an optimization of *operator-at-a-time* is to process only parts of a column at once, so-called *blocks* or *vectors*, where *vector* denotes to any number of values between the actual vector register size and the whole column. This is called *vector-* or *block-at-a-time*. The idea is to write smaller intermediate results, such that they fit into the cache. If these intermediates even fit into the registers, operators can still be fused.

A system using the *vector-at-a-time* model is MonetDB/X100, which uses MonetDB as a base and is able to run vectorized queries on a Cell Engine [HNZB07]. A few adjustments were necessary to reach this. First, the operators were rewritten. Some have been adjusted to be better suited for auto-vectorizers, e.g. a bitmask is used instead of an offset vector to scan previously selected elements. Other operators, e.g. the grouped aggregation, are explicitly vectorized using the intrinsics provided for the cell architecture. Second, the *next()*-interface has been changed to return a collection of vectors instead of individual tuples. Finally, the algebra interpreter still runs on a traditional CPU core, called *PPE* (Power Processor Element) on the Cell engine. This *PPE* triggers the actual execution on the *SPEs* (Synergistic Processing Element), which contain vector registers and the means to process them. Unfortunately, this solution is not applicable anymore, because the Cell Engine is outdated and not produced anymore. However, the idea of processing columnar data in vectors was kept and turned into a commercial product called Actian Vector[2], formerly Vectorwise[ZB12].

There are also approaches targeting more recent hardware, e.g. VIP, which uses the most recent SIMD instruction set by Intel, AVX512 [P+19b]. VIP does not build on an existing system, but implements a bottom-up approach. Explicitly vectorized and precompiled suboperators are used to build full operators, where suboperators, e.g. hash probing, can be used in multiple operators. Query execution works by suboperators processing blocks of tuples from one column and keeping the intermediates in cache wherever possible. A compression algorithm is also supported and realized by combining suboperators. VIP supports dictionary coding with subsequent bitpacking. The use of a recent instruction set and the according large registers (512 bit) combined with the avoidance of compilation during runtime, leads to competitive performance.

In this work, we focus on the opportunities of vectorization, not on those of optimized processing techniques. Therefore, we use the plain *operator-at-a-time* technique without

[2] https://homepages.cwi.nl/~boncz/, accessed: 20/07/2020

fundamental changes such as *block-at-a-time* processing. Nevertheless, all the concepts we will present are also suited for an application in a *block-at-a-time* processing environment.

All mentioned processing techniques can profit from minimizing the overhead of reading and writing intermediates. There are two methods to reach this goal: (1) Avoid intermediates completely, i.e. fuse the operators of a query, and (2) Compress the intermediate results.

As already explained, the first option can easily be applied in *tuple-at-a-time* processing, although there are the mentioned drawbacks to this approach. However, HyPer processes vectors instead of tuples, but also applies operator fusion wherever possible [KN11]. Intermediate results are only materialized at pipeline-breakers, which are operators requiring more tuples than there are in registers. HyPer translates queries at runtime using the LLVM framework. Vectorization is thereby introduced by modeling LLVM vector types [Neu11].

MorphStore is a system, which compresses the intermediate results while heavily applying the SIMD paradigm [DUP+20]. In MorphStore, all intermediate results are materialized. This comes with a few advantages, e.g. intermediate results can be used multiple times, and extensively nested loops, which cannot be unrolled by the compiler, are avoided. To minimize the overhead of writing and reading these intermediates, they are compressed using lightweight integer compression, e.g. static bitpacking or delta. For this purpose, a so-called *morphing-wrapper* decompresses the content of a vector register before it is passed to an operator, and re-compresses the result before it is written to memory. This way, specialized operators do not require an implementation for each compression format. Instead, the core of the operator stays the same. The authors call this method *on-the-fly de-/re-compression*.

2.2.4 Summary

Vectorization, is widely used in different aspects of analytical query execution. Data parallelism is increased by using SIMD registers and instructions in operators and by compressing data. Vectorization is also used to speed up compression itself. We presented a selection of vectorized operators and compression algorithms. All mentioned approaches use explicit vectorization, i.e. they do not solely rely on auto-vectorization. This provides the highest degree of control and, therefore, the best control over any vectorization related optimization knobs. For example, steps like the reorganization of data to enable linear memory access with vector instructions are identified manually. Therefore, it will not happen implicitly. Additionally, auto-vectorization by the compiler is error prone. Some, but by far not all, of the reasons why the compiler might fail to vectorize a piece of code are insufficient optimization of complex loops [M+17], the use of function calls or global pointers, or an unexpected data alignment [ZR02]. This is the reason, why we will address explicit vectorization in this work.

Finally, a processing technique is required, which glues operators and compression together to evaluate a query according to an execution plan. We decided for the operator-at-a-time technique to make the best use of modern CPU features.

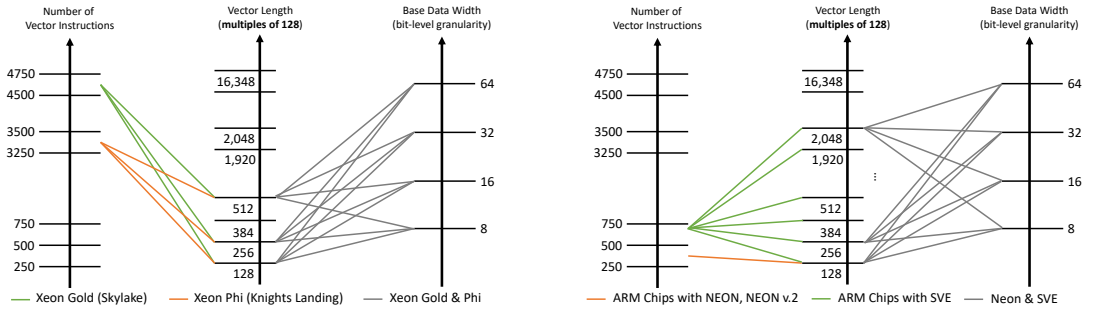
2.3 HARDWARE TRENDS AND EFFECTS ON QUERY PROCESSING

As shown in the last section, there is a huge variety of approaches to vectorize operators, compression, operators for compressed data, and even whole execution engines. However, all of them address a specific target architecture limiting their applicability, because hardware architectures are diverse and change over time, e.g. the cell engine MonetDB/X100 [HNZB07] is optimized for, is not produced anymore. The SSE and AVX instruction sets, which many originally vectorized operators and compression algorithms were developed for, e.g. [WPB+09, ZHNB06, ZR02, SGL10], have been extended by other instruction sets, which offer wider registers or more instructions, i.e. AVX2 and AVX512. Moreover, different instruction sets are available on different CPUs. The SIMD instruction landscape is diverse. Figure 2.6 shows the availability of SIMD instruction sets and vector registers on different Intel and ARM CPUs. The left side of each graph shows the number of available instructions, the middle shows the size of the available vector registers, and the right side shows the possible sizes of the vector elements. All of these general-purpose CPUs are able to work on the granularities of common integral data types, i.e. 8, 16, 32, and 64 bit. However, the number of available instructions and the vector register sizes differ, even between CPUs of the same manufacturer. For instance, the Xeon Gold and the Xeon Phi are both made by Intel® and support the SSE-, AVX-, and AVX512- instruction sets, but they support different subsets of AVX512. Thus, even though both CPUs can work on 128-bit, 256-bit, and 512-bit registers, the exact supported instructions are partially different. AMD® CPUs partially support the same SIMD instruction sets as Intel® CPUs, i.e. SSE and AVX/AVX2, but also implement their own instruction set extensions, e.g. SSE4a. Recent ARM® CPUs provide the NEON and NEON2 instruction set, which only work on 128-bit registers. SVE, the newest instruction set for ARM® cores, which can be tested in an emulator until supporting hardware is available [3], provides scalable vectors between 128 and 2048 bit [SBB+17].

Generally, the trend in SIMD hardware shows two directions. First, there are more instructions providing specialized functionality, e.g. in-register conflict detection with Intel AVX512. Second, the vector registers become larger, e.g. Intel doubles the size with each new major instruction set while ARM introduces scalable vectors up to 2048 bit with their most recent SIMD instruction set SVE. Unfortunately, there is no trend for a standardization of the instruction sets across architectures. As shown with the different supported AVX512 subsets on the Intel CPUs, the opposite is the case. There is no architecture independent standard of what a SIMD instruction set has to offer and how it can be used.

Implementing the same algorithm for different architectures requires significantly different code. Even the port to higher vector sizes on the same architecture includes a certain amount of manual refactoring due to the partially inconsistent naming schemes. For this reason, the implementations of the presented approaches in Section 2.2 cannot be used on architectures they are not designed for, because they use one concrete instruction set and vector size. Moreover, any port to a more recent instruction set, must be done by hand, e.g. an implementation using SSE does run on recent Intel CPUs, but do not use the wider 512-bit registers or any newer instructions, which might increase the performance. Hence, porting any solution to another architecture requires a domain expert of the available instruction set and of the application. However, all previously presented operators and compression implementations use the instruction set and register size, which was the most recent by the time the solutions were developed. This rises two questions: (1) When and how is it beneficial to port a solution to larger vector registers and new instruction sets? (2) How can the manual part of porting a solution be minimized without

[3] Emulator available at https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator, accessed: 11/04/2020



(a) Intel (b) ARM

Figure 2.6: Diversity in the SIMD hardware landscape.

eliminating the explicit vectorization as used in the solutions presented in Section 2.2? We address these questions in Section 2.3.1

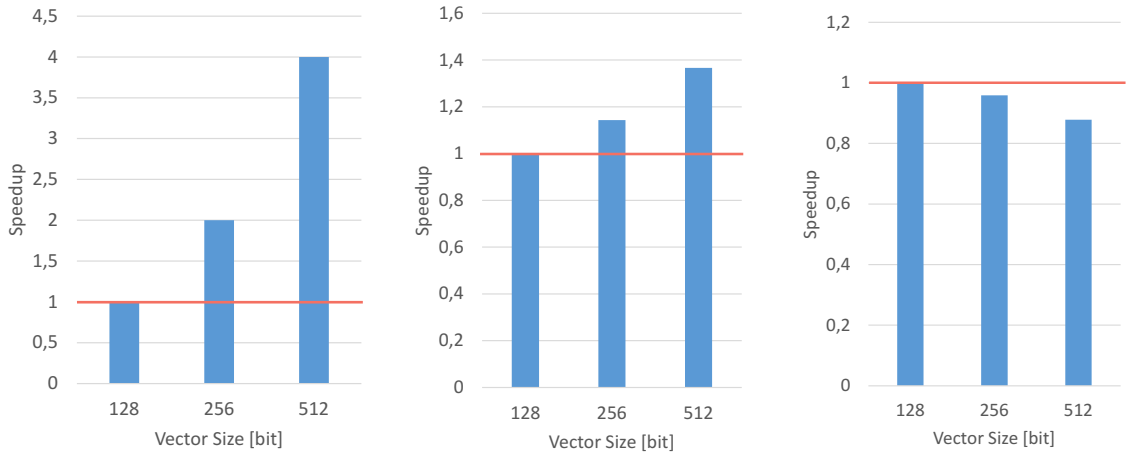
Finally, the chosen instruction set and vector size influences the CPU frequency [Int20] potentially leading to more internal changes, e.g. the memory bus frequency. This is because the circuits implementing SIMD functionality, are more complex than those implementing scalar functionality, but the number of transistors, which can be powered at the same time, is limited [EBA+11]. This leads to a different energy consumption depending on the used instruction set. Hence, it makes sense to not only optimize a vectorized solution for performance, but also for energy consumption. Section 2.3.2 describes the relationship between energy consumption and a chosen instruction set in more detail.

2.3.1 Diversity of Instruction Sets and Vector Sizes

When using wider registers, the increased parallelism should lead to a higher performance. That expectation is depicted in Figure 2.7(a). To check, if this holds true in reality, we run the Star Schema Benchmark (SSB), which is based on the data warehousing benchmark TPC-H [OOC07]. It consists of 13 analytical queries on denormalized tables organized in a star schema. This means that there is one fact table, called *Lineorder*, which references several dimension table, e.g. *Customer* and *Supplier*.

We run the SSB with different instruction sets on an Intel Xeon, which supports the instruction sets SSE, AVX2, and AVX512 with their native vector widths of 128, 256, and 512 bit. All values were dictionary encoded, so all queries process integer values. All data is stored column-wise and reside completely in main memory. The operators for all vector sizes follow the same algorithm, regardless of the used instruction set, and the query execution plan is the same as well. Hence, we used a basic in-memory column-store query engine. From this benchmark, we extracted the results of two representative queries as shown in Figure 2.7(b) and (c). The graph shows the speed-up for different vector widths compared with 128-bit registers. Query 1.1 scales almost as expected. While the speed-up from using a larger register is not as high as expected, there still is a performance gain compared to the use of a smaller register. The limited scalability could be explained by the limited memory bandwidth. In contrast, query 4.1 shows the reverse behavior, which cannot be explained trivially. The widest register size offering the worst performance seems to be in conflict with the increased data parallelism.

As a consequence, the choice of the SIMD instruction set and the according vector size turns out to be an important optimization knob. This optimization potential becomes even larger when the vector size is not chosen on the query level, but on the operator



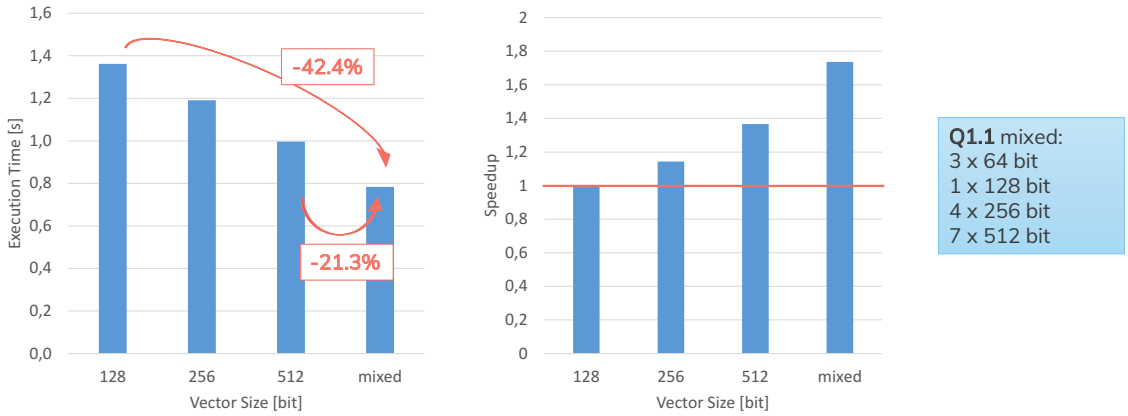
(a) Expectation (b) SSB Q1.1 (c) SSB Q4.1

Figure 2.7: Expectation versus reality. The queries of the Star-Schema-Benchmark scale differently and not as expected. A wider vector size does not guarantee faster query execution.

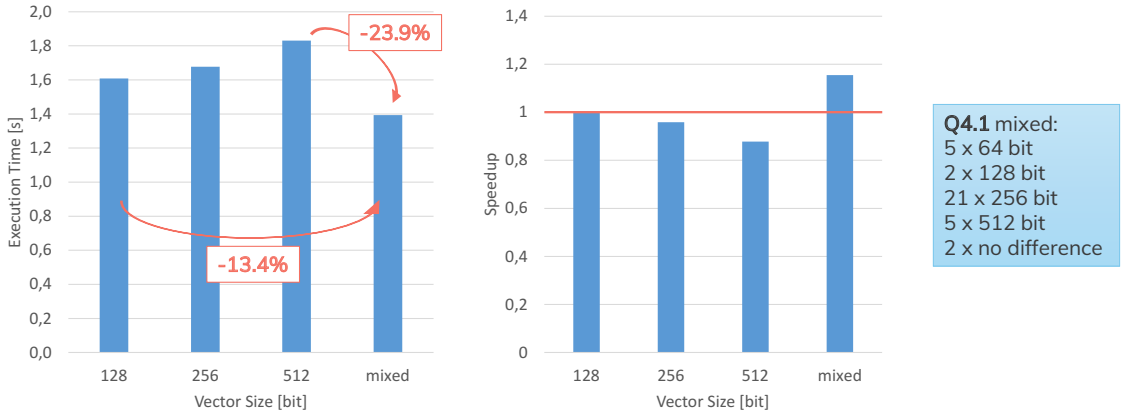
level. Figure 2.8 shows the results of the SSB queries 1.1 and 4.1 again with an added bar for the theoretically reachable optimum. This optimum represents a mixed choice of vector sizes. It is the sum of the execution times of each operator, where the vector size with the shortest execution time is chosen per operator. We also considered scalar processing for this mixed scenario. The boxes next to the graphs indicate how many operators use each available vector width. They show, that both queries require all available vector sizes to reach the best performance. An interesting insight is, that the vector size, which is optimal for most operators is not necessarily the best for the whole query. In this case, most operators in query 4.1 profit from using 256-bit registers. But the overall query has the shortest execution time with 128-bit registers. In this specific query, the execution is dominated by a semi join taking at least six times longer than any other operator. This semi join has the longest execution time when using registers, which are 256 bit or larger, which is sufficient to render the benefits of the other operators useless. This seemingly unpredictable behavior exposes a differentiated potential for optimization.

Absolute optimization potential is the difference between the worst case and the mixed optimum. For query 1.1 (Figure 2.8a), the worst case scenario is the use of 128-bit registers. The optimization potential, i.e. the difference between 128-bit registers and the mixed processing, is 42.4%. For query 4.1 (Figure 2.8b), the worst case scenario is 512-bit registers. The according absolute optimization potential is 23.9%.

Relative optimization potential is the difference between an already existing optimization and the mixed optimum. An already existing trivial optimization would be the use of the largest available registers, which is 512 bit in our case. For this trivial approach, the relative and absolute optimization potential for query 4.1 are the same. For query 1.1, 512 bit is the best case scenario for a register choice on query granularity. The relative optimization potential is 21.3%, almost half of the absolute optimization potential, but still significant. Assuming that there is already a way to find the best performing vector size on a per query basis, the relative optimization potential for query 1.1 would not change, but for query 4.1 it would decrease to 13.4%. However, to the best of our knowledge, such a reliable optimization does not exist, yet.



(a) Query 1.1



(b) Query 4.1

Figure 2.8: The optimization potential of choosing the right vector size. The mixed bar shows the theoretically reachable execution time if every operator was executed with the optimal vector size.

The example shows two main challenges for vectorized column-store engines: First, there seem to be new bottlenecks introduced by larger registers. In Section 3.1, we will have a deep dive into the reasons for this behavior and present a concrete solution for RLE using specialized instructions of the AVX512 instruction set, which go beyond basic logic and arithmetic operations. Second, the right choice of the vector register size and the instruction set is beneficial for the performance of query execution. To do this, there must be implementations to choose from. The implementation of another physical operator for each combination of register size (r), instruction set (s), data type (t), and operator algorithm (o), is a possible solution. However, this solution requires $r \cdot s \cdot t \cdot o$ implementations for each operator, and ideally, they all provide the same interface. Moreover, the developer solving this task has to be a domain expert not only in database development, but also in all target architectures, i.e. he has to know all the instruction sets. The optimal solution for the database developer, as well as for a potential optimizer, is that there is only one implementation of each physical operator, which is independent of the register size, data type, or instruction set. The mapping to these quantities is not done during development but during compile time or runtime. This concept provides a unified interface for the optimizer and a separation of concerns for the developer. Concretely, no multi-domain expert is necessary anymore because the database developer does not implement for a number of specific architectures, while the back-end developer needs no knowledge about the application domain. This concept can be realized by a library, which provides

81xx and 61xx processors

SKU	Cores	LLC (MB)	TDP (W)	Base non-AVX Core Frequency (GHz)	# of active cores / maximum core frequency in turbo mode (GHz)																								
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
8176	28	38.50	165	2.1	3.8	3.8	3.6	3.6	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	2.9	2.9	2.9	2.9	2.8	2.8	2.8	2.8
8170	26	35.75	165	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.3	3.3	3.3	3.3	3.0	3.0	3.0	3.0	2.8	2.8	2.8	2.8	2.8	2.8		
8164	26	35.75	150	2.0	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.7	2.7	2.7	2.7	2.7	2.7		
8160	24	33.00	150	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	3.0	3.0	3.0	3.0	2.8	2.8	2.8	2.8				
6152	22	30.25	140	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	2.9	2.9	2.9	2.9	2.8	2.8						
6138	20	27.50	125	2.0	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.2	3.2	3.2	3.2	2.9	2.9	2.9	2.9	2.7	2.7	2.7	2.7								
6140	18	24.75	140	2.3	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	3.0	3.0										
8153	16	22.00	125	2.0	2.8	2.8	2.6	2.6	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.3	2.3	2.3	2.3												
6130	16	22.00	125	2.1	3.7	3.7	3.5	3.5	3.4	3.4	3.4	3.4	3.1	3.1	3.1	3.1	2.8	2.8	2.8	2.8												

81xx and 61xx processors.

SKU	Cores	LLC (MB)	TDP (W)	Base AVX-512 Core Frequency (GHz)	# of active cores / maximum core frequency in turbo mode (GHz)																								
					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
8176	28	38.50	165	1.3	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0	1.9	1.9	1.9	1.9
8170	26	35.75	165	1.3	3.5	3.5	3.3	3.3	2.9	2.9	2.9	2.9	2.5	2.5	2.5	2.5	2.2	2.2	2.2	2.2	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9	1.9	1.9		
8164	26	35.75	150	1.2	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9	1.8	1.8	1.8	1.8	1.8	1.8		
8160	24	33.00	150	1.4	3.5	3.5	3.3	3.3	3.0	3.0	3.0	3.0	2.6	2.6	2.6	2.6	2.3	2.3	2.3	2.3	2.1	2.1	2.1	2.1	2.0	2.0	2.0	2.0				
6152	22	30.25	140	1.4	3.5	3.5	3.3	3.3	2.9	2.9	2.9	2.9	2.5	2.5	2.5	2.5	2.2	2.2	2.2	2.2	2.0	2.0	2.0	2.0	2.0	2.0						
6138	20	27.50	125	1.3	3.5	3.5	3.3	3.3	2.7	2.7	2.7	2.7	2.3	2.3	2.3	2.3	2.0	2.0	2.0	2.0	1.9	1.9	1.9	1.9								
6140	18	24.75	140	1.5	3.5	3.5	3.3	3.3	2.8	2.8	2.8	2.8	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	2.1	2.1										
8153	16	22.00	125	1.2	2.6	2.6	2.4	2.4	2.0	2.0	2.0	2.0	1.7	1.7	1.7	1.7	1.6	1.6	1.6	1.6												
6130	16	22.00	125	1.3	3.5	3.5	3.1	3.1	2.4	2.4	2.4	2.4	2.1	2.1	2.1	2.1	1.9	1.9	1.9	1.9												

- The 8176, 8170, 8160, and 6140 have 1.5 TB/socket memory capacity versions (8180M, 8170M, 8160M, and 6140M – not listed previously) with identical frequencies.

Figure 2.9: The maximum core frequencies of different Intel CPUs depending on the number of active cores when no SIMD extension is used and when AVX512 is used. Images taken from [Int20]

generic interfaces for all required functions and operators. The challenge is to find out how such a library must be realized to be applicable for the development of a query execution engine. We will cover this challenge in Section 3.2, where we introduce the general design of such a library.

2.3.2 Physical Constraints and Energy Consumption

Energy consumption is not only an important optimization goal because of environmental and economic reasons. It is also a limiting factor, especially when using specialized instruction sets like SIMD. The power consumption, and therefore also the energy consumption during a period of time, cannot be increased infinitely. Intuitively, one would assume that the power consumption of a chip scales with its area. This would require the power density to stay constant, even when the circuit elements, i.e. the transistors, become smaller. Indeed, this effect was already proven to exist in 1974 [DGR+74] and is called Dennard scaling or MOSFET scaling. During this time, "very small" circuits referred to dimensions in the order of one µm, while today's CPUs are manufactured with dimensions of barely more than 10 nm. Effects like an increased heat dissipation due to the dense chip layout played no major role in the 70s, but this has changed when the dimensions decreased by two orders of magnitude. Thus, Dennard scaling fails with modern CPUs. To prevent overheating, not all transistors can be powered at the same time [EBA+11], resulting in longer CPU cycles. The CPU core frequency decreases as

well as the power consumption. This leads to a number of side effects on the system when using SIMD extensions.

First of all, vector registers and vector instructions are more complex than their scalar counterparts and thus, they require more transistors. This means, that a signal takes longer to traverse the circuit, which implements an instruction, if this circuit is serially-connected. Since a CPU cycle has to be as long as a signal can take in the worst case, i.e. when it takes the longest path through the circuit, this causes longer cycles. On the one hand, complex instructions can be broken down into multiple cycles. On the other hand, it's sometimes useful to combine functionality into a single instruction to be able to internally pipeline the processes and gain performance or lower latencies this way. This has exemplarily be shown in [HKA+16] and in [AHF+14] for instruction sets, which processes compressed bitmaps and provide hardware-backed hashing.

Further, the power limits are met earlier with SIMD processing than with scalar processing. For instance, Figure 2.9 shows the maximum core frequencies for different CPUs when AVX512 is used. The tables are taken from a manual by Intel [Int20]. The more cores, i.e. the more transistors, are used, the lower the maximum frequency becomes. For comparison, the first CPU in the list scales down from 3.5 GHz with one active core to 1.9 GHz with 28 active cores. The same CPU without the use of SIMD scales between 3.8 GHz and 2.8 Ghz if only one resp. all 28 cores are used. This shows that not only the initial single-core frequency is lower when using SIMD, but it scales down faster when more cores are used. Since the power consumption is lower when the core frequency is lower, and we assume that the right choice of the SIMD extension finishes query execution faster than scalar processing, it can also be assumed that energy consumption goes down. In summary, it can be said that if not all transistors can be used at the same time, the right transistors must be used. Figure 2.8 showed that these are not always the same.

The frequency scaling of the CPU also has effects on other parts of a system, e.g. the memory bus. To reduce stalling cycles of the CPU, the memory bus frequency and the CPU frequency are synchronized. Ideally, the memory bus cycle length is an integral multiple of the CPU cycle length, which is influenced by the circuit length, i.e. the complexity of the used instructions. For this reason, the memory bus can be clocked down when a CPU core is clocked down, which is fine as long as all cores using the same memory bus run on the same frequency. However, if this is not the case, the other cores have to access the memory with the reduced memory bus frequency, too. This leads to a decreased effective bandwidth, an effect, which was already observed, e.g. in [UKHL16]. Alternatively, the bus frequency can be kept high, risking unnecessary and expensive stalling cycles and a higher energy consumption. The exact way this is realized depends on the hardware developer. Hence, each system can behave differently. The most efficient combination of CPU frequency and instruction set are not always obvious. This adds another challenge to the task of using the right transistors. For the software developer, this complex interplay means that energy consumption and performance are not necessarily proportional and that they influence each other. For this reason performance and energy consumption cannot be regarded as independent of each other. An optimizer should pay attention to this and regard energy-efficiency and performance not as a binary choice between optimization goals or as an automatic consequence following from each other. There are previous system, which already apply more sophisticated approaches [TWZX14, MRS+14, KHL18]. However, none of these systems consider the choice of the instruction set.

In Section 4 we will present a model and an according benchmark to characterize the relation between energy-efficiency and performance on individual platforms. Further, we propose an optimizer, which regards these optimization goals as equal and adjustable according to user defined constraints.

2.3.3 Summary

The current hardware landscape provides the user with a wide choice of vector sizes and instructions, where the right choice is not always the obvious one. For instance, the widest available vector registers are not always beneficial for the performance and can even slow query processing down. The reasons for this behavior and a selected solution for RLE compression is presented in Section 3.1. Further, the missing standardization of the available instruction sets complicates the port between different architectures or even vector sizes and the choice between them, because a domain expert of the target architecture and of the application is required. In Section 3.2 we will present a library , which aims to solve this challenge. Finally, the used instruction set has an effect on the frequencies of different parts of a system, e.g. the CPU and the memory bus. Therefore, not only the performance, but also the energy-efficiency is influenced. These two quantities are in a complex relation to each other, which has been examined before, but never with regard to the instruction set. We will close this gap in Chapter 4.

TEMPLATE VECTOR LIBRARY

As we have shown, vectorization can be beneficial to accelerate query execution. However, there are two main challenges introduced by growing vector register sizes and varied and evolving landscape of instruction sets: (1) The largest register size does not always offer the best performance, and (2) The same architecture offers different SIMD instruction sets and therefore also different implementations. Since there is no architecture independent standard of the functionality such instruction sets offer, porting between instruction sets goes far beyond renaming function calls and data types. In this chapter, we will cover both of these challenges[1]. We will address the first challenge in Section 3.1, where we have a deep-dive into the specific challenges of large vector registers and how special instructions can help to overcome these challenges. In Section 3.2 we will address the second challenge. We present a library, which hides the instruction set heterogeneity by introducing a common interface for so-called vectorization primitives, where the exact mapping to instruction sets and vector sizes is done by template parameters. On the one hand, this enables the application developer to vectorize their code in a *hardware-oblivious* way, because no knowledge about the target system is necessary to write vectorized code. The resulting code is portable to every instruction set with an according back-end. On the other hand, the back-end developer needs no knowledge about the application domain, because he only implements the primitives. We close this chapter with a discussion of related work and a summary.

3.1 SPECIAL INSTRUCTION SET INTEGRATION

The growing size of vector registers expose new challenges. Trivial ports of existing solutions are not always beneficial as shown in the last chapter. Special instructions are required to manage the larger amount of data elements in one vector register. The reasons for this can be distinguished into two groups: First, there are **inherent** challenges. Some operations require larger amounts of code or become a performance bottleneck because of their increased complexity. Second, there are also **algorithmic** challenges. Whenever algorithms are designed to load data into registers multiple times depending on runtime evaluations, this reloading of data produces more overhead the larger the vector registers are. Additionally, the larger the register is, the higher is the probability of collisions (equal values) within the registers, which often serves as a trigger for the said reloads. This chapter shows different situations, where the instructions, commonly found in SIMD instruction sets, are not sufficient to handle the vector sizes found in modern CPUs. Further, we will discuss an exemplary solution for a lightweight compression (RLE), which makes use of specialized instructions introduced with AVX512.

3.1.1 Inherent Challenges

With a growing size of vector registers, a higher number of elements fits into each register. This raises two major challenges:

1. The *logic* becomes more complex causing a loss of performance, i.e. additional scalar registers and more expensive instructions are required.

[1]Parts of the material in this chapter have been developed jointly with Johannes Pietrzyk, Patrick Damme, Alexander Krause, Erich Focht, Dirk Habich, and Wolfgang Lehner. The chapter is based on [U+18, UPD+20]. The copyright of [U+18] is held by the Institute of Electrical and Electronics Engineers (IEEE); the original publication is available at https://doi.org/10.1109/ICDEW.2018.00023. [UPD+20] is published under a Creative Commons Attribution License; the original publication is available at http://cidrdb.org/cidr2020/papers/p28-ungethuem-cidr20.pdf.

```
template<typename T>
...
static __m128i
gather( T const * const p_DataPtr, __m128i p_vec ) {
        return _mm_set_epi64x(
                p_DataPtr + _mm_extract_epi64(p_vec,1) * 8),
                p_DataPtr + _mm_extract_epi64(p_vec,0) * 8)
        );
}
```

Figure 3.1: Gather function for SSE. Type checks and casts are not shown for the sake of simplicity.

2. The *code* becomes more complex the more elements there are in a vector, up to a degree where it is not manually manageable anymore.

To illustrate the first point, the *gather* function serves as an example. The already introduced *compress store* will show the potential degree of the code complexity.

Logic Complexity

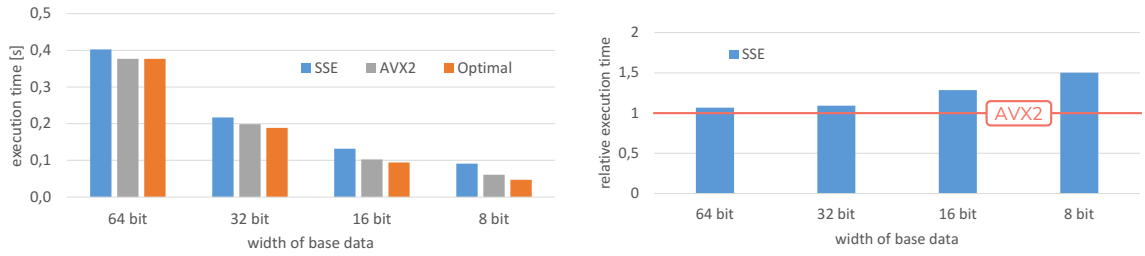
The *gather* operation is a crucial part of different query operators, e.g. the projection or some join variants. The parameters of a *gather* are a memory address and a vector with offsets as arguments. It returns a vector containing the values at the memory address with the added offsets. If we assume that each value of a dataset has a bitwidth of 64 bit, a *gather* is easily implemented for the 128-bit SSE registers: There are two values to load from memory. The two offsets are extracted into scalar registers and added to the address. Then, a set instruction is used to fill a register with the values at the two retrieved addresses. Figure 3.1 shows the according code of a gather function.

Going from 64-bit values to 32-bit values, there are four offsets to extract and four memory addresses to compute. Together with the base address pointer, this accumulates to 9 required registers. This does not pose a big challenge, because there are 16 general purpose 64-bit registers on modern Intel CPUs, which are available in user mode. However, when gathering 16-bit values, already 17 registers needed. This means, that not the whole gather function can be executed completely in registers. When gathering 8-bit values, not even half of the computed and extracted addresses and offsets fit into the registers at the same time. Additionally, each extraction from the vector register and each computation of a new address is a separate instruction. Note that there are no actual 16-bit or 8-bit general-purpose registers, but the least significant bits of a larger register are used to store smaller data types. When extracting a value from a vector register into one of the scalar general purpose registers, the remaining bits are zeroed[Int16]. This is likely to result in higher latencies the more elements there are in a vector register.

Fortunately, since AVX2, there are dedicated gather intrinsics for 32-bit and 64-bit values, which map directly to the *vpgatherqq*- and *vpgatherdd*-machine-instructions. These intrinsics can also be used to gather 8-bit and 16-bit elements. For instance, the source code for 16-bit elements is shown in Figure 3.2. First, the elements in the offset-vector are copied into a second vector and shifted in a way, that each offset uses the 16 least significant bits of a 32-bit element (l. 5-6). Using these two shifted vectors, the data elements are gathered into two vector registers (l. 7-8). After this, all elements in one of the registers with the gathered values are shifted by 16 bit to use the 16 most significant bits of each

```
1  template<typename T>
2  ...
3  static __m256i
4  gather( T const * const p_DataPtr,  __m256i p_vec ) {
5      __m256i p_vec_1 = _mm256_srli_epi32(p_vec, 16);
6      __m256i p_vec_2 = _mm256_srli_epi32(_mm256_slli_epi32(p_vec, 16),16);
7      __m256i d_vec_1 = _mm256_i32gather_epi32(p_DataPtr, p_vec_1, 2);
8      __m256i d_vec_2 = _mm256_i32gather_epi32(p_DataPtr, p_vec_2, 2);
9      return _mm256_or_si256(
10         _mm256_slli_epi32(d_vec_1, 16),
11         _mm256_srli_epi32(_mm256_slli_epi64(d_vec_2, 16), 16)
12         );
13 }
```

Figure 3.2: 16-bit gather function for AVX2. Type checks and casts are not shown for the sake of simplicity.



(a) (b)

Figure 3.3: Projection of 10^8 values with different bitwidths of the base data elements. (a) Runtime. The figure also shows the optimal runtime, which would be achievable if it scaled linearly. (b) The runtime offset of using SSE over AVX2. The *gather* function, which does not have an according intrinsic in SSE, is a crucial part of the projection.

element, while the other register stores the gathered values in the 16 least significant bits (l. 10-11). Finally, the two registers are combined with a bitwise OR (l. 9-11).

This requires no additional scalar registers and only 6 vector registers, while there are at least 16 256-bit registers available with AVX2. The according 8-bit version requires 10 vector registers, which is still within the available amount. Hence, the expected performance loss for AVX2 is smaller than for SSE.

To check whether our assumptions are true, we implemented a projection, which uses the *gather* as a core function. Then, we created a synthetic dataset containing 10^8 values and the same amount of arbitrary offsets. We did this for the bitwidths 8, 16, 32, and 64. Then we called the projection with a pointer to the synthetic dataset and the offsets for different processing styles using AVX2 and SSE, as well as 8-, 16-, 32-, and 64-bit base data. The system we used was equipped with an Intel Xeon Gold 6130. The dataset fit completely into the main memory. The measured runtimes are shown in Figure 3.3(a). We additionally showed the theoretical optimum, which could be achieved if the AVX2-*gather* scaled linearly. The graph shows, that neither SSE nor AVX2 scale linearly. However, AVX2 is noticeably faster than SSE, although a projection should be highly bound by the bandwidth of random memory access. Obviously, the workarounds necessary for SSE, introduce enough overhead, that this is not the case. On the other hand, the workarounds for 16 and 8 bit in AVX2 do not scale worse than the 32-bit intrinsic. We assume that this is because all additional computations can be done without accessing

```
//a) AVX-512
_mm512_mask_compressstoreu_epi64(dataPtr, mask, vec);
//b) SSE
switch (mask){
    case 0: return; //store nothing
    case 1: _mm_storeu_si128(dataPtr, vec);
            return; //note: depending on the base data type, using
                _mm_extract* can be possible
    case 2: vec = _mm_shuffle_epi8(vec, _mm_set_epi8
        (7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8));
            //exchange lanes before storing
            _mm_storeu_si128(dataPtr, vec);
            return;
    case 3: _mm_storeu_si128(dataPtr, vec);
            return; //store everything
}
//c) NEON
switch (mask){
    case 0: return; //store nothing
    case 1: vst1q_lane_u64(dataPtr, vec, 0); return; //store 1st lane
    case 2: vst1q_lane_u64(dataPtr, vec, 1); return; //store 2nd lane
    case 3: vst1q_u64(dataPtr, vec); return; //store everything
}
```

Figure 3.4: Different vectorized *compressstore* specializations.

the cache, which is fast enough to do it while waiting for memory access. In contrast to that, the SSE workaround uses more instructions and, for 8 and 16 bit, more values than there are available registers. This results in a higher computational overhead and cache accesses, which cannot be hidden by the latency introduced by random memory access. In Figure 3.3(b) the overhead of using SSE over AVX2 is shown for different base data widths. While this overhead is negligible for 64 and 32 bit, where all values fit into the registers and the number of additional instructions is still manageable, it increases to 28.5% and 50% for 16 bit and 8 bit respectively.

The gather function is not the only case, where the increasing vector widths demand new instructions because they become a performance bottleneck otherwise. The same holds true for other operations, e.g. permutations, scatter, horizontal aggregations, to name just a few.

Code Complexity

The additionally required logic is not the only issue arising from large vector registers. In some cases, the sheer amount of source code becomes unmanageable. One of these cases is the already introduced *compress store*, which stores selected elements of a vector consecutively, where the selection is done by a bitmask. In Figure 3.4, we show the implementations for 64 bit base data using SSE, AVX512, and Arm Neon. For smaller base data, i.e. 8, 16, and 32 bit, the code grows for every instruction set, which does not have a dedicated *compress store* instruction. Currently, this applies to all SIMD extensions, we have used, except for AVX512. For instance, with SSE, 128-bit registers can be used. Assuming base data with 32 bit, the corresponding bitmask contains 4 bit, because 4 elements fit into one vector. This bitmask has one out of $2^4 = 16$ possible values and each of these possible values must be treated in the implementation. If the base data is smaller, there are even

```
1  template<typename T>
2  ...
3  static void
4  compressstore( T * p_DataPtr,  _m128i p_vec, uint16_t mask ) {
5      while(mask != 0){
6          if((mask & 0x1) == 0x1){
7                  *p_DataPtr = _mm_extract_epi8(p_vec, 0);
8                  p_DataPtr++;
9          }
10         mask = (mask >> 1);
11         p_vec = _mm_srli_si128(p_vec, 1);
12     }
13     return;
14 }
```

Figure 3.5: 8-bit compress store for SSE. Type checks and casts are not shown for the sake of simplicity.

$2^8 = 256$ values for 16 bit and $2^{16} = 65\,536$ values for 8 bit. This would obviously result in a switch-case statement, which is not written by hand anymore. Moreover, if the value of the bitmask is one of those at the end of the 65 536 case-statements, the comparisons done by the time this statement is reached, will have had a significant negative impact on the performance. The case becomes even more severe when assuming 8-bit base data in a 256-bit register, which results in 2^{32} different possible mask values. Thus, another solution must be found.

One way of doing a *compress store* for 8 bit manually using SSE, is shown in Figure 3.5: The least significant bit of the mask is checked (l. 6). If it is set, the corresponding value in the vector is extracted and stored (l. 7), and the output pointer is incremented (l. 8). Afterwards, the mask is shifted by one bit (l. 10) and the vector is shifted by one byte (l. 11). Then, the least significant bit of the mask is checked again. This is repeated until no more set bits in the mask are left (l. 5). This solution can be further optimized, e.g. by introducing a switch-case-statement, which only treats those cases likely to appear often, e.g. consecutive matches at the beginning of the mask. However, it is not as efficient as a hardware supported instruction.

A similar code complexity is also reached with other functions if there is no according intrinsic, i.e. all masked operations.

3.1.2 Algorithmic Challenges

When algorithms are vectorized, this is often done for SSE or AVX, which are common SIMD instruction sets available on all non-museum Intel CPUs. The provided vector sizes are 128 bit (SEE) and 256 bit (AVX). This means, one vector register can hold two 64 bit elements or four 32 bit elements with SSE, and four 64-bit elements or eight 32-bit elements with AVX. These elements are evaluated in parallel by using the according intrinsics, but the originally sequential algorithm often requires an additional step, when it is vectorized. This step can be one of the two following cases:

1. There are possible collisions (equal values) within a vector register before or after the evaluation, which have to be detected and treated separately. This introduces additional memory access and computation.

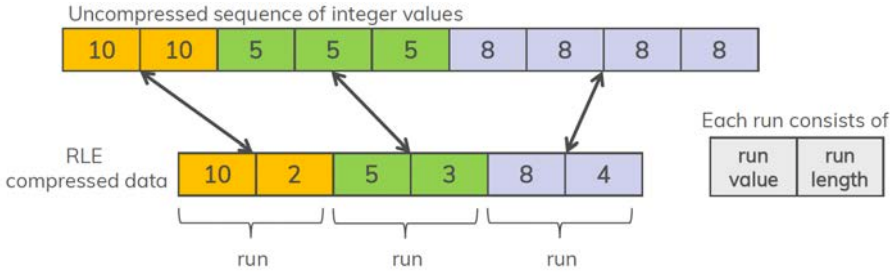


Figure 3.6: Example for input and output of RLE compression.

2. Only a part of the evaluated register meets a condition. The remaining part has to be reloaded and evaluated again.

If there are never more than eight elements in one register, this additional step is manageable. However, with a growing number of elements per vector, the potential for collisions in a register increases as well as the overhead for additional memory access to fill another vector register. The impact of additional gather instructions caused by collisions in hashing, are described in [P+19a]. An example, where additional memory access is introduced because a register meets a condition only partially, is the state-of-the-art vectorized run length encoding (RLE). RLE is one of many compression techniques, which can be used to compress base data or even intermediates to accelerate query processing [D+19]. In Figure 3.6, we show an example of the input and output of RLE on a sequence of integer data. In this section, we will use RLE as an example to illustrate the significance of the additional load instructions and to show a solution. First, we show the comparison based state-of-the-art vectorization of RLE, which reloads those parts of a vector register, which are not part of the current run. Then, we show the shortcomings of this comparison-based approach in more detail. Finally, we propose and evaluate an alternative implementation relying on a special instruction, which detects conflicts within one register.

State-of-the-Art Vectorization of RLE

Generally, to compress a sequence of integers with RLE, the corresponding runs have to be determined and this can be done by comparing each element with its predecessor. If they are equal, a run continues. If they are not equal, a new run starts. These comparisons can be done for more than one element at once using SIMD instructions as shown in [DHHL17, UDP+17]. In detail, this state-of-the-art RLE comparison-based vectorization works as follows, whereby the authors used 128-bit vector registers:

1. One 128-bit vector register v_1 is loaded with four copies of the current input element.
2. The next four input elements are loaded into a vector register v_2.
3. The intrinsic `_mm_cmpeq_epi32()` is employed for a parallel comparison, so that the four elements in v_1 and v_2 are pair-wise compared at once. The result is stored in a vector register.
4. Next, a 4-bit comparison mask is obtained using the intrinsic `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing one-bits in this mask is the number of elements for which the run continues. If this number is 4, then a run's end has not been reached and the execution continues at step 2 (new iteration). Otherwise, a run's end is reached that means that run value and run length are appended to the output. The execution continues with step 1 at the next element after the run's end (new iteration).

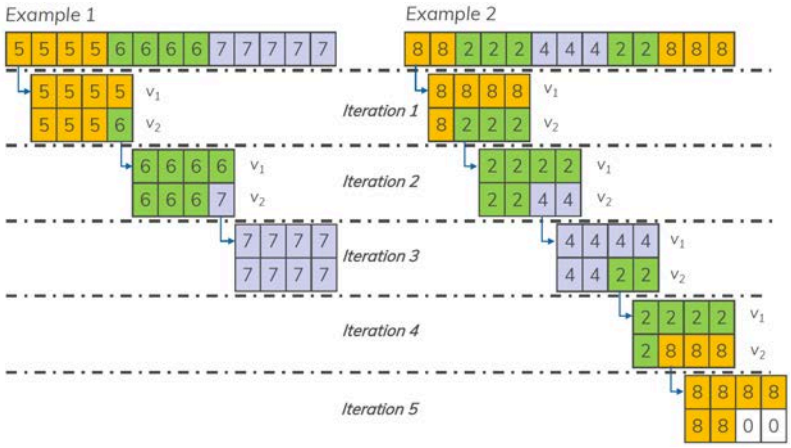


Figure 3.7: Execution behavior of the comparison-based implementation. As illustrated, the number of necessary iterations depends on data characteristics.

The execution behavior of this vectorization concept is depicted in Fig. 3.7 for two different data sets. A detailed description follows in the next section. We refer to this 128-bit implementation as RLE128. Since only common intrinsics are used, this comparison-based implementation can easily be adapted to 256 and 512 bit-wide registers by loading more elements in the wider registers and by using the appropriate intrinsics of AVX2 (256 bit) or AVX-512. Additionally, step 3 and 4 can be merged into one step in AVX-512, because there is an intrinsic producing a bitmask directly from the comparison. The corresponding implementations are denoted as RLE256 and RLE512.

Shortcomings of this Comparison-based Vectorization

As already mentioned, Fig. 3.7 highlights the resulting execution behavior for two different input sequences of integers. Both have in common that in each iteration, four integers are loaded and compared with a vector containing the current run value. If this comparison for equality is not true for all elements, the current run ends. In this case, the register with the run value is filled with four copies of the new value and the next four elements after the beginning of the new run are loaded. Obviously, the number of necessary iterations is data dependent and Fig. 3.7 shows that clearly. In detail, *Example 1* in Fig. 3.7 depicts a *fully vectorized* execution behavior. Fully vectorized means that each integer value is only processed once. In contrast to that, in *Example 2* several integers are loaded and compared multiple times. The redundant processing is usually negligible as long as the overhead is not dramatic.

To analyze the magnitude of this redundant processing, we counted the load instructions for different average run lengths and all possible variances for each average run length, whereby we used an input sequence with 100 million integers in all experiments. For instance, the maximal variance for an average run length of 5 is ± 4 resulting in the interval $[1, 9]$ for the possible run lengths. Then, we selected the minimal and the maximal number of load instructions and visualized them in Fig. 3.8(a) for RLE128, RLE256, and RLE512. The x-axis shows the average run length and the y-axis shows the number of loaded elements as a percentage of the elements in the input sequence, e.g. 200% means that on average every element is loaded twice. The colored area shows the range between the maximal and minimal number of load instructions. Fig. 3.8(b) shows a close up of Fig. 3.8(a) with the y-axis ranging only until 200%. From these experiments, we can conclude:

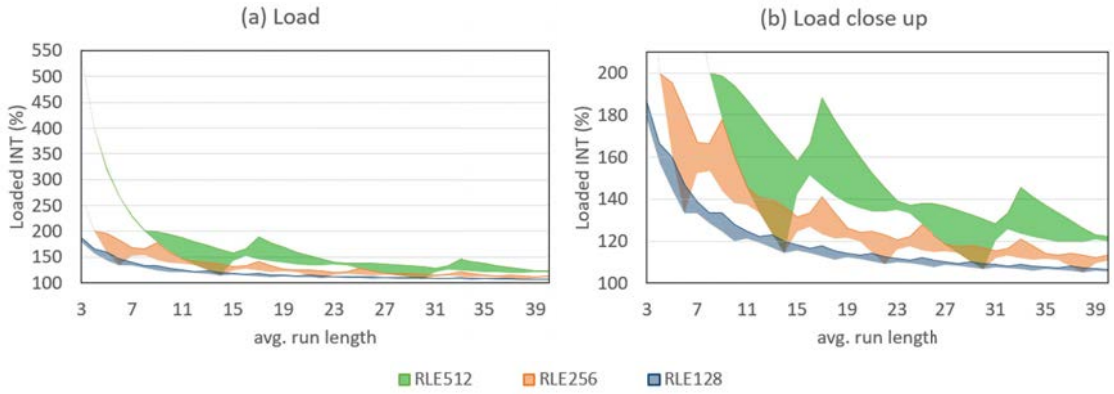


Figure 3.8: (a) The number of loaded integers as a percentage of the integers in the uncompressed data set. Depending on the vector width, the average run length, and the variance of the run length, the number of loaded integers differs heavily. In particular for data sets with small average run lengths. (b) A close up of (a) to show the repeating pattern at every vector size.

1. The state-of-the-art RLE vectorization uses a significantly higher number of load operations for sequences with short runs than for sequences with long runs.
2. The redundant processing dramatically increases with increasing vector widths. For example, RLE512 processes each element 5 times on average when the average run length is 3. Furthermore, not only the absolute number increases, but also the size of the covered area grows.
3. There is a pattern with a minimum and a maximum spanning exactly one vector width, which is repeated for every vector width (in number of elements).
4. For large run lengths, the number of loaded integers approaches more or less 100%, i.e. every value is only processed once, which is the optimal scenario.

Impact on Performance The presented high proportion of redundancy for sequences with small runs has a negative effect on the performance–measured in million integers per second (mis)–as illustrated in Fig. 3.9(a). In this experiment, we used again input sequences with 100 million integers and varied the average run length, whereby we used a fixed variance of ± 5. However, the results were the same for other data characteristics. As shown, only run lengths, which are greater than ~ 150 reach the peak performance for all vector widths, while small run lengths reach only a fraction of the peak performance. Additionally, the performance increases not even smoothly for RLE512. This becomes more obvious when looking at the speed up in Fig. 3.9(b). The speed up of RLE512 compared to RLE128 increases until a run length of ~ 8 is reached and decreases afterwards. The sampled run lengths in this region are 20 and 36, both being shortly after a maximum load number in Fig. 3.8. For larger run lengths, the number of loaded values becomes smaller and the speed up becomes constant.

Conflict Detection-based Vectorized RLE

Intel's latest version of their vectorization extension is AVX-512. In addition to an increased vector width of 512-bit (16 x 32-bit), AVX-512 also offers a variety of new instructions. One of the new instruction feature sets is called *Conflict Detection* (AVX-512 CD) which allows the vectorization of loops with possible address conflicts. This instruction

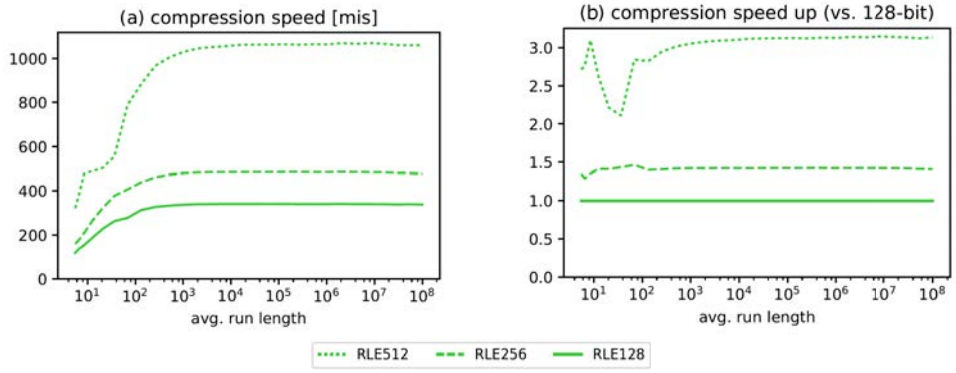


Figure 3.9: RLE compression speed and speed up for different average run lengths, a fixed run length variance of ± 5, and different vector widths.

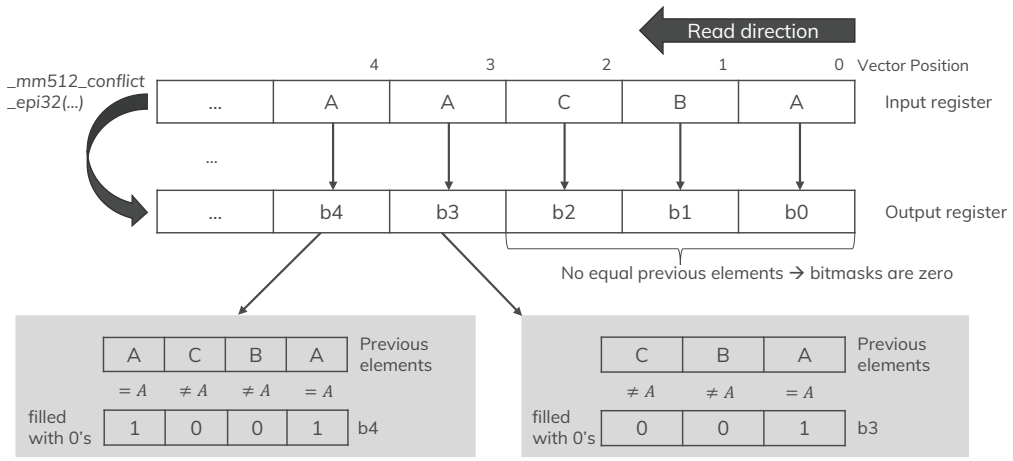


Figure 3.10: Example for the *_mm512_conflict_epi32* intrinsic.

feature set is currently supported by Intel Xeon Phi Knights Landing (KNL) and by recent Xeon processors.

Some key features of AVX-512 CD are (1) the generation of conflict free subsets, i.e. subsets which contain no equal elements, and (2) the count of leading zeros of the elements in a vector. For example, the intrinsic `_mm512_conflict_epi32` creates a vector register containing a conflict free subset of a given source register. An example for this is shown in Fig. 3.10. In other words and as illustrated in this figure, this intrinsic transforms a vector register with 16 32-bit elements (illustrated by A, B and C) in a new vector register with 16 bitmasks (each represented by 32-bit values). Each bitmask encodes the positions of equal previous elements in the vector. The bitmasks for the first three elements A, B, and C are zero in our example, because there are no equal previous elements. The A element at the third position in the input register is in conflict (equal to) with the element at position 0 in the input register. Thus, the least significant bit of the corresponding bitmask is set to 1, the rest of the bitmask is filled with zeros. The element A at position 4 is in conflict with the previous elements at positions 3 and 0 (equal previous elements). Therefore, the corresponding bits in the bitmask are set to 1, all other bits are zero. Another CD-feature is the intrinsic `_mm512_lzcnt_epi32`, which counts leading zeros. Given a vector of 16 values, this intrinsic counts the number of leading zeros for all values at once and writes the results in a vector register with 16 values.

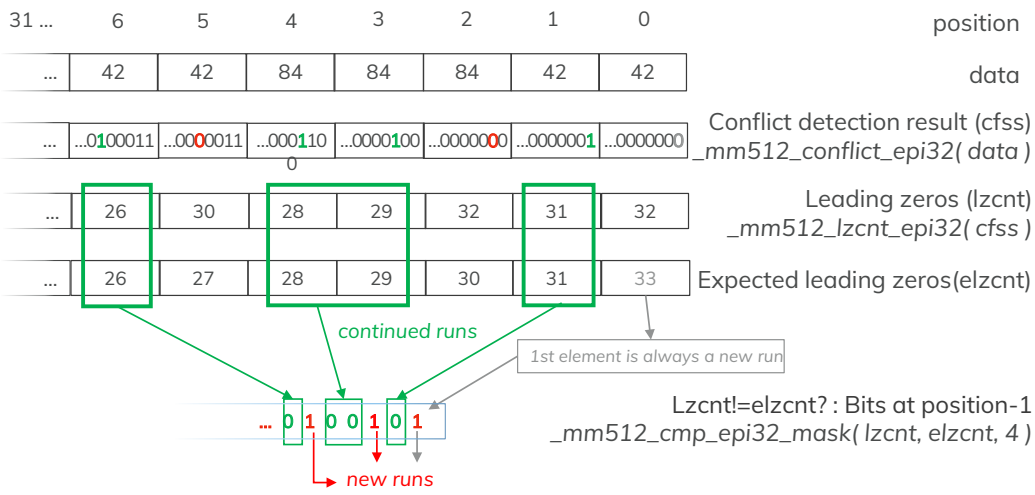


Figure 3.11: Run detection using *conflict detection* instructions

RLE Implementation Concept with AVX-512 CD To overcome the presented shortcomings of the comparison-based RLE vectorization, our novel approach–called *RLE512-CD*–uses the conflict detection innovations of AVX-512 in an appropriate way. Generally, our novel approach consists of four steps, which are repeated until all input elements are processed:

Loading Step: In this first step, 16 input elements are loaded into a 512-bit vector register.
Run Detection Step: In the second step, we detect if there are any runs beginning in this register and where they begin.
Run Length Detection Step: The run length of all finished runs have to be determined in the third step.
Storage Step: The determined runs are written to memory.

While the *loading step* is trivial, the steps 2-4 are explained in more detail below.

Run Detection Step To avoid any redundant element processing for small run lengths, the main challenge of this step is to detect *all* runs included within the loaded 16 elements. This challenge can effectively be realized using the AVX512-CD innovations as illustrated in Fig. 3.11:

In the first sub-step, we create a new vector register containing a **c**onflict **f**ree **s**ub**s**et (*cfss*) of the given source register with the 16 loaded elements using the `_mm512_conflict_epi32` intrinsic. The example in Figure 3.11 shows the first 7 values of a vector register containing two different values spread over 3 runs. As described above, the newly created vector consists of 16 bitmasks, where each bitmask shows the equality to all previous elements. However, for detecting a run it is sufficient to know if the direct predecessor of an element is equal because all elements are either the beginning of a new run or the continuation of another run. If an element is equal to its direct predecessor, the element continues a run. If they are not equal, a new run starts. Hence, only one bit in every bitmask of *cfss* is of interest, i.e. the bit which indicates the equality with the direct predecessor. To find this bit for all elements in parallel, two more operations are necessary:

First (second sub-step), we count the leading zeros of all bitmasks in *cfss* (*lzcnt*). The number of leading zeros should decrease with every element if a run is continued because

Figure 3.12: Run length determination using *conflict detection* instructions.

there is always one more bit set in the subsequent element, e.g. the bitmask at position 1 should have $32 - 1 = 31$ leading zeros, the bitmask at position 2 should have $32 - 2 = 30$ leading zeros and so on. If a run is not continued, the next bit is not set and the number of leading zeros does not decrease. To find out if the number of leading zeros is decreasing, we compare *lzcnt* with a predefined vector, containing decreasing numbers, for inequality (third sub-step). As shown in Fig. 3.11, this comparison returns 0 for every element which continues a run. Vice versa, it returns 1 for all elements which start a new run. Thus, the position of the ones in the final bitmask indicates the position of the start of all runs in this register. Note that the first element always starts a new run.

Run Length Detection Step With the previous step, we know the start positions and the run values of all runs within the register. The next challenge is to determine the run length of each run. Fundamentally, the run length is already encoded in the results of the conflict detection (*cfss*) operation, because each continuous sequence of 1s in the bitmasks indicates a subsequent occurrence of equal numbers. Hence, the number of the most significant subsequent 1s in the bitmask of every last element of a run indicates the length of the run. To get this number, at first the position of the last element of every run has to be determined. This can be done by using the bitmask generated as the result of the run detection (*cfss*). Since every 1 in this bitmask indicates the beginning of a new run, we can get the end of the runs by shifting this mask one bit to the right. Now every 1 indicates the end of a run. Then, the bitmasks at these end positions in the output of the conflict detection (in *cfss*) are selected. In Fig. 3.12, which continues the example from Fig. 3.11, one bitmask is selected as an example. In this example the second run, consisting of 3 elements is treated. The continuous sequence of 1s is highlighted. There are only 2 instead of 3 set bits because the bit of the first element, i.e. the least significant element, of a run is always set to 0. We will add this bit later. In order to retrieve the number of subsequent set bits in this bitmask, 3 sub-steps are executed:

1. Shift the elements in the result of `_mm512_conflict_epi32` by the number of leading zeros (leading zeros were derived during run detection). In Fig. 3.12 we shift by 28 bits. Now, the sequence is at the beginning of the bitvector.
2. There is no intrinsic for counting leading 1s, so the result from the previous sub-step is inverted.
3. Then, the leading zeros are counted in the third sub-step. In the example, there are two leading zeros.

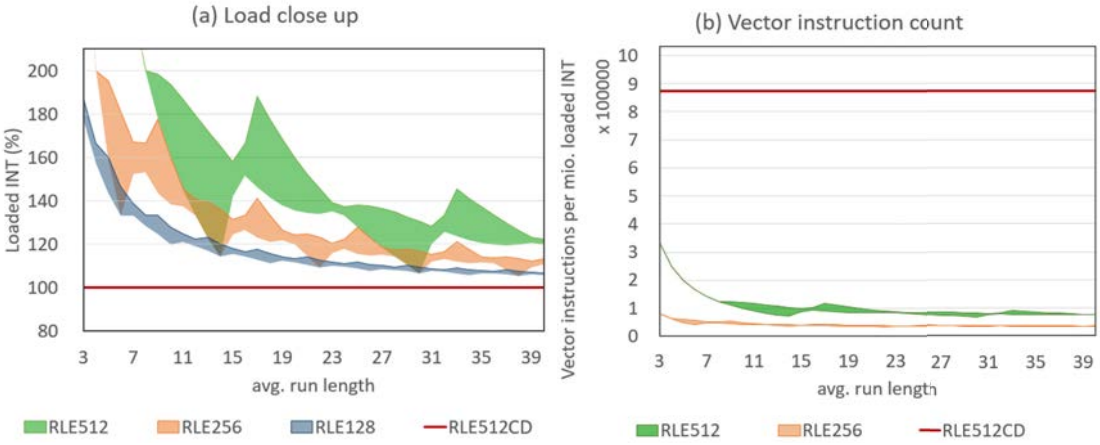


Figure 3.13: (a) The number of loaded integer as a percentage of the integers in the data set. Only *RLE512-CD* shows a constant behavior. (b) The number of vector instructions per million loaded integers (excluding loading and storing) is significantly higher for *RLE512-CD* compared to RLE512 and RLE256. This shows that the lower number of loaded integers do not come for free.

Since the bit for the first element of a run is always set to 0 during conflict detection, the result has to be increased by 1. Hence, the run length for the second run is $2 + 1 = 3$. In our implementation, these steps are executed in parallel for all runs by using the intrinsics shown in Fig. 3.12.

Storage Step Before storing the results, it must be checked whether the first run of a register is a continuation of the last run of the previous register. If it is a continuation, the run lengths are added and the run is stored once. While these are trivial steps, we avoided branching by applying bitmasks instead of conditions, i.e. the comparison between the last and the first run of two registers returns whether the last bit in a bitmask is set. This bitmask is then used to add the run lengths by applying `_mm512_mask_add_epi32`, which adds the content of two registers only if the corresponding bit in the bitmask is set.

Finally, the run values and run lengths must be written back to main memory. For this, there are two possible cases: (a) per integer or (b) per vector. Case (a) represents the output format proposed by the state-of-the-art implementation [DHHL17], where a sequence of (value, run length)-tuples is stored. An advantage of option (a) is that the output is independent of the vector word size. The disadvantage is that the values and run lengths cannot be loaded sequentially into a vector register again for processing the compressed values, e.g. for aggregating. Case (b) stores sequences of values and run lengths which are as long as a vector word, e.g. 16 values followed by 16 run lengths. Option (b) requires the vector word width as necessary metadata but it is also ideal for processing the compressed data with vector instructions. We implemented both cases, case (a) using a scatter store provided by AVX-512 and case (b) using the result of the *run detection* as a write mask.

Evaluation

In this section, we evaluate and compare our novel CD-based RLE implementation (RLE512-CD) with the state-of-the-art implementation. For this evaluation, all implementations are done with C/C++ and we compiled them with g++ 7.0.1 using the optimization flag -*O*3. Then, all experiments were executed on an Intel Xeon Phi KNL 7250

and on an Intel Xeon Gold 6130 supporting all vector widths of 128 (SSE), 256 (AVX2), and 512 (AVX-512). The maximum core frequency is 1.6 GHz. Moreover, all experiments were performed with a benchmark framework for data compression techniques [DHL15] running entirely in main memory and single-threaded.

Data Processing Behavior We already analyzed the processing behavior of the state-of-the-art implementation. For this analysis, we also used the above mentioned evaluation setting with the Xeon Phi. As we have shown, this implementation suffers from a significantly higher number of redundant load operations for sequences of integer values with short average run lengths. In contrast to that, our novel *RLE512-CD* implementation is branch-free and every integer sequence value is only loaded once as illustrated in Fig. 3.13(a). The y-axis shows the number of loaded integers as the percentage of the integer count in the uncompressed data for RLE128, RLE256, RLE512, and for *RLE512-CD*. It is clearly visible that our novel implementation loads the input data set only once, independent of the data characteristics, and that the amount of loaded data is smaller than for the state-of-the-art implementation. Additionally, we observe that the difference is smaller when the average run-lengths are longer.

However, this constant data loading behavior comes at a cost. The total number of executed vector instructions of *RLE512-CD* is higher than for the state-of-the-art implementation. Fig. 3.13(b) shows the number of vector instructions per million loaded integers (excluding operations on masks and other scalar operations) for RLE512, RLE256 and *RLE512-CD*. Thus, it comes down to the number of loaded and processed integers versus the amount of executed instructions. Depending on the system, this can have different effects on the compression speed.

Performance Fig. 3.14 shows the compression speed for RLE512 and *RLE512-CD* with two different storage options: *RLE512-CD* stores the result data integer-wise like RLE512 with a scatter store while *RLE512-CDAligned* stores the result vector-wise. Again, each run length has been tested with all possible run length variances. The first obvious finding is that RLE512-CD shows an almost constant compression speed as expected. The second finding is that the run length, where one or the other algorithm is more effective, varies depending on the system. However, for both systems, the scatter store used in RLE512-CD is too slow to compete with RLE512. Only for a run length of 3, RLE512-CD is slightly faster than RLE512. For RLE512-CDAligned, there are 3 different regions: (1) RLE512-CDAligned always outperforms RLE512 for very small run lengths ($<$12 resp $<$13). (2) Between the run lengths of 12 and 40 resp. between 13 and 24, there is no binary decision possible between RLE512 and RLE512-CDAligned. RLE512 shows the highest peak performance but also the lowest possible performance. RLE512-CDAligned does not reach the peak performance but guarantees an almost constant compression speed, i.e. it is robust. Although, there is a small difference between the fastest and the lowest compression speed of RLE512-CDAligned on the Xeon Gold, the size of this second region is roughly the same for the best and the worst case. (3) for run lengths greater than 40 resp. 24, the state-of-the-art implementation always shows the highest compression speed. Hence, at the transitions of these regions, the applied implementation should be changed. Additionally, in region (2) a decision between maximal peak performance and robustness must be made.

The same regions as for the compression speed (in mio integers per second) can be shown for the speed up in Fig. 3.14. Here, the baseline is RLE512 and the maximal and the minimal speed up is shown for RLE512-CDAligned. The lower curve compares to the maximal compression speed of RLE512 and the upper curve compares against the minimal compression speed. The graph shows that the chances to gain a speed up greater than 1 are higher, the lower the run length is. As already mentioned, this graph looks different on different systems, where the execution of vector instructions or the loading of a vector register is faster or slower. Additionally, in a multi-threaded scenario the loading of integers might become a bottleneck earlier, e.g. because of shared caches.

Figure 3.14: The compression speed for RLE512 varies depending on the run length and the variance of the run length, while our novel implementation shows a constant compression speed for two different tested systems. The costs of the scatter store for RLE512CD are clearly visible. The speed up for RLE512-CDAligned compared to RLE512 scales accordingly.

3.1.3 Summary

In this section, we examined the challenges coming with large vector registers. On the one hand, there are inherent challenges, which originate from the sheer amount of elements per register resulting an increased complexity of the logic as well as the code. We could show that this quickly leads to a loss of performance and to code sizes, which are not manageable anymore. On the other hand, there are algorithmic challenges, because many algorithms were designed for scalar processing or for the common but small 128- or 256-bit registers. We explained this with a state-of-the-art example for lightweight compression (RLE) and provided a different solution using features of Intel's most recent SIMD instruction set AVX512. We could show that this solution has a constant compression speed, which is higher than the state-of-the-art approach for small run lengths.

3.2 HARDWARE-OBLIVIOUS VECTORIZATION

Vectorization is usually done by using intrinsics, which are functions wrapping the underlying machine calls. Unfortunately, these intrinsics are different for each instruction set. Even the same manufacturer is not entirely consistent with the naming scheme across their different SIMD extensions. Furthermore, not only the naming schemes are different, but also the function range as we will show in the following section. To hide this heterogeneity, we developed a library, the Template Vector Library (TVL)[2], which enables *hardware-oblivious* vectorization. Unlike existing approaches, e.g. [L+14, KL12b], we do not rely on overwriting standard operators and do not abstract the explicit vectorization away. This enables the use of functions beyond the basic arithmetic operations, e.g. the *conflict detection* provided by AVX512 or *gather/scatter* operations, to achieve the best performance. We will discuss such abstraction guidelines for our TVL in the following section in more detail. Then, we show our realization for the query processing domain. Finally, an evaluation will show the efficiency of our TVL for different hardware and compare it with manual implementations.

3.2.1 Abstraction Guidelines

The SIMD hardware landscape is increasingly diverse as illustrated in Figure 2.6. For this diversity, three metrics are important: (1) The number of available vector instructions[3], (2) The vector length, and (3) The granularity of the bit-level parallelism, i.e., on which data widths the vector instructions are executable. Figure 2.6(a) shows the metric values for two recent Intel architectures: *Xeon Skylake* and *MIC Knights Landing*. Generally, Intel offers several SIMD extensions such as SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions), AVX2, or AVX-512. Both architectures in Figure 2.6(a) offer SIMD functionality up to Intel's latest extension AVX-512, resulting in a very high number of vector instructions with three different vector lengths of 128-, 256-, and 512-bit. The SIMD processing can be done on 64-, 32-, 16-, and 8-bit data elements on both architectures. As depicted, the number of vector instructions differs between both architectures, because not all subsets of AVX512 are supported by all architectures.

In contrast to that, ARM, for example, pursues a completely different approach as highlighted in Figure 2.6(b). Instead of providing a high number of vector instructions, ARM supports much wider vector lengths. While the ARM NEON extension (available in ARMv7-A and ARMv8-A) was limited to a vector length of 128-bit, the Scalable Vector Extension (SVE) (available in ARMv8-A AArch64) aims at supporting much more vector lengths from 128 to 2,048 bits, in 128-bit increments [S+17]. In all cases, the SIMD processing can be done on 64-, 32-, 16-, and 8-bit data elements.

With this diversity in mind, developing a *hardware-oblivious* concept will become equally important as achieving optimal performance. Therefore, the *hardware-oblivious* concept should provide the following core aspects:

- **Portability and Extensibility:** The vectorized code written in a *hardware-oblivious* way should be easily portable between vector processing units with different SIMD capabilities. In addition, the implementation effort for the integration of new SIMD functionalities offered by a specific vector processing unit should be manageable. Of course, this also means that the *hardware-oblivious* approach should be extensible with new functions that are necessary for the explicit vectorization of application logic.

[2]Source code available at https://github.com/MorphStore/TVLLib, last accessed 23/07/2020
[3]We counted the instructions in the hardware vendor intrinsic guides.

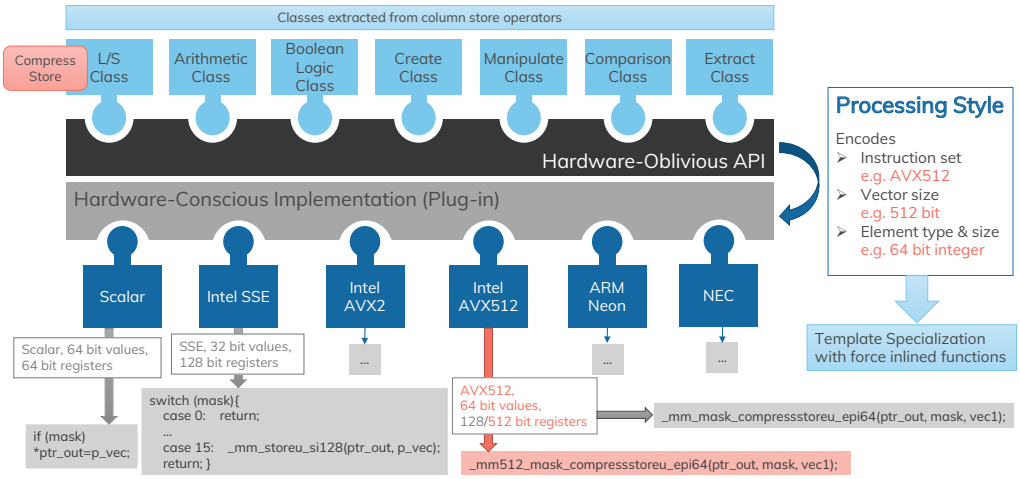


Figure 3.15: Architecture of the Template Vector Library. The compress store is illustrated as an example.

- **Enabling Explicit Vectorization:** To achieve the best performance, explicit vectorization of application logic is still the best way [P+19b]. Even if a given code can be auto-vectorized implicitly, this poses new challenges to the overall system as described in [KL12b]. Thus, a *hardware-oblivious* concept should enable an explicit vectorization for the diversity in the SIMD hardware landscape (see Figure 2.6) with the following properties. On the one hand, the diversity characteristics of SIMD functionality, vector length and the granularity of the bit-level parallelism should be treated independently of each other. This is the best way to represent diversity in a meaningful way. On the other hand, the *hardware-oblivious* concept should allow a separation between application logic implementation and specification of the three diversity characteristics at compile- or run-time. This enables the highest degree of flexibility in mapping of application logic to a specific SIMD hardware.

3.2.2 Realization for the Query Processing Domain

To fulfill the mentioned requirements, we developed the Template Vector Library (*TVL*), an extensible header-only library tailored for data access and analysis. The *TVL* follows a separation of concerns concept. That means, it offers hardware-oblivious, but column-store specific primitives as generic functions. This explicitly enables database systems programmers to implement each query operator in a vectorized, but hardware-independent fashion, on the one hand. On the other hand, the *TVL* is also responsible for mapping the provided hardware-oblivious primitives to different SIMD hardware. For this mapping, our *TVL* includes a plug-in concept, where each plug-in has to implement each provided hardware-oblivious primitive for a specific SIMD hardware in a hardware-conscious manner.

Hence, the *TVL* is extensible in two ways: (1) Functionality can be added by introducing new primitives, and (2) Hardware support for another architecture can be added by implementing the hardware-conscious primitives for this architecture. Existing code, which was written using the *TVL*, will then be able to compile and run on the added architecture without changing anything in the source code but one template parameter.

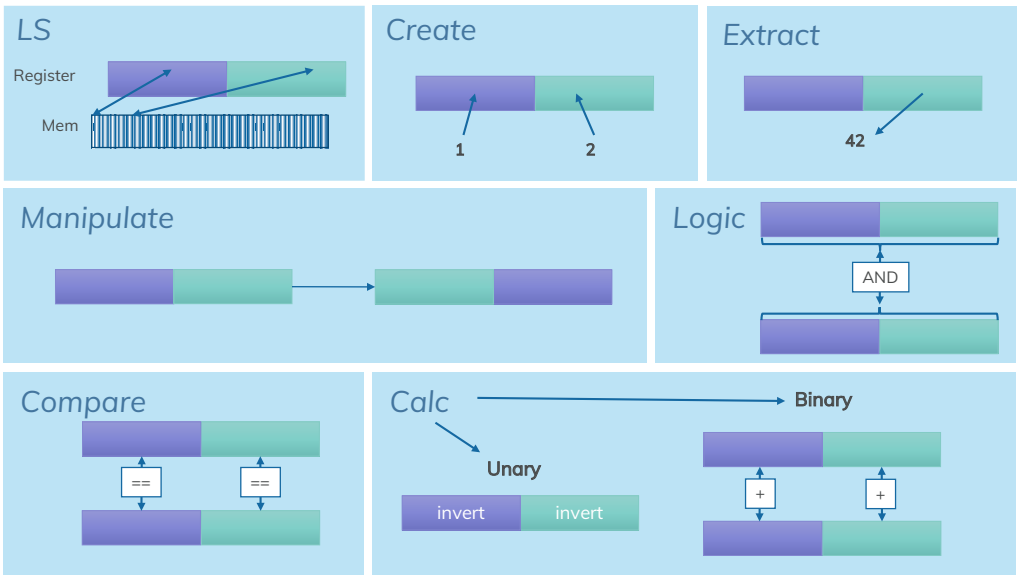


Figure 3.16: Illustration of the primitive classes of the *TVL*. The registers in this illustration contain two vector elements (green and purple).

Hardware-Oblivious Primitives

To enable a *hardware-oblivious* approach without sacrificing the performance, the *TVL* primitives abstract from SIMD intrinsics in such a way that there is the same interface for all SIMD architectures. While scalar or plain mathematical computations are always a combination of load/store operations, comparisons, arithmetic calculations, and boolean logic, vectorized query processing in in-memory column-stores requires additional functionality, e.g. permutation of vector elements as discussed in the literature [P+15, P+19b, L+15, D+17, K+18, M+17]. Based on this observation, we define seven different classes of *TVL* primitives as illustrated in Figure 3.15. In the following, the characteristics of these classes are explained. Additionally, Figure 3.16 illustrates the differences between them, i.e. the number of involved vector registers, the origin and destination of their elements (main memory, literals/scalar register, vector register), and their function (e.g. loading, storing, arithmetic, comparison).

Load/Store Class: The *Load/Store Class* contains different load and store primitives. The most obvious members of this class are sequential load and store primitives. Random memory access is realized by gather and scatter primitives. A special primitive for selectively storing the elements of a vector is called *compressstore*. A *compressstore* takes a bitmask, a vector, and a pointer as arguments. Then, it stores all elements of the vector, which have a corresponding set bit in the bitmask, consecutively into memory without gaps for non-selected elements. The widely supported masked store keeps those gaps, which is not the desired output of most database operators. Furthermore, data can be aligned or unaligned in memory, and there are instruction sets, which explicitly support streaming, also known as non-temporal memory access. To differentiate between these cases, a template parameter is used, which can have the values `ALIGNED`, `STREAM`, and `UNALIGNED` for alternative implementations. Load and store primitives are used in virtually every operator.

Arithmetic Class: The *Arithmetic Class* provides different unary and binary function primitives, which are mainly used in aggregations. Currently, the unary function primitives contain the aggregation of all elements of a vector by summing them up, the change

```
template<...>
   void
   compressstore(
      typename base_t * dataPtr,
      typename vector_t vec,
      typename mask_t mask
   );
```

Figure 3.17: Compressstore template declaration in C++.

of the sign of a number, and shifting the elements of a vector by a fixed distance. Further, there are unary functions working on masks, e.g. counting the leading zeros or the number of set bits. The currently provided binary function primitives are the four basic arithmetical operations, modulo, and shift operations, where each element of a vector is shifted by an individual number of bits. The result returned by arithmetic primitives is always a single vector or scalar, regardless of the input. The granularity of bit-level parallelism of the operations is again provided by a template parameter.

Comparison Class: This class provides element-wise comparisons between vectors, e.g. for equality or greater/less than. These primitives are required by a number of operators, i.e. intersect, different joins, and selections. The input parameters are the same as for the binary arithmetic primitives, but the output is a bitmask instead of a vector. This is especially useful, when the result of a comparison is stored. Instead of storing the bitmask, the corresponding values can be stored by using the compressstore of the *Load/Store Class*.

Bitwise Logic Class: In this class, boolean algebra is treated. Currently, a bitwise AND and a bitwise OR are provided. The necessity for this primitive class is not straight forward. It can be used for the comparison of masks, e.g. in range queries, or for a number of compression techniques.

Create Class: Sometimes, the elements of a vector are not loaded from memory, but computed at runtime, or loaded from immediate values or constants. These cases are treated by the *Create Class*. This class poses the challenge, that setting the elements of a vector to different values is an operation whose interface depends on the number of elements of a vector, i.e., depending on the number of elements per vector, the number of input parameters varies. This is incompatible with the concept of code which is portable to different vector lengths. For this reason, we provide two additional primitives in this class: *set1* and *set_sequence*. The first one sets all elements of a vector to the same value. The second one fills a vector with a sequence of numbers, where the first number and the distance between the following numbers are provided as parameters. This is especially useful for the initial creation of record indexes.

Extract Class: Whenever it is necessary to get a single element out of a vector, the `extract` primitive is used. This is the only member of the *Extract Class*. The same effect can be achieved by using a *compressstore* (Load/Store Class) with a single bit set in the bitmask, but `extract` avoids the round trip to the main memory, if a corresponding instruction is available on the targeted architecture.

Manipulate Class: The last class is the *Manipulate Class*, which takes care of vector manipulations on the element-level, i.e., permutations of the vector elements. A special permutation is the rotation. The rotate primitive rotates the elements in a vector by one element. If there are only two elements in a vector, e.g. two *double* values in a 128-bit register, this results in swapping these two elements.

These classes and primitives are specific for an efficient vectorized query processing in in-memory column stores. An example *TVL* primitive from the **L/S Class**, which is

used in selective database operators, is the *compressstore* function template as depicted in Figure 3.17. This specific function takes a pointer (`dataPtr`) to a memory location, a vector (`vec`), and a bitmask (`mask`) as input parameters. Then, the task is to store the vector register (`vec`) in such a way that the vector elements with a corresponding set bit in the bitmask are stored consecutively to memory. How this function is realized is not the subject of this template declaration, but of the hardware-conscious specialization.

Additionally to the primitives, we introduce generic datatypes:

- **base_t:** The base type can be any scalar type. There are hardware-conscious backends for different integer types, and partially for floating point and double types.
- **vector_t:** The vector type contains one or more values of the same base type.
- **mask_t:** A mask is a scalar value, which is large enough to store one bit for each element in a vector.

Depending on these types, there are also automatically derived parameters, e.g. the number of elements per vector.

Hardware-Conscious Specialization

From a template metaprogramming perspective, our *TVL* primitives are generic interfaces to implement in a vectorized way, but for the execution, we require a hardware-conscious function template specialization for the underlying SIMD hardware. This function template specialization has to be implemented, whereby the implementation depends on the available functionality of the SIMD hardware. However, this is independent of the query operators and must be done *only once* by a domain expert for a specific SIMD hardware.

In the best case, we can directly map a *TVL* primitive to a SIMD intrinsic. However, if the necessary SIMD intrinsic is not available, we are able to implement a work-around in a hardware-conscious way. To illustrate this specialization, Figure 3.4 shows the implementation of the *compressstore* primitive for three different SIMD instruction sets assuming a 64-bit base data type.[4] To the best of our knowledge, Intel's AVX-512 is the only instruction set, which contains an intrinsic doing exactly what a *compressstore* is meant to do, and we map directly to this intrinsic. For architectures without AVX-512, a work-around has to be implemented. In this work-around, we have to treat all possible values of the bitmask manually. If no bit in the bitmask is set, the function returns without storing anything. If all bits are set, the whole register is stored. If only a subset of bits is set, there are different ways to store the corresponding vector elements. In NEON, there is an intrinsic to store selected vector elements. In SSE, we could either use an intrinsic to extract values into a scalar register and then write them to memory, or shuffle the according elements to the beginning of the register before writing it to memory. We decided for the latter because of a higher compatibility with different base data types.

To eliminate the overhead of a function call when using primitives, we inlined all primitives with `inline __attribute__((always_inline))`. This ensures that the overhead over using intrinsics is negligible as we show in Section 3.2.4. A nice side effect of our overall concept is that we are also able to map to a scalar specialization. In this case, the vector length can be 8-, 16-, 32-, or 64-bit, and we map our *TVL* primitives to the corresponding scalar instructions.

[4] Note that, for the sake of simplicity, the function headers are not shown.

```
//let outPtr, mask, and vec be local variables
//a) Using AVX-512 intrinsics directly, no TVL
_mm512_mask_compressstoreu_epi64(outPtr, mask, vec);
//b) Naive implementation using TVL
tvl::compressstore < avx512<v512<int64_t>> , tvl::UNALIGNED, 64>(outPtr,
    vec, mask);
//c) Fully portable implementation using TVL
using processingStyle = avx512<v512<int64_t>>;
tvl::compressstore <processingStyle, tvl::UNALIGNED,
    processingStyle::base_t_size_bit>(outPtr, vec, mask);
```

Figure 3.18: Example usage of the *TVL*.

Interplay

To connect our hardware-oblivious primitives with the different hardware-conscious specializations during query compile-time, we decided to use three nested template parameters and call a combination of these parameters a *processing style*. The template parameters are derived from the description of the SIMD variety in Section 3.2.1: (1) the vector extension (e.g. SSE, AVX, NEON, or scalar), (2) the vector size in bit, and (3) the base data type with bit granularity (e.g. int8, int64, float). An additional optional template parameter is the bit-granularity of a primitive. This way scenarios can be handled, where an operation is not performed on the granularity of the base type, e.g. loading 32 bit values into 64 bit registers to handle potential overflows during later processing. The *Bitwise Logic* class does not involve a granularity parameter for the obvious reason that the comparisons are always executed on a bit granularity. Furthermore, the primitives in the Load/Store class require the already mentioned template parameter indicating the alignment in memory. This enables us to define the exact mapping in a very fine-grained and flexible way. That means, each primitive within a query operator can be called with its own processing style. An example of how exactly the processing styles are used, is shown in Figure 3.18. In a), the intrinsics provided by AVX-512 are used directly. This code will only work on 512-bit registers on Intel architectures providing this instruction. Snippet b) uses the *TVL* in a very naïve way. The primitive *compressstore* is called with a processing style, that maps to AVX-512. The second template parameter indicates that the data does not need to be aligned. The mapping to the hardware-conscious implementation is also highlighted in Figure 3.15. To make the code fully portable to other SIMD architectures, the derived constants, such as `base_t_size_bit`, can be used, which are also provided by our *TVL* as shown in snippet c) in Figure 3.18.

3.2.3 Integration of new Instructions into the Template Vector Library

The integration of special instructions into the *TVL* is straight forward because of the extensible design of the *TVL*. Most operations only require to add the interface of the according primitive to the respective class and implement this primitive with the special instruction for all desired target processing styles. For instance, the horizontal aggregation is a unary arithmetic calculation, where *unary* refers to a single input vector instead of a single input value. Permutations change the vector register by manipulating the position of the elements. Hence, the permutation fits into the manipulate class. However, in some cases, more adjustments have to be done:

```
1  template<class ProcessingStyle, int Granularity>
2  struct equal {
3      //nice macro for an ugly compiler directive
4      MSV_CXX_ATTRIBUTE_FORCE_INLINE
5      static
6      typename ProcessingStyle::mask_t
7      apply(
8          typename ProcessingStyle::vector_t const & p_vec1,
9          typename ProcessingStyle::vector_t const & p_vec2
10         //Default this to a deleted function to get compiler errors instead
11         //of linker errors if this primitive is not supported for a target
12     ) = delete;
13 };
14
15 template<class ProcessingStyle, int Granularity>
16 struct conflict_detection {
17     MSV_CXX_ATTRIBUTE_FORCE_INLINE
18     static
19     typename ProcessingStyle::vector_t
20     apply(
21         typename ProcessingStyle::vector_t const & p_vec1,
22
23     ) = delete;
24 };
```

Figure 3.19: Interfaces of a comparison for equality between two different vector registers (l. 1-13) and between the elements in one vector register (*conflict detection*, l. 15-24). The lines, which differ between these two interfaces, are highlighted.

New (sub)classes The existing classes and subclasses may not be sufficient for new primitives, either because the interface does not fit or because there is no intuitive mapping to any of the existing classes. For instance, detecting conflicts within the same register, is a comparison but with only one input vector. Additionally, the *conflict detection* does not return a mask, but a vector, which contains a collection of masks. This introduces a new primitive sub-class, the horizontal comparison. Figure 3.19 shows the interface of the *conflict detection* (line 15-24) compared to the interface of an equality check between two input vectors (line 1-13). The different return types and the different amount of input vectors is highlighted.

Additional template parameters There are primitives, where the existing template parameters are not sufficient to distinct between all different primitive implementations. This is especially obvious in the *gather* primitive. As already explained in chapter 3.1.1, this primitive becomes tricky for small base data sizes. Besides the sheer performance loss, there is a second issue, which concerns the size of the offset vector. Typically, the offset has the same size as the data to be gathered. Hence, if the base data type is only 32 bit, the offset is also only 32 bit. But this way, not the whole dataset might be addressable. This has been solved to a certain degree in AVX2 by introducing intrinsics for 32-bit gathers, which use 64-bit indices. However, the distinction between the different offset sizes must be made independently of the base data size. For this reason, the *gather* primitive of the *TVL* requires a second granularity template argument with the size of the offsets. An overview on the available primitives of each class with their template parameters is displayed in Table 3.1.

Class	Primitives	Template Parameters	Parameters	Returns
L/S	load, store, gather, compressstore	Processing Style, IO-Variant, Granularity, Size offset in byte (gather)	Memory Address, vector_t (store), mask_t (compressstore)	void (store), vector_t (load)
Arithmetic	add, sub, div, mul, min, max, count_matches, count_leading_zero, shift_left(_individual), shift_right(_individual), inv (unary), hadd (unary)	Processing Style, Granularity	vector_t (unary), 2x vector_t (binary), mask_t (count)	vector_t, short (count)
Comparison	equal, less, lessequal, greater, greaterequal, conflict_detection (horizontal)	Processing Style, Granularity	2x vector_t, vector_t (horizontal)	mask_t, vector_t (horizontal)
Bitwise Logic	bitwise_and, bitwise_or	Processing Style	2x vector_t	vector_t
Create	set_sequence, set1	Processing Style, Granularity	base_t, step size (set_sequence)	vector_t
Extract	extract	Processing Style, Granularity	vector_t, Index	base_t
Manipulate	rotate	Processing Style, Granularity	vector_t	vector_t

Table 3.1: *TVL* overview of the currently available primitives and their interfaces

Once the new primitives are implemented for all target processing styles, alternative physical operators using these primitives, can be implemented. This can involve work-arounds for those architectures, which do not support a certain functionality. We already showed the *compress store* as an example in chapter 3.2.2 and *gather* as an example in chapter 3.1.1. Nevertheless, it is not always beneficial to provide a primitive for every processing style. This is the case when a vectorized work-around has no benefit over the scalar version. For instance, a work-around for the *conflict detection* would require to extract all elements from a vector register and do a scalar comparison with all previously extracted elements, just to write those results back into a vector register. This is basically a scalar comparison with the added overhead of extracting and setting vector elements. Hence, a scalar comparison without this overhead is the more efficient solution. This nonexistence of a primitive for a certain processing style poses a challenge for the user, who will be presented with cryptic linker errors when compiling a primitive with an unsupported processing style. For this reason, all primitive prototypes default to a deleted function as shown in Figure 3.19 in line 12 and 23. This way, the error is thrown during compile time, i.e. before linking, which results in the well understandable error message "file, line x:y: use of deleted function".

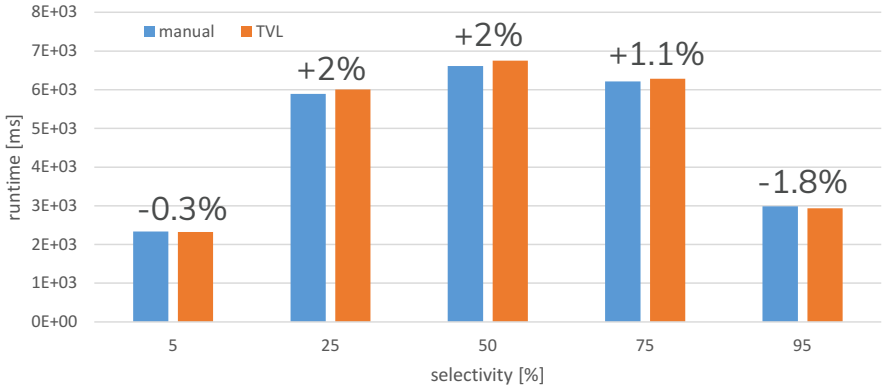


Figure 3.20: Runtime overhead for column scan. The dataset contained 10^6 values, which does not fit into L2 cache anymore.

3.2.4 Evaluation

We used the *TVL* to realize different columnar query operators. For this evaluation, we focused on the different Intel instruction sets. Therefore, we implemented template specializations, i.e. *TVL*-backends, for scalar processing and different Intel SIMD extensions. In particular these extensions are SSE, AVX2, AVX-512. Additionally, we realized some hand-vectorized operators to measure the potential runtime overhead introduced by the *TVL*.

Runtime Overhead Consideration

In a first series of experiments, we compared the runtimes of hand-written query operator code using SIMD intrinsics with runtimes of TVL-enabled query operators. To keep the results comparable, both variants use the functionalities offered by Intel AVX2. For all these microbenchmarks, the complete processed data fit at least into the L3-cache of the Xeon Gold 5120 (Skylake, max. core frequency: 3.2 GHz) and the benchmarks ran single-threaded. All micro-benchmarks were executed 900 times to minimize the impact of time measurement. Figure 3.20 exemplary shows the different average runtimes for a column scan with different range predicate selectivities. We used these selectivities to vary the runtime behavior. The selectivities are plotted on the x-axis while the y-axis shows the complete runtime of the operator in milliseconds (lower is better). On the one hand, the variant using TVL performs slightly better than the variant using SIMD intrinsics for selectitivies of 5% and 95%, respectively. On the other hand, for selectivities of 25% and 50%, the operator variants behave inversely, resulting in an average runtime overhead for our TVL approach of around 1.02% for this operator. For the other query operators, we observed a similar overhead resulting in a negligible overhead. Hence, we conclude that our TVL offers high flexibility at virtually no performance cost.

Comparison with Autovectorizer

For a second series of experiments, we compare the runtime of code using the TVL to the runtime of auto-vectorized code. We will further show some limits of vectorization in general. For these experiments, we consider different memory access patterns, because the ability of compilers to vectorize code tends to vary depending on the memory access pattern:

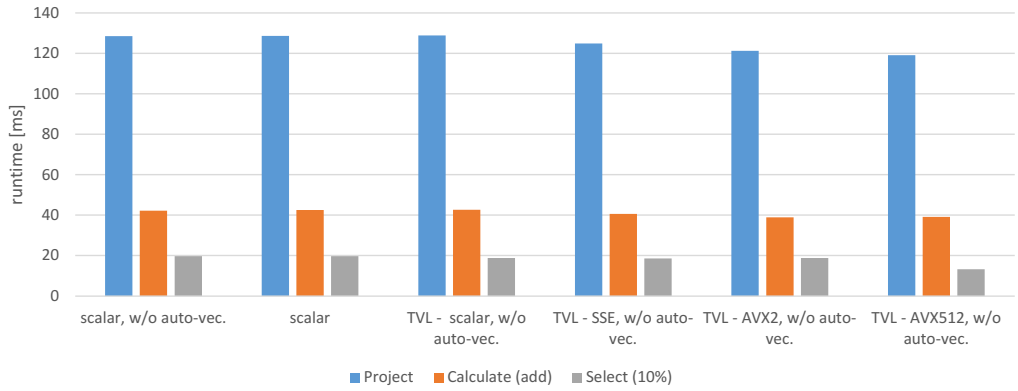


Figure 3.21: Comparison of scalar, autovectorized scalar, and *TVL* vectorized code for different operators. The dataset contained 10^7 values, which does not fit into L3 cache anymore.

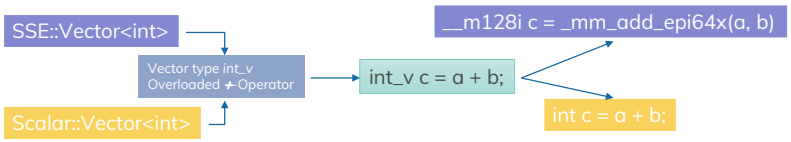
1. A projection includes sequential and random memory read access as well as sequential memory write access.
2. A calculation, taking two columns as an input, doing an element-wise aggregation, and returning another column, addresses only sequential read and write access.
3. Finally, a selection with a selectivity of 10%, incorporates sequential read and write access, where the read to write ratio is only 1 to 10.

The system, we used for these experiments, was an Intel Xeon Gold 6130 (Skylake, max. core frequency: 3.7 GHz). The compiler was g++ (version 8.3). All experiments use 10^7 values per column, where each value is a 64-bit integer. Thus, they do not fit into cache anymore. Figure 3.21 shows the runtime of the mentioned operators. The first group of bars shows the hand-implemented scalar code with disabled auto-vectorization.[5] The second group shows this code with enabled auto-vectorization. The remaining bars are the runtimes of the *TVL* enabled operators for different *processing styles*. We chose the native vector size for each SIMD extension.

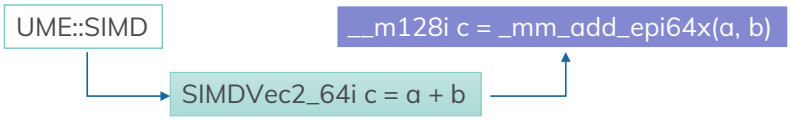
The projection shows no significant runtime difference for all scalar versions and only a small speed-up for the *TVL*-vectorized versions. The relatively stable performance is caused by the heavily bandwidth bound characteristics of this workload. Further, auto-vectorization fails with the message that it is not profitable. In contrast to that, the auto-vectorization of the calculation is successful. However, it decided to use 256-bit vectors, where 512-bit vectors are also available. Additionally, the auto-vectorized code is not faster than the scalar code, but the *TVL*-vectorized code is, even only by a small amount. The selection has a lower memory write access rate than the other operators, which is the most expensive operation in all of the chosen examples. Hence, some optimization potential can be expected from optimizing the steps before accessing the memory to write the result. Such an optimization could be the efficient extraction of the selected values from a vector register. Unfortunately, the selection also fails to auto-vectorize. Where the *TVL*-enabled code uses a compress store, the auto-vectorizer tries to implement a masked scatter but fails due to "bad data references". Hence, the shortest runtime is retrieved by using the TVL with AVX512, which translates directly to the compress store intrinsic. The other *TVL* variants show no mentionable performance gain, because they cannot use a hardware-backed intrinsic, but must rely on workarounds.

[5] We used gcc version 8.3 for compiling

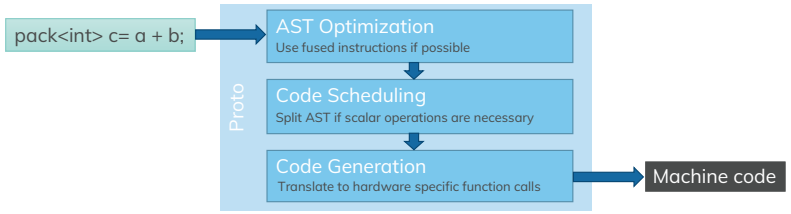
(a) Sierra



(b) Vc



(c) UME::SIMD



(d) Boost.SIMD

Figure 3.22: An addition using different vector libraries. (a) The *Sierra* compiler (a clang fork) translates the code into vectorized LLVM code, which is then compiled and linked into an executable. (b) *VC* overwrites operators depending on the chosen namespace (c) *UME::SIMD* encodes the vector- and element-size in the vector type (d) *Boost.SIMD* introduces an embedded DSL for vectorization. Boost.Proto, a compiler construction toolkit for EDSLs is used to optimize, schedule, and translate the AST.

3.2.5 Summary

We introduced the Template Vector Library (*TVL*), which efficiently hides the heterogeneity of instruction sets and vector register sizes from the application developer while enforcing explicit code vectorization. The *TVL* allows porting code across all ISAs, which offer a corresponding back-end. Instead of intrinsics and ISA specific data types, the TVL offers generic vector and mask data types, and primitives, which are templated functions. We identified seven fundamentally different primitive classes, which are necessary to implement query operators, and developed the according back-ends for the most common Intel® and ARM® instruction sets. Finally, our evaluation empirically shows, that our hardware-oblivious operators with the TVL can efficiently run on different SIMD extensions with virtually no performance overhead. Additionally, we showed that using explicit vectorization over an auto-vectorizer does not have a negative effect on performance. Moreover, a positive effect can be observed in some cases.

3.3 RELATED WORK

There is a plethora of approaches for hardware-conscious vectorization for in-memory column-stores as presented in Chapter 2, e.g. [P+15, P+19b, L+15, D+17, K+18, M+17]. However, these solutions assume a certain instruction set on the target hardware. This limits their portability significantly and erases the option to combine different instruction sets. For this reason, hardware-oblivious approaches have been developed. There are hardware-oblivious concepts designed specifically for in-memory column-stores, and there are concepts, which do not target a specific application scenario. In the following, we present examples for both cases.

In-Memory Column-Stores

In addition to hardware-conscious vectorization, there are also some concepts available towards hardware-oblivious support for in-memory column-stores. For example, Heimel et al. [H+13] presented a hardware-oblivious extension for MonetDB by using the parallel programming framework OpenCL. As they have shown, they can map single-source query operators implemented in OpenCL to different parallel processing architectures like multi-core CPUs or GPUs. Another hardware-oblivious approach in this direction is Voodoo, which is a declarative intermediate algebra abstracting the detailed architectural properties [P+16]. The Voodoo compiler also produces OpenCL code to support multi-core CPUs and GPUs. Moreover, Voodoo includes a small set of vector operations like scatter. However, the main bottleneck of both approaches is OpenCL from a vectorization performance perspective. This is because OpenCL-based solutions focus on multi-core scenarios rather than on single-thread performance. Behrens et al. [B+18] have clearly shown that, e.g. vectorized hashing based on SIMD intrinsics outperforms OpenCL-based hashing. In contrast to that, our *TVL* approach achieves similar performance compared to SIMD intrinsics.

General Concepts for Hardware-Oblivious Vectorization

There are already approaches for vector libraries, which avoid additional layers like OpenCL and reduce the additional overhead to a minimum, although none of them was designed for the use in in-memory column-stores, or for any data driven scenario in general. Two examples for such libraries are *VC* [KL12b] and Sierra [L+14]. Both approaches automatically translate a generic vector type to the largest available vector size and overload standard operators to work with this vector type. Note that the largest available vector size in *VC* depends on the chosen namespace. This reduces the translation of the written code to exactly one combination of vector size and instruction set, even if the system offers more, and potentially faster, variants. Additionally, very specialized functions, e.g. a stream store, which can enhance the performance in some cases but have no equivalent standard operator, cannot be used. Hence, *VC* and *Sierra* work well in plain mathematical scenarios, but the application in our domain is limited. There also used to be a SIMD template library, which was supposed to become a part of the boost library [EFGL14]. This library was realized as an embedded DSL (EDSL). This allowed for optimization and efficient scheduling of the AST by applying Boost.Proto, a compiler construction toolkit for EDSLs. Nevertheless, it did not make it into a final release of Boost. Metacale, the company driving the development of *Boost.SIMD*, has been renamed into Numscale and later became a part of the Agenium Group. Agenium's *Boost.SIMD* github repository is empty and refers to a successor project called *nsimd*, which is still open source

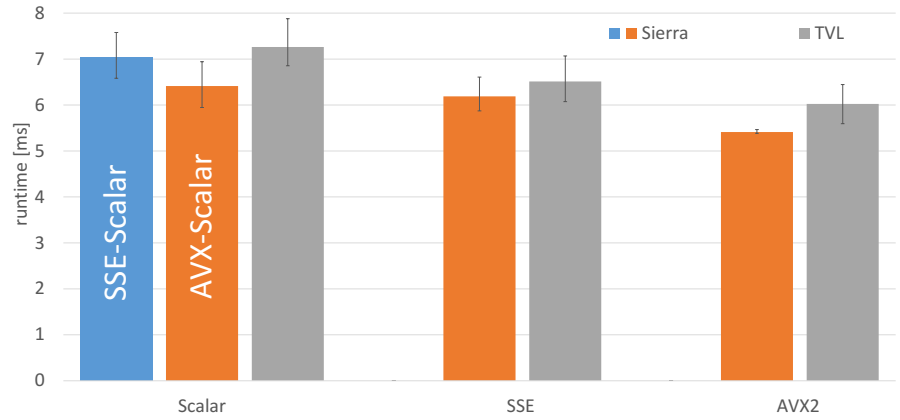


Figure 3.23: Runtime comparison of an aggregation between Sierra (blue and gray) and the TVL (orange). The dataset contained 10^7 integers.

but a license fee applies for updates and support[6]. Additionally, there is currently no publication explaining how much of the original idea of *Boost.SIMD* is left in *nsimd*. Another API, which aims to provide a portable SIMD abstraction is *UME::SIMD* [KM17]. It supports a wider function range than overloaded operators, e.g. permutations and horizontal aggregations. Additionally, *UME::SIMD* leaves the choice of the vector element size and the vector element count to the user by encoding both of these quantities into the typename of the vector. This enables the use of the actually fastest variant, but this variant might be a different one depending on the hardware and the instruction set it is mapped to. To change the code to another variant, all type mentions must be changed, which requires some amount of refactoring. However, it is possible to realize an aggregation with all of these libraries, because this only requires a basic arithmetic addition. Figure 3.22 illustrates the realization of an addition with the mentioned libraries. Since Sierra is available on github and it is documented including example files, we chose to implement an aggregation using Sierra and compare the runtime with the *TVL*-enabled aggregation.

Comparison with Sierra We compared the runtime performance of the *TVL* to *Sierra* [L+14]. Since the functionality of Sierra is limited, we did not implement any selective operator. Instead, we implemented an aggregation, which is trivially vectorizable. To not measure the efficiency of an auto-vectorizer, but of *Sierra*'s SIMD mode and the *TVL* primitives respectively, we turned off auto-vectorization. To compile the implementation for *Sierra*, we used the sierra fork of clang (version 3.3). For the *TVL*, we used the regular clang compiler (version 7). We chose to not compile the *TVL* code with clang version 3.3 because of some language features required by the *TVL*, which were introduced later. All experiments were run single-threaded on an Intel Xeon Gold 6130 CPU (max. core frequency: 3.7 GHz). Fig. 3.23 shows the runtimes for SSE, AVX, and scalar processing. In addition to the median of 10 runs, the range of the runtimes is shown. In *Sierra*, there are two different versions of scalar processing. The first one, *SSE-Scalar*, is compiled for SSE, but with a vector length of 1, i.e., every vector contains only one element but the compiler is allowed to use SSE functionality. The second one, *AVX-Scalar*, also has a vector length of 1, but is compiled for AVX. For the *TVL*, we used scalar, SSE, and AVX2 processing styles with the native vector lengths of the corresponding instruction set, e.g. 128-bit registers for SSE. The graph shows that in this use-case, the size of

[6] https://github.com/agenium-scale/boost.simd and https://store.agenium-scale.com, accessed 09 March 2020

registers does not scale proportionally with the performance. This is because the workload mainly consists of sequential memory read access, which is bandwidth limited. The ranges of the runtimes for *Sierra* and the *TVL* are overlapping in all cases except for AVX2. However, *Sierra* performs slightly better than the *TVL*. We suppose that this is because of the highly SIMD-optimized *Sierra* compiler. A six times lower cache-reference number and 30% more instructions per CPU cycle, with slightly slower CPU cycles, for the *Sierra* implementation supports this assumption. Considering, that *Sierra* puts tight limits on the usable functionality of any vector extension, especially when it comes to selective operations, it is not applicable for a database system despite the small performance gain compared to the *TVL*.

3.4 SUMMARY

In this chapter, we addressed the challenges of the growing size of vector registers in recent SIMD instruction sets, especially AVX512. In particular, there are inherent challenges introduced by the sheer amount of elements in a vector, and there are challenges arising from the vectorized algorithms, which are not always fit for large registers. Some recent instruction sets offer specialized functionality, which helps to solve these challenges. However, the different function range, vector sizes, data types, and intrinsics, require another implementation for each target architecture, which is not necessarily done by simple refactoring. To overcome this issue, we introduced the *TVL*, a library for hardware-oblivious explicit vectorization, tailored for the use in query processing. We implemented a set of operators necessary to run analytical queries and showed that there is virtually no overhead to a hand-vectorized implementation. Finally, we discussed related work and compared our approach against the highly optimized but functionally limited *Sierra* compiler. We found that our approach has a small but negligible overhead.

4

BALANCING PERFORMANCE AND ENERGY FOR VECTORIZED QUERY PROCESSING

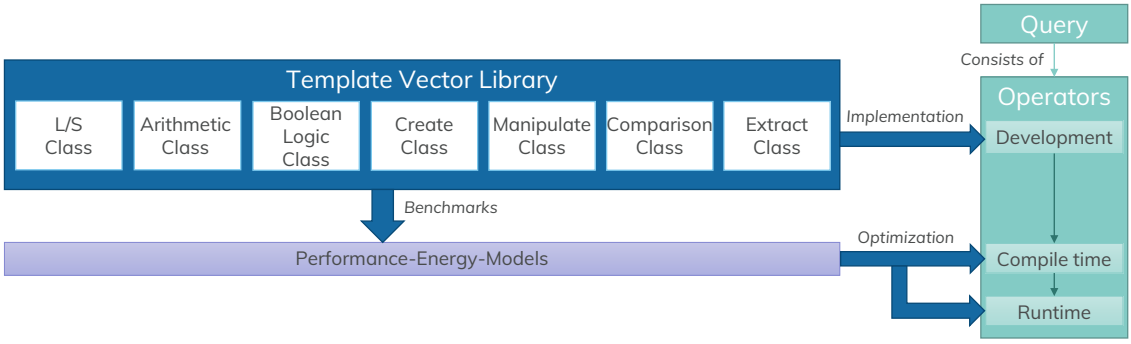


Figure 4.1: In the last chapter we introduced the Template Vector Library (*TVL*). In this chapter we will use the *TVL* as a basis to optimize queries.

In the last chapter, we introduced the Template Vector Library (*TVL*), which enables an easy selection of the SIMD instruction set to achieve optimal performance. As we have shown in Section 2.3.1, there is a significant performance optimization potential when the instruction set is chosen per operator, and not per query. However, as we have also shown, this choice is not trivial and different queries benefit from a different choice. Moreover, in Section 2.3.2 we explain that the use of SIMD puts tighter limits to other optimization knobs, e.g. frequency scaling. This also affects the energy-efficiency, which has become another important optimization goal. For instance, the achievable CPU frequency, which is a main factor for the energy consumption of CPUs, is influenced by the chosen instruction set and the number of active CPU cores. Hence, performance and energy-efficiency are related optimization goals influencing each other. To optimize for both of these two goals, a mapping between performance and energy-efficiency is necessary. In this chapter, we propose a model, called *Work-Energy-Profiles* (WEPs), which shows this mapping. We create such *WEPs* for the primitives of the classes in our *TVL* and some frequently used primitive combinations. Then, we use these *WEPs* to create models for more complex operators, which were implemented with the *TVL*. These models can then be used to select the optimal instruction set and cpu frequency on an operator level. Figure 4.1 shows a rough outline of this approach. The upper part of the figure, i.e. the *TVL* and the implementation using it, was covered in the last chapter. The lower part, i.e. the optimization using *WEP*s, is the subject of this chapter[1]. In particular, we the following topics are discussed:

1. We show why benchmarks are necessary to create reliable models in Section 4.1, i.e. we explain why a completely analytic model is not feasible.
2. In Section 4.2, we explain our *Work-Energy-Profiles*. This includes the general idea, the characteristics of our test systems, and example profiles for these test systems.
3. Since benchmarking is time-consuming, we present a method for creating *WEPs* for all applications, which rely on the *TVL*, by combining the *WEPs* of the primitives (Section 4.3).
4. We show how to apply *WEP*s in order to optimize a workload or a query in Section 4.4.
5. Finally, in Section 4.5 we focus on previous approaches of optimization for energy-efficiency in a discussion of related work.

[1]Parts of the material in this chapter have been developed jointly with Alexander Krause, Patrick Damme, Johannes Pietrzyk, Thomas Kissinger, Willi-Wolfram Mentzel, Dirk Habich, and Wolfgang Lehner. The chapter is based on [UKHL16, UKM+16, UDP+17]. The copyright of [UKHL16] and [UDP+17] is held by Springer International Publishing AG; the original publications are available at https://doi.org/10.1007/978-3-319-54334-5_10 and https://doi.org/10.1007/978-3-319-67162-8_5. The copyright of [UKM+16] is held by the authors; the original publication is available at https://doi.org/10.1145/2882903.2899390.

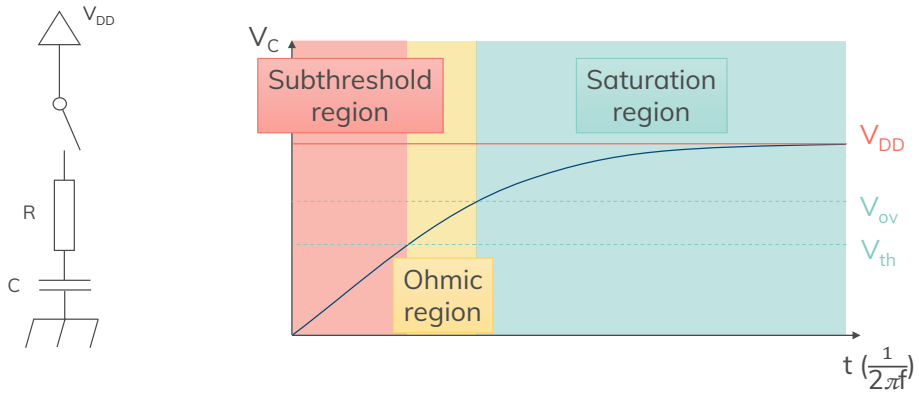


Figure 4.2: Circuit for the input capacitance of a MOSFET including the lead resistance (left) and the characteristic curve (right) of the capacitance. Additionally, the threshold and overdrive voltages of the MOSFET are illustrated. In the *subthreshold region*, there is no current at the drain, the transistor is switched off. In the *ohmic region*, there is a current passing the capacitor, but gate switching is not reliable. Transistors in CPUs work in the *saturation region*. Note that real characteristic curves are steeper, and the *subthreshold region* is narrower and the threshold and overdrive voltages are usually not shown in these diagrams. This figure aims to explain the sheer concept.

4.1 WHY WE NEED BENCHMARKS

The exact performance and energy consumption of a CPU are not trivially predictable, and they are not computable in an acceptable amount of time. To understand the reasons for this claim, a short excursion to the characteristics of transistors is useful. There are different kinds of transistors. In modern CPUs, mostly MOSFETs (metal–oxide–semiconductor field-effect transistors) are used. The transistors consist at least of a source, a drain, a gate between the source and the drain, a body, and an isolator separating the gate from the body. The name MOSFET was kept, even when the gate material changed from metal to silicon. Simply put, a transistor works like a resistor, which is controlled by voltage. By changing the voltage between the gate and the source, the resistance between the drain and the source is changed. This causes a change in the current. A high and a low current level equal a 1 or a 0. In the equivalent circuit diagram[2] of a transistor for the frequencies relevant in CPUs, this is modeled by a resistance and three capacitances. For the sake of simplicity, we show only the resistance and a single capacitance in Figure 4.2. The supply voltage is denoted as V_{DD}. The diagram in Figure 4.2 shows the characteristic curve of the capacitance, i.e. the voltage as a function of the time. Only if the voltage between the gate and the source surpasses a threshold V_{TH}, there is a channel between the source and the drain, i.e. there is a current. If the voltage increases, the current also increases. The region below this threshold is called the *subthreshold region*, in which the transistor is switched off. The region above this threshold is called the *linear region* or *ohmic region*. In this region, the drain current grows approximately proportional to the drain-source-voltage. This can be used to amplify incoming signals. If the voltage passes a second limit, the overdrive voltage V_{OV}, the current is not changed significantly anymore. Instead, the supply voltage can be used directly to change the current. There is a clear distinction between the current levels when the switch is open versus when the switch is closed. This region is called the *saturation region* or *overdrive region*. CPUs work

[2] An equivalent circuit diagrams do not show existing circuits, but the behavior of an electric component. For this reason, we are using the terms *resistance* and *capacitance* over *resistor* and *capacitor*.

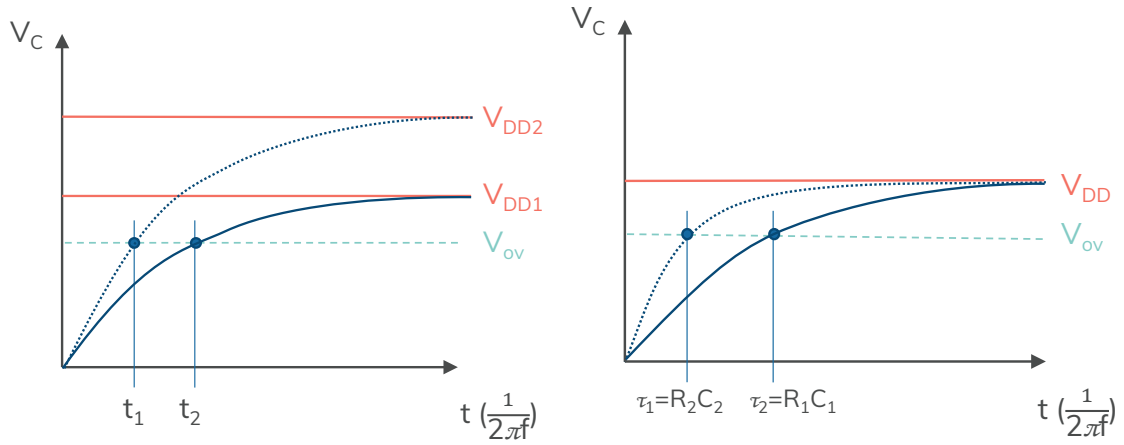


Figure 4.3: There are two ways of changing the physically possible gate switching time: (1) The supply voltage is increased and (2) the resistance and/or capacitance is decreased. The latter are constants, which cannot be changed after the transistor has been built, because they depend on the hardware design.

in this *saturation region*, because transistors working in the *linear region* do not reliably map the supply voltage to distinct 0 or 1 values[3].

This means, the minimal length of a cycle (x-axis), and therefore the maximum switching frequency, is determined by the intersection of the characteristic curve of the capacitance with the overdrive voltage. To increase the maximum frequency, there are two different ways: (1) Increase the supply voltage, and (2) Change the shape of the curve. Both of these possibilities are pictured in Figure 4.3. The left side shows option (1). As shown, the curve becomes steeper when the supply voltage is increased from V_{DD1} to V_{DD2}. This results in a smaller cycle length at t_1 instead of t_2. Option (2) is shown on the right side. The shape of the curve depends on the resistance R and the capacitance C. If R or C changes, the curve changes, too.

However, both options have limits. Changing the capacitance or the resistance can only be done during hardware development, because these quantities follow from the materials and the physical design of the transistors. The supply voltage can be changed dynamically, but the switching power consumption depends quadratically on the supply voltage. Additionally, the energy conversion efficiency of a transistor is not 100%. There is leakage energy, which is emitted as heat. A high number of densely packed transistors working in the saturation region produce enough heat that the threshold and overdrive voltage are affected, which requires a higher increase of the supply voltage to increase the frequency. But again, this produces even more heat. Thus, increasing the supply voltage only makes sense to a certain degree, where this degree depends on the number and density of active transistors. This is a main reason why the maximum CPU frequency for vectorized processing is below the frequency for scalar processing, which uses fewer transistors per instruction, and why it decreases even further when using multiple cores.

Even when the supply voltage stays within these boundaries, energy consumption is not trivially predictable. For instance, an increased voltage would lead to a higher energy consumption per cycle, if the cycle length does not change. But a shorter cycle length could amortize for the higher voltage. Jejurikar et al. showed this complex relationship

[3]This statement is backed by personal experience with the Tomahawk chip, where we were able to select the supply voltage manually.

exemplarily for the Transmeta Crusoe Processor [JPG04]. They show that with an increasing supply voltage, the static energy consumption and the leakage energy per cycle decrease, while the switching energy per cycle increases. If all of these parts are added to show the overall energy consumption per cycle, there is an optimum for energy consumption, which is far below the maximum frequency.

While this is an interesting finding, it is still not sufficient to estimate the energy consumption for a given application on a modern CPU. First, the Crusoe CPU was constructed to be especially energy-efficient and simple. The compatibility with x86 CPUs is reached by emulating the instructions, not by implementing them in hardware. However, in most modern CPUs there are several instruction sets available, e.g. for vectorization, and the properties of the transistors are not something a user can find in a documentation. To estimate their energy consumption it would be necessary to know how many and which transistors are used for each instruction, and the physical chip design. As we have shown, even a single transistor is already a quite complex element. A gate can be realized with different amounts of transistors, e.g. there are implementations of the XOR gate, which use between 4 and 16 transistors. An instruction is built from an arbitrarily high number of gates. With the introduction of more instruction sets on a CPU, the variety of gate combinations increases further. Nowadays, the way these gates are placed and connected on a chip is not even known by the developer. Chip design is done for the logical layout, while the physical layout is automatically computed. On top of this, all instructions, which use data from main memory or any shared cache, introduce additional potential stalling cycles, where the occurrence of stalling cycles depends not only on the CPU, but on the memory and its usage by any other system components.

In conclusion, there are several hardware layers built on each other: transistors are connected to create gates, which are connected to create instructions, which are scheduled depending on the software and the availability of shared resources. Each of these layers has its own properties, which are usually not known by the software developer or which can only be determined during runtime. If all of these variables are known, it is possible to retrieve reliable estimations from a simulation done with the automatically generated physical design of the CPU and all other system components. However, such simulations have a longer runtime than the application on the actual hardware, and the developer would have to provide the design. Hence, this method is not practicable. For this reason, we use a benchmark-based approach, which does not require any knowledge of the chip design but provides real-world and hardware-specific insights.

4.2 WORK-ENERGY-PROFILES

Modern hardware and operating systems offer several control knobs to adjust hardware settings and accordingly influence the performance and the energy consumption. However, a mapping between the hardware configuration, performance, and energy-efficiency is not always trivial as explained above. For example, two CPU cores processing a query in scalar mode might perform as well as a single core processing the same query using SIMD, but their energy consumption differs. Further, the performance equality in this example might not exist for all applications, e.g. if it is bandwidth bound, such that enabling a second core hardly produces a performance gain. Nevertheless, the determination of a hardware configuration offering the best energy-efficiency for a desired performance is important for applications. Energy-efficiency has even been called "The new holy grail of data management systems research" in a highly respected publication by Harizopoulos et al. [HSMR09]. To capture all hardware configuration possibilities and to consider all hardly predictable effects, we propose to solve this challenge using our so called *Work-Energy-Profiles (WEPs)*.

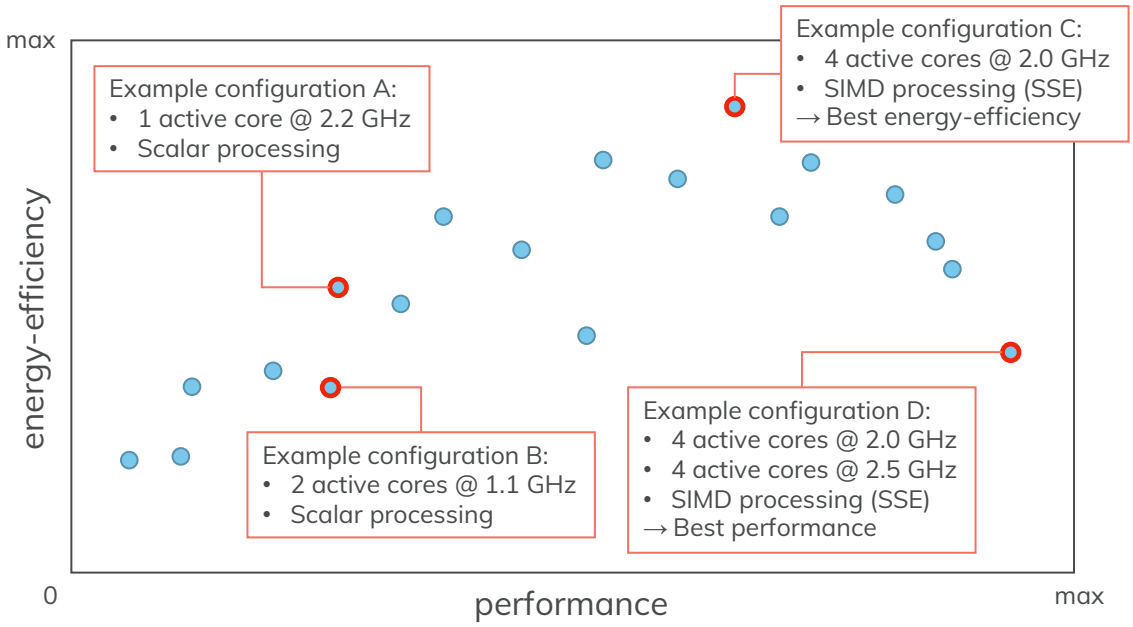


Figure 4.4: A fictional minimalist *Work-Energy-Profile* with highlighted example configurations. It shows some effects as observed in real profiles.

Basically, a *WEP* is a mapping between performance and energy-efficiency for all possible hardware configurations. The *WEPs* have to be determined for a specific application and on a concrete hardware system. Based on these *WEPs*, we are able to select an energy-efficient hardware configuration for a requested application performance range. First, we introduce our *WEPs* as a general solution. Then, we introduce the systems we use to demonstrate our approach and a benchmark concept to create *WEPs* for these systems Finally, we discuss selected example profiles.

4.2.1 A Model for Performance-Energy Mapping

Work-Energy-Profiles (WEPs) show the mapping between performance and energy-efficiency for different hardware configurations. In this context, a configuration covers all system knobs, which are adjustable independent of the implementation. Depending on the hardware and operating system, this can include different properties. The Template Vector Library (*TVL*) allows to choose the instruction set if there are more than one available. This makes the instruction set a possible part of the configuration. However, as already explained, there are more properties influencing the performance as well as energy-consumption. The following list shows the most common of these properties:

- CPU core frequency
- Number and ID of active cores, where the ID is important if there are multiple sockets or heterogeneous cores
- Number of active threads and use of multi-threading
- Instruction set and (vector) register size

Figure 4.4 illustrates the idea of *WEPs*. Performance and energy-efficiency span a two-dimensional space, and each configuration is represented by a position in this space. Even if this example is a fictional one to keep it simple, two effects are shown, which we

Number of Cores	64 (4 sockets à 16 cores)
Hyperthreads per core	2
Frequencies	1.0 GHz - 3.7 GHz
Frequency Steps	100 MHz
Number of Freq. Steps	28
SIMD Extensions	SSE4.2, AVX2, AVX512
L1, L2, L3 Cache	2 x 32 kB (instruction and data cache), 1024 kB, 22528 kB

Table 4.1: Configuration options of the Intel Xeon Gold 6130

also observed in real *WEPs*: (1) The best performing configuration is not always the most energy-efficient one (configuration C vs. D), and (2) There are multiple configurations, which offer a similar performance, but a different energy-efficiency, and vice versa (configuration A vs. B). In the next section, we will explain the energy-efficiency metric we are using.

Generally, a *WEP* can represent a *system view* or a *thread view*. A *system view* covers the configurations from a system perspective, i.e. the number and ID of the active cores and the frequencies of the active and inactive cores. This kind of profile is illustrated in Figure 4.4. A *thread view* does not include this system knowledge, but only the properties connected to a single thread, e.g. the frequency of the CPU core the thread is running on, or the instruction set it is using. A *Work-Energy-Profile* of a *thread view* can look different for the same application. This depends on the load of the remaining system, e.g. if other threads are using shared resources like main memory or shared cache. A *Work-Energy-Profile* of a *system view* is always the same for the same application[4], but becomes more populated the more complex the system is, i.e. the more configurations exist. This complexity can be caused by a larger number of cores or heterogeneous cores. To demonstrate the general applicability of our approach, we used different systems with a different complexity, which we will introduce in the following.

4.2.2 Test Systems and Configurations

We chose a set of fundamentally different systems designed for different use-cases: A server, a notebook, and a single-board computer, which mainly serves as a test platform for mobile devices. A necessary requirement for the test systems was the possibility to measure the power consumption without the overhead of an external measuring device. With an external measuring device, the power used by peripherals and the dissipation power of the device would distort the measurement results. The power consumption in the server and the notebook is determined by using RAPL counters, which are fine-grained enough to measure even short-running applications [HDVH12]. The power consumption of the two clusters of the single-board computer are measured with dedicated sensors, which are part of the board. All of these Systems run a Linux operating system. The server is running an Ubuntu 18.10. The notebook runs Ubuntu 18 LTS. The single-board computer runs an Ubuntu 16 LTS. Since the configurations we can change are properties of the CPU, we focus on the description of the different CPUs.

[4]Minimal differences can be caused by aged hardware, processes of the operating system, and inaccuracy of measurement.

System A: Server with multi-core CPU

Our first system is a server equipped with an Intel® Xeon Gold 6130. We've already used this system for the evaluations in Chapter 3, because the Intel SIMD instruction sets up to AVX512 are available. There is a total of four CPU sockets. Each of these sockets features 16 cores. Hyperthreading is possible with two hyperthreads per core. Table 4.2 summarizes the properties of this system. A special feature compared to the notebook and the single-board computer is the cpu driver. This CPU supports the *intel_pstate* driver. Generally, on Linux systems, users can use the *CPUfreq* interface to select a so-called scaling governor. This governor estimates the cpu frequency, which should be used according to a specific algorithm, e.g. to reach a maximum performance or to minimize power consumption. Then, the driver is responsible for setting this frequency. However, with the *intel_pstate* diver, the governors are bypassed and the driver uses its own scaling algorithms. These algorithms can be selected like governors and even have the same names, i.e. *performance* and *powersave*, but they do not behave in the same way as the governors with the same name[5].

System B: Notebook with many-core CPU

The second system we use is a conventional notebook equipped with an Intel i7-3960X. A summary of its properties is shown in Table 4.2. This CPU has 6 cores and each core can run two hyperthreads. Hence, 12 threads can run at once. The frequency can be adjusted between 1.2 GHz and 3.3 GHz, where the first frequency step is 100 MHz from 1.2 to 1.3 GHz, and the remaining steps are 200 MHz. L2 cache is 256 kB per core. The 15360 kB L3 cache is shared between all cores. The most recent available SIMD extensions are SSE4.2 and AVX. AVX2 and AVX512 are not available on this CPU. For this reason, the classical vectorized RLE-encoding shown in Section 3.1.2 will only work on 128-bit registers, and the RLE-encoding with conflict detection will not run at all. However, in contrast to system A, this system runs with the *acpi* CPU driver, which allows for a wide variety of selectable CPU governors, which are shown in the following list:

- **Performance** Runs at maximum frequency to reach maximum performance
- **Powersave** Runs at minimum frequency to draw minimum power
- **Conservative** Scales the frequency up or down in mainly equidistant steps according to the current load
- **Ondemand** Scales the frequency up or down according to the current load, but more dynamically than the *conservative* governor, e.g. the frequency can jump directly to the maximum without intermediate steps
- **Userspace** The user can define the frequencies, the same effect can be achieved by narrowing the frequency range for the other governors

System C: Single-board computer with heterogeneous CPU

Our third system is a single-board computer equipped with an ARM® big.LITTLE™ CPU, the ODROID-XU3. We chose this system because it is different from systems A and B in several ways. First, all components are hard-wired onto one board including the main memory, where the main memory of the other systems is conventionally plugged into DRAM slots. Second, the CPU features ARM® cores instead

[5]source: https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html, accessed 14/05/2020

Number of Cores	6
Hyperthreads per core	2
Frequencies	1.2 GHz, 1.3 GHz - 3.3 GHz
Frequency Steps	200 MHz
Number of Freq. Steps	7
SIMD Extensions	SSE4.2, AVX
L1, L2, L3 Cache	32 kB, 256 kB, 15360 kB

Table 4.2: Configuration options of the i7-3960X

	LITTLE Core Cluster	big Core Cluster
Core Description	ARM-Cortex A7	ARM-Cortex A15
Number of Cores	4	4
Hyperthreads per core	1	1
Frequency Range	0.2 GHz - 1.4 GHz	0.2 GHz - 2.0 GHz
Frequency Steps	100 MHz	
Number of Freq. Steps	13	19
SIMD Extensions	NEON	NEON
L1, L2, L3 Cache	32 kB, 512 kB (shared), -	32 kB, 2048 kB (shared), -

Table 4.3: Configuration options of the ARM® big.LITTLE™ ODROID-XU3

of Intel® cores. Third, the cores are heterogeneous. There are two clusters, one with four Cortex A7 cores, and one with four Cortex A15 cores. Both clusters can be used independently, i.e. the system is not restricted to cluster switching and frequencies can be set per cluster. All cores offer the same instruction set, but the A15 cores are more powerful and their cluster is equipped with more L2 cache. Hence, this cluster is referred to as the *big* cluster, while the A7 cluster is referred to as the *Little* cluster. The L2 cache is shared between all cores of the same cluster. There is no L3 cache. An overview of the properties of the two clusters can be found in Table 4.3.

The heterogeneity on this system produces a wide spectrum of configuration choices. There are 13 frequency steps on the A7 cluster and 19 on the A15 cluster, as well as 24 different combinations of active and inactive cores. The 24 combinations are composed of 5 variants per cluster (0 - 4 active cores), where one variant is subtracted, which only includes inactive cores. Hence, there are $5^2 - 1 = 24$ combinations. In combination with the cpu frequencies, this sums up to $13 \cdot 19 \cdot 24 = 5928$ different configurations per instruction set. Since the SIMD instruction set NEON is offered additionally to the scalar instruction set, there is a maximum of $5928 \cdot 2 = 11856$ available configurations. This is not further reduced by the CPU frequency of a potentially inactive cluster, because that frequency influences the performance of the active cluster if resources, e.g. memory buses, are shared, as explained before.

4.2.3 Creation of Work-Energy-Profiles - Benchmarking

One of our main challenges is the creation of *WEPs* covering a large number of possible hardware configurations. To tackle this challenge, we developed a benchmark concept to examine the behavior of performance and energy-efficiency for different hardware configurations in a uniform way. Fundamentally, the same test-case or application task has to be repeated and recorded for all possible configurations on the target systems. Moreover, not only the task but also the test data has to be the same in order to produce

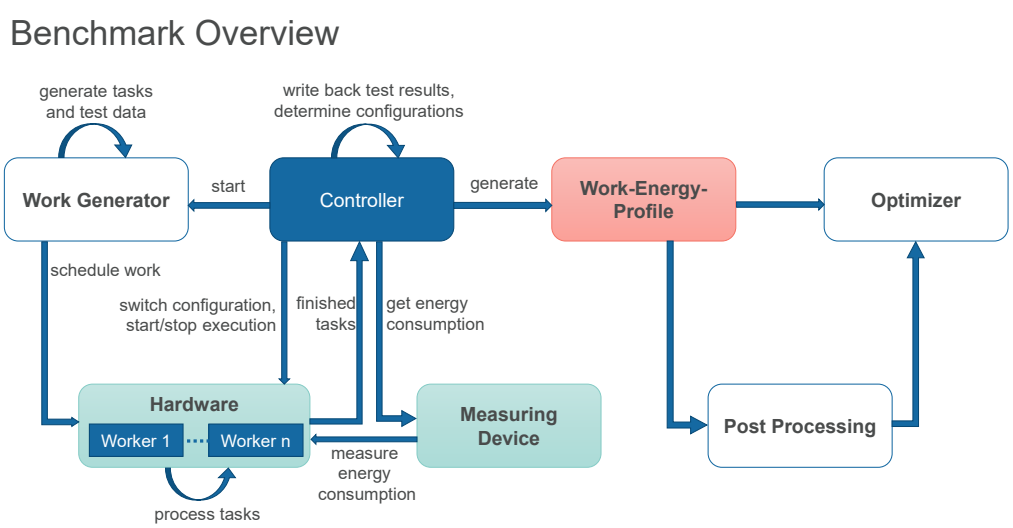


Figure 4.5: An overview of the benchmark setup.

comparable results. Therefore, we separated the generation of test-cases and data from the control-flow of our benchmark concept. Generally, an overview of our benchmark concept is depicted in Figure 4.5. The `Controller` is the centerpiece of our benchmark. It starts the `Work Generator` which produces tasks and test data. This `Work Generator` has to be adjusted for each application scenario. After the `Work Generator` has finished, the `Controller` chooses the first hardware configuration and starts the first test run. Within a test, the corresponding tasks are processed by every worker, whereas a worker is a thread running on a (virtual) core. The workers count their finished tasks. After a fixed time span, the `Controller` shuts down the threads and collects the number of finished tasks which are later used for calculating the performance. During an active test, the values necessary for the energy computation are recorded by a `Measuring Device`. Depending on the abilities of the `Measuring Device`, the energy computation is either done by the device itself or by the `Controller`. In both cases the final values are collected by the `Controller`. For eliminating odd side effects, a test can be run multiple times. This is repeated for all configurations, with the same tasks on the same test data. After all configurations have been processed, the `Controller` generates a *Work-Energy-Profile* for the selected task and data as depicted in Figure 4.6. This profile can then be used for in-depth analysis and optimization purposes, e.g. for choosing an energy-efficient configuration satisfying the requested performance constraints or for optimizing the applied algorithm.

Metrics

In addition to this general benchmark sequence as presented above, it is important, which metrics have to be measured at all. In our case, performance and energy-efficiency are the relevant metrics and specified in detail below:

Work and Performance: The *hardware* processes the *work* generated by the `Work generator`. This *work* consists of a task and the data to be processed. The task is repeated over the same amount of data, e.g. scan of records or hashing of keys, for each hardware configuration. Then, the number of finished tasks during a fixed time span is denoted as $work\,done$. Accordingly, the performance is denoted as:

$$performance = \frac{work\,done}{time}$$

The goal of our benchmark is not the evaluation of a real-life scenario but the comparability between the test runs with the same task and data definitions. Therefore, the processed data must have the same size and type in every task execution and break conditions must be reached after processing the same amount of data. Furthermore, since tasks can implement different operations on different data, a quantitative comparison between them is only possible when the performance is normalized to the same amount of processed data.

Energy: The electric power p is the product of the amperage $i(t_1)$ and the voltage $v(t_1)$ measured at a time t_1 (Eq. 4.1). Thus, it describes the power consumption of a measured system at a discrete point in time. In contrast, the electrical energy E is the integral over time of the whole power curve, consisting of all power values taken during the measurement (Equation 4.2). Thus, the consumed energy grows while time passes, whereas the power can rise and drop.

$$p(t) = v(t) \cdot i(t) \tag{4.1}$$

$$E = \int_{t_\mathrm{t}}^{t_\mathrm{end}} p(t)\,\mathrm{d}t = \int_{t_0}^{t_\mathrm{end}} v(t) \cdot i(t)\,\mathrm{d}t \tag{4.2}$$

The `Measuring Device` is responsible for determining amperage and voltage. The on-board power sensors of our ODROID-XU3 hardware satisfy this property. Intel® also introduced on-board energy sensors, called "Running Average Power Limit" (RAPL), with their Sandy Bridge microarchitecture [HDVH12]. Therefore, our benchmark concept can be applied to all of our test systems without special instrumentation. The computation of the energy from these measured values can either be done by the `Measuring Device` or by the `Controller`. For our test systems, the RAPL interface provides a counter for the energy, while the interface of the ODROID-XU3only provides the power, which leaves the computation of the energy to the controller.

An optimization for power is necessary, e.g. for thermal chip design or for dimensioning the necessary cooling, but the reduction of the electricity bill and the extension of the life span of a battery charge require energy optimization, i.e. the costs on the electricity bill are calculated from the energy drawn since the last meter reading. Hence, the primary goal for increasing energy-efficiency is the reduction of the overall energy drawn by the system while doing the same amount of *work*. Only reducing the power, e.g. by reducing the core frequencies, could lead to longer execution times and therefore to a higher energy consumption.

Energy-Efficiency: To achieve the objective of reducing the overall energy consumed by the system for a specific amount of work, the natural decision for quantifying the energy-efficiency is to calculate it as the quotient of *work done* and *consumed energy* [HSMR09, THS10, WFXS11]. Accordingly, we call this relation the *Work-Energy Quotient (WEQ)*.

$$WEQ = \frac{work\,done}{energy}$$

Since $^{work\,done}/_{time}$ equals the $performance$ for a certain operation, the mentioned publications rewrite the quotient as follows:

$$\frac{work\,done}{energy} = \frac{work\,done}{power \cdot time} = \frac{performance}{power}$$

However, this notation is an oversimplification. As already discussed, the power level can change during the measurement and the energy is its integral over time. Thus, the real power values cannot be restored from the energy, although the rewritten equation implies that this was possible. Vice versa, energy cannot be computed by an average power value when the required time is not necessarily the time needed for the computation but a part of a potentially normalized performance value. For this reason and for avoiding a confusion of execution time and normalized performance values, we argue to use $^{\text{work done}}/_{\text{energy}}$ instead of $^{\text{performance}}/_{\text{power}}$. For not mistaking the *WEQ* for energy efficiency definitions in other fields, e.g. the energy conversion efficiency, we do not just call this definition *energy-efficiency* but *work-energy quotient*.

Benchmark Setup and Dependencies

Up to now, we described our general benchmark sequence and defined our measured metrics. A full benchmark tests the same tasks and data configuration on all possible *hardware configurations* either on thread level or on system level, and these hardware configurations have to be set by the `Controller`. A *hardware configuration* contains the settings of hardware components which can be adjusted. Thus, it depends on the hardware system the benchmark is running on. A common configuration could consist of the following parts:

- Bitmask for defining active workers (a_i)
- Frequency of the physical cores (f_i)
- If available: Bitmask or index indicating if a specialized instruction set is used, e.g. SSE (s_i)

A configuration containing these options could be described by the vector $\{a_{0,..,n-1}\,,\ f_{0,..,m-1}\,,\ s_0,..,s_{m-1})\}$ with $n = m \cdot hyperthreads\ per\ core$, where m is the number of physical cores. Such a description is used to iterate over all possible hardware configurations by our `Controller`. For our example test hardware C, the ODROID-XU3, a configuration consists of an 8-bitmask $\{a_0, ..., a_7\}$, two frequencies $\{f_{A7}, f_{A15}\}$, and a byte indicating whether NEON is used or not. The bitmask indicates which cores are actively processing tasks. Since there are no hyperthreads, one bit per core is sufficient. The frequencies can be adjusted per cluster. Hence, there are two frequencies in every configuration, one for the A15-cluster and one for the A7-cluster.

Furthermore, we have to consider the following aspect: The generated work, the processing speed of the hardware and the specifications of the measuring device influence the accuracy of the tests. This implies that certain aspects have to be considered before implementing this benchmark. First, the `Measuring Device` must be suitable for the power range of the system, e.g. an accuracy of 1 W might be accurate enough for a rough estimate of a system drawing between 100 and 500 W but not for one drawing between 1 and 10 W. In our implementation, we use the integrated current sensors of the ODROID-XU3 and the RAPL counters on the Intel® systems, which fulfill this requirement. Second, the runtime of a single test must be long enough to gain significant results. In detail, it has to fulfill the following requirements:

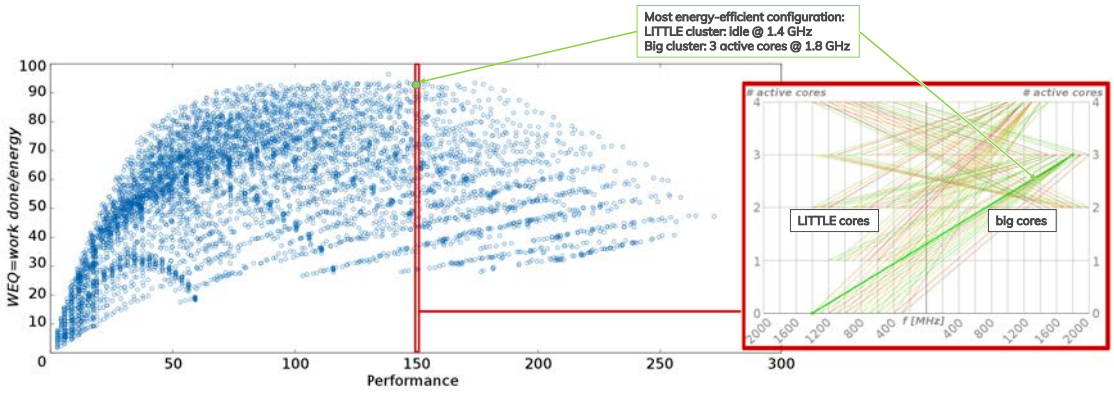


Figure 4.6: *WEP* of a system view on test system C, and a close-up of the highlighted performance range.

- Compensate for out-dated values due to the update frequency of the measurement device. Ideally a new test always starts right after an update cycle of the measurement device.
- Compensate for varying power values, e.g. if the power level changes during a test case but only one power value is recorded, the measurement result of this particular test case has only limited expressiveness, because the energy is only computed from this single power value.
- Finish a significant amount of work on all workers, e.g. if half of the cores are not able to process at least one request, there is no *work done* which can be compared, even if part of the cores would have finished much faster than the other ones.

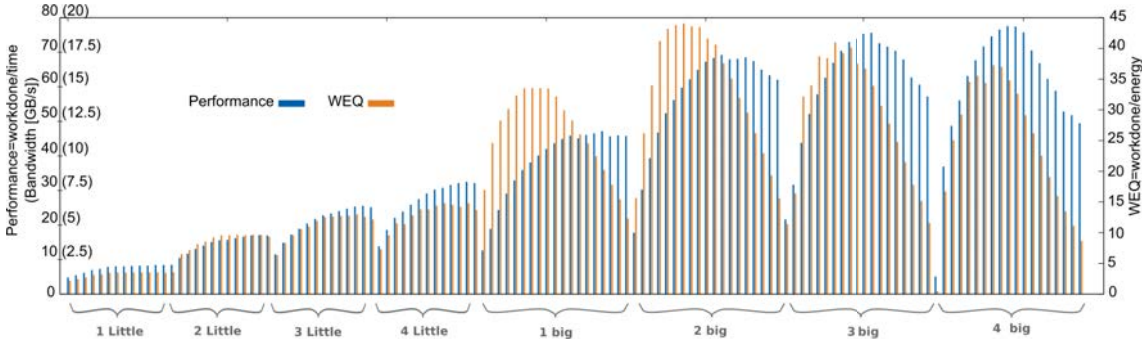
The integrated sensors of our example test system C only update their values approximately every quarter second. Hence, depending on the exact task, we run every test between 5 and 10 s to gain between 20 and 40 values to compute the energy from, and to finish some tasks on every worker even in very low performing configurations. On the test systems A and B, where we use the RAPL counters and the CPU frequency does not decrease below 1 GHz, one second per task is sufficient to fulfill the requirements.

4.2.4 Selected Work-Energy-Profiles

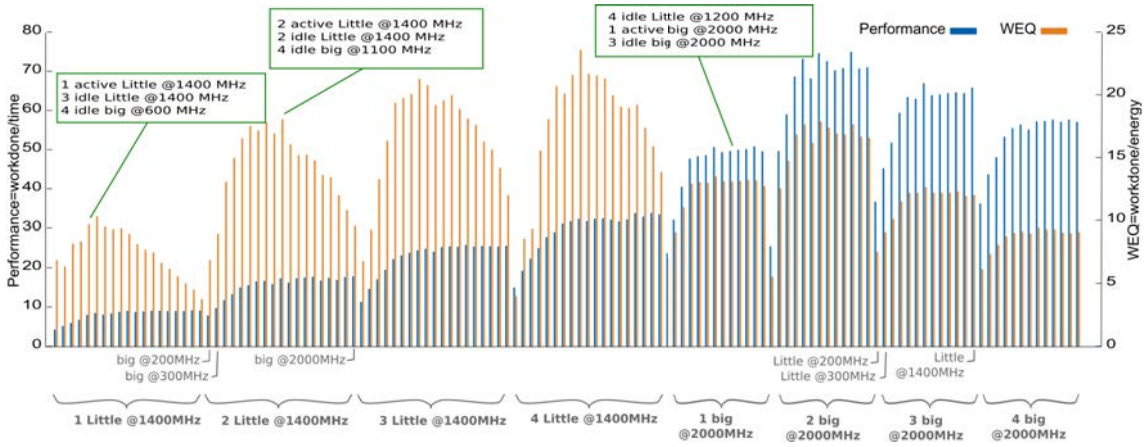
In this section, we present selected *Work-Energy-Profiles*. Each of these profiles was chosen to demonstrate certain effects. The first example shows a system profile on our heterogeneous test system C, which leads us to a deep-dive view of the effects of busy shared resources. The remaining examples run on the Intel systems. The second example, also a system profile, shows a comparison between our manually set configurations and the results of different CPU governors. Finally, the third example is a thread profile for different instruction sets. It shows that the significance of the chosen instruction set can change, depending on the load of the whole system.

Example 1: System Profile for a Synthetic Scenario on Test System C

To get a deeper understanding of our *WEP* approach, Figure 4.6 illustrates an example for an application task running on our test hardware. The application is a synthetic scenario for scalar processing and consists of an equal ratio of memory accesses and numeric computations. The left chart in this figure shows the corresponding *WEP*. The performance



(a) Varying frequency on active cluster. The idle cluster runs at full frequency.

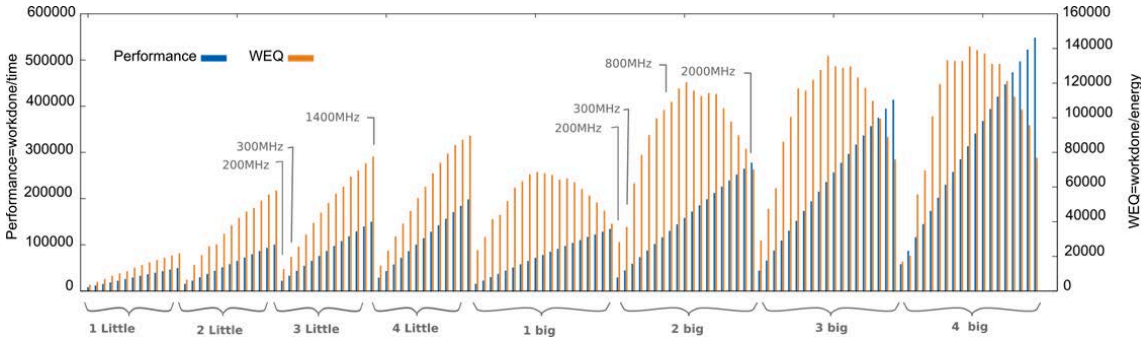


(b) Varying frequency on idle cluster. The active cluster runs at full frequency.

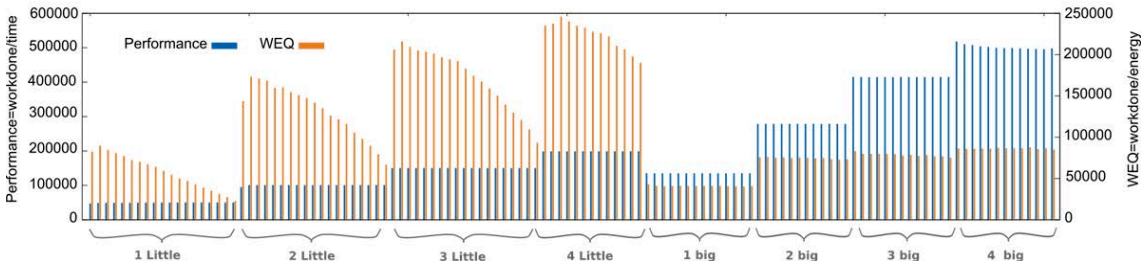
Figure 4.7: A memory bound use case with a reduced amount of configurations. Only the cores of one cluster are active at a time. The bars are clustered by the number and type of active cores. Within one of these clusters, each bar shows a different core frequency. The WEQ for the whole CPU (including both clusters) is shown.

is plotted on the x-axis, the y-axis shows the *WEQ*. Each dot in this chart represents a specific hardware configuration. As we can see, different hardware configurations offer a similar performance with a high variance in the *WEQ*. From this *WEP*, we are able to derive various insights.

Generally, we are able to utilize such *WEPs* to directly identify the most energy-efficient configuration (high *WEQ*) for a desired performance range and application task. In Figure 4.6, we highlighted a specific performance range using a vertical slice. This performance range can be realized with various hardware configurations as depicted in the right chart of this figure. In this chart, the most energy-efficient configuration is highlighted by a thick green line. The x-axis indicates the frequency of the clusters, the y-axis shows the number of active cores. The left side shows the A7 cluster, the right side the A15 cluster. A line connects the configuration of both clusters and forms the complete configuration. The least energy-efficient ones are marked with a thin red line. This close-up shows the variety inside the configurations, which produce the same performance but a different *WEQ* and the most energy-efficient configuration is not necessarily the most obvious one. For the highlighted performance range, the most energy-efficient configuration consists of 3 active A15 cores running at 1.8 GHz, while all A7 cores are idle at 1.4 GHz. The graph also shows a number of configurations with high CPU core frequencies on both, the idle cluster and the active cluster, which show a comparatively low *WEQ*.



(a) Varying frequency on active cluster



(b) Varying frequency on idle cluster. The active cluster runs at full frequency.

Figure 4.8: A compute bound use case on test system C with a reduced amount of configurations. No shared resources, i.e. no L2 cache and no memory i/o, is used. Only the cores of one cluster are active at a time. The WEQ for the whole CPU (including both clusters) is shown.

As already mentioned, the frequency of an idle cluster can still influence the overall performance if resources are shared. In this specific case, the cache and core clock are only shared within the same cluster, but the memory and the memory bus clock are shared between both clusters. We assume that the shared bus clock is the reason for the unexpected behavior.

To investigate this assumption, we run a completely memory bound scenario for a reduced set of configurations, which cover only active cores on one cluster. Figure 4.7 shows the results of the measurements. It shows the performance and WEQ for the configurations which contain only one active cluster. In Figure 4.7(a) the frequency on the active cluster is increased, in Figure 4.7(b) the frequency on the idle cluster is increased. The other cluster is running at its maximum frequency. In both cases the *WEQ* does not grow anymore for any core configuration when switching to the higher frequencies. When changing the frequency on the A15-cluster it even decreases after ≈1 GHz. As already discussed in Section 4.1, a higher frequency requires a higher supply voltage if the overdrive voltage cannot be reached within the given cycle time, and the switching power depends quadratically on the supply voltage. We assume that this point is reached at ≈1 GHz and that this is the reason for the decreasing *WEQ* alongside the decreased performance. When looking at the number of active cores on the A15 cluster, the *WEQ* decreases when enabling more than two cores and setting the frequency to more than 1 GHz. Additionally, the mean performance decreases in Figure 4.7(b) while in Figure 4.7(a) it increases by an insignificant amount.

Looking at the *big* cluster in Figure 4.7(a), the performance decreases when the frequency is increased over 1.2 GHz. When the frequency of the active cluster is increased, memory access is required after shorter intervals. If additionally more cores are activated, these

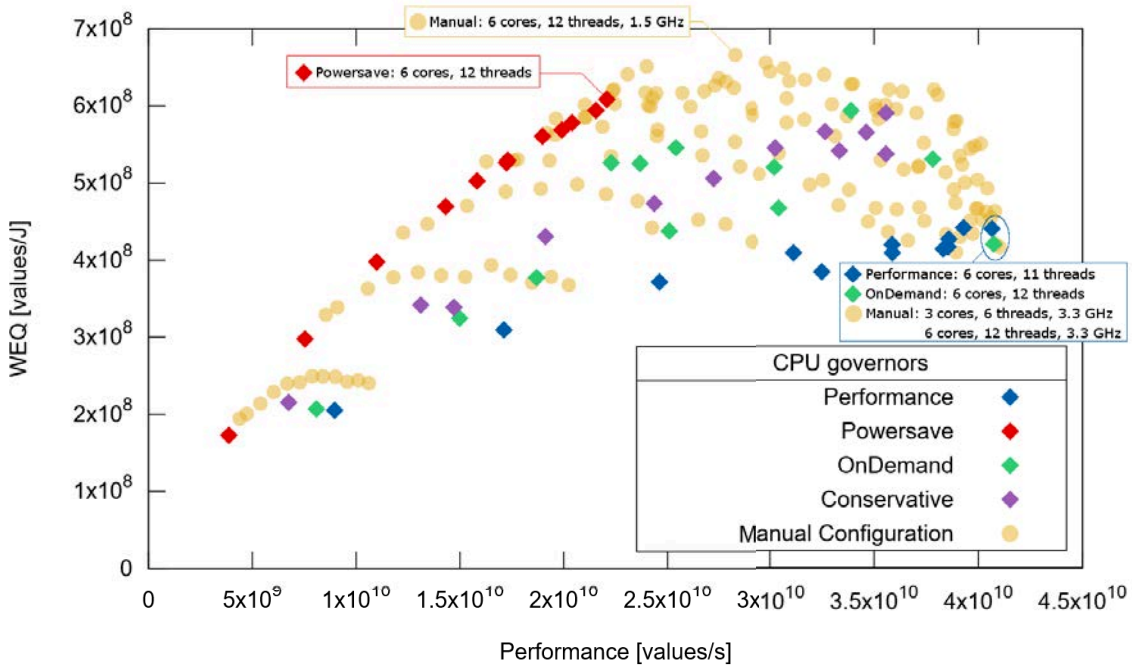


Figure 4.9: A *WEP* on system B for RLE with a run length of 100000000 (negligible write access) and a comparison to the behavior of the different CPU governors.

cores also require memory accesses in short intervals. These requests cannot be satisfied immediately once the memory buses are saturated. When this happens, the cores have to wait one or more stalling cycles until their request is answered. Hence, the performance goes down. The *LITTLE* cluster has a maximum CPU frequency of 1.4 GHz, which is not enough to suffer from this effect.

In Figure 4.7(b), where the frequency of the idle cluster is changed, the performance does not drop after a peak has been reached. This is because the idle cluster does not access the memory and therefore, it does not cause any stalling cycles on the other cores. However, the frequency of the idle cluster has to be increased to at least 500 MHz to show the maximum performance on the active cluster. This supports the assumption that the memory bus is shared and clocked down indirectly by the low frequency of the inactive cluster. This is because a CPU cannot process the data as fast as it is delivered, if the memory bus has a higher frequency than the CPU core. Hence, the bus has to be clocked down, at the latest when the cache is completely used by buffered data from the memory, which still has to be processed.

The counter check for the explanations of the observed effects can be done with a completely compute bound use-case, which does not use any shared resources, i.e. no L2 cache and no main memory accesses. We run such a case, too. The results are shown in Figure 4.8. Without the use of shared resources, the performance should increase while the CPU frequency of the active cluster increases. As shown in Figure 4.8(a), this holds true. However, the *WEQ* does not increase further beyond 1.2 GHz on the big cluster, which is again caused by the superlinearly increasing power consumption. If the effects in Figure 4.7(b) are caused by a memory bus frequency scaling, they should not be observable, when the memory bus is not used. This is confirmed by our counter check experiment as shown Figure 4.8(b). The performance stays at the same level regardless of the frequency of the inactive cores.

Example 2: System-Profile for RLE on Test System B

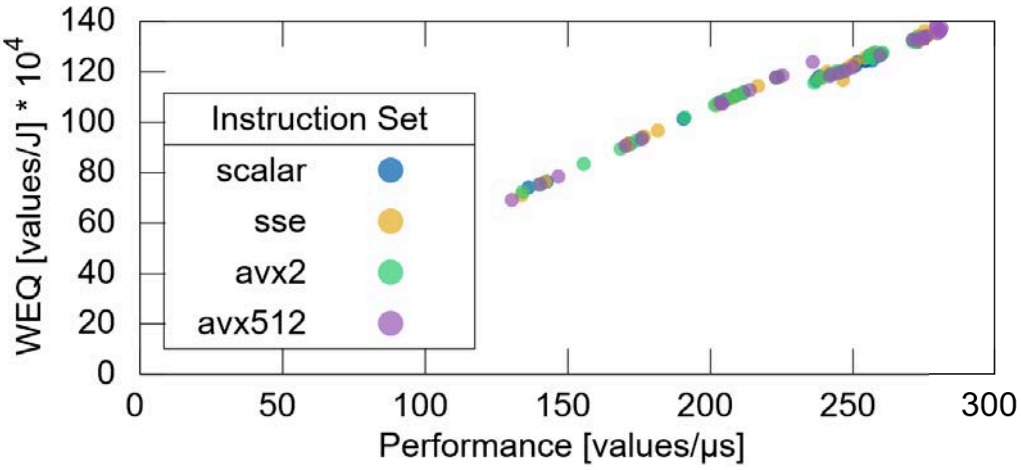
A feature of our test system B is the availability of different scaling governors. This enables us to compare our benchmarked *Work-Energy-Profiles* to the behavior of the system depending on the chosen governor. To do this, we run a classical vectorized RLE compression. Our uncompressed test dataset with $4 \cdot 10^8$ integer type numbers requires about 1.5 GB of space. We chose to include only 4 runs in this data set, realized by a run length of 10^8, to keep the offset of memory write access low and to read the majority of numbers only once. However, reading 1.5 GB ensures an execution time, which is long enough for the *conservative* and *ondemand* governors to adjust the CPU frequencies. The *performance* and *powersave* governors always run on maximum resp. minimum frequency. We first increased the number of used cores, then we enabled hyperthreading at one core after another. The core frequencies were either set by a governor or manually by our benchmark. Figure 4.9 shows the results for these benchmarks.

As in the first example, there are multiple configurations, which provide a similar performance, but a different *WEQ*. However, each governor only covers a subset of the full *WEP*. The lower performance range is covered by the *powersave* governor. This governor provides a similar *energy-efficiency* as the most energy-efficient configurations of our full profile for the performance range it covers. However, since the *powersave* governor runs all cores at their minimum frequency, even the maximum number of active threads only provides about half of the possible performance. For a higher performance, one of the other governors has to be chosen. None of these governors provides the optimal *WEQ* for any performance range. Only the highest possible performance is covered by several configurations of our profile as well as by the governors. These configurations are highlighted in Figure 4.9. Besides the *ondemand* and *performance* governors, there are two configurations of the full *Work-Energy-Profile*, providing maximum performance. Surprisingly, one of these configurations uses only half of the available cores and threads. A reason for this could be that the memory bus is already saturated with 6 threads. A higher number of threads, which also access the shared memory buses, can cause the performance to stop increasing or even to decrease again, as already explained in example 1.

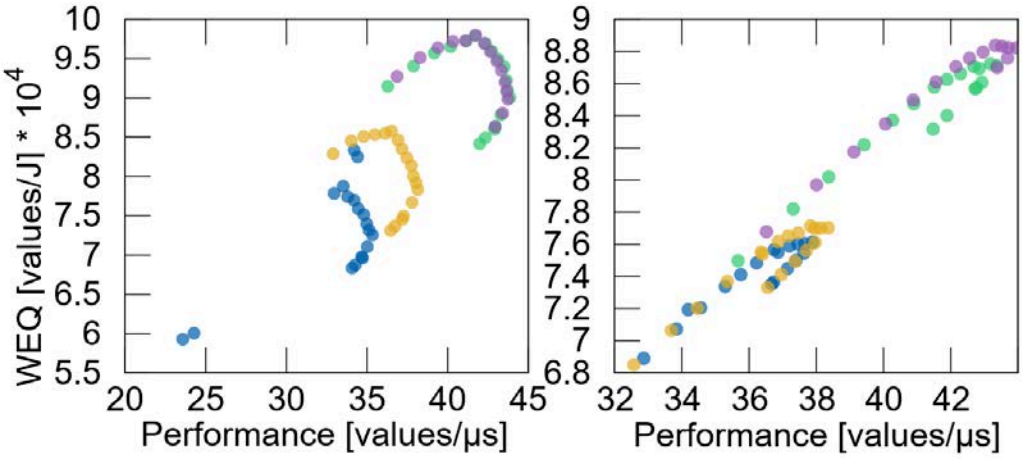
When interpreting these results, it is important to note that only one governor can be chosen at a time, while all of them have their individual limitations. The *powersave* governor only covers the lower performance range but shows optimal *WEQ* in this range. The *performance* governor covers a wide performance range up to the maximum reachable performance, but shows the lowest *WEQ*. The *conservative* governor, which adapts its frequency iteratively, provides a medium *WEQ*, but does not reach maximum performance. Finally, the *ondemand* governor, which is not bound to iterative steps when increasing the CPU frequency, also shows a medium *WEQ*, but it can reach the highest performance. In conclusion, the CPU governors can span the same performance range as our *WEP*, but not with the same *WEQ*. The latter can only be reached by the *powersave* governor for a limited performance range. The highest possible *WEQ* is not even reached by the *powersave* governor. It requires the maximum number of threads at 1.5 GHz, which is apparently not a frequency set by any of the governors.

Example 3: Thread-Profile for a Project Operator on System A

Our last example shows a thread profile for different instruction sets. The workload is a project operator, which consists mainly of *gather* and *store* primitives. We chose system A, because it offers the most instruction sets out of our test systems. Since this system runs with the *intel_pstate* driver instead of the acpi driver like system B, the cpu governors are not available by default and the frequency setting is done by the driver. For being able



(a) No other system load. The frequency is set for all cores of the system.



(b) Memory I/O system load. CPU frequency set for the whole system (left) and for the observed node only (right).

Figure 4.10: Thread profiles for a projection on system A with different instruction sets. In this example, a meaningful choice of the instruction set becomes more important when the shared resources, i.e. the bandwidth, are busy.

to change the frequency manually, we set the driver into passive mode. This enables our benchmark to use the *CPUfreq* interface, which is also used for the other test systems. However, this system still behaves different. For instance, when checking for our set frequency, we recognized, that our frequency setting is ignored for higher frequencies when the system load is high. This leads to configurations, which are set but are not active. We call such configurations dead configurations. Dead configurations are not a hindrance for the application of a profile. Since we measure the performance and energy for whatever the CPU driver does with our configuration, the profile we create still shows the results for our set configuration. This causes dense clusters of configurations in the graphical depiction of our *WEP*. Albeit, knowing these limitations can reduce the time necessary for benchmarking.

For our example, we changed the frequency on the system level, i.e. all cores were assigned the same frequency, and on the NUMA node level. Setting the frequency on a core level resulted in ignored frequency settings. Thus, we do not consider this option on system A. Then, we changed the system I/O load from none to fully busy. This means, that

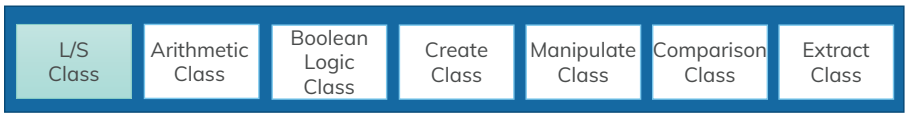
under full load, all other cores run a thread, which produces continuous memory access. Figure 4.10 shows the corresponding *WEP*. In Figure 4.10(a), our observed thread is the only thread, which produces any load on the system caused by the user[6]. The frequency in profile (a) is set for all cores on the system. The graph shows, that there is no significant difference between the instruction sets. Performance and WEQ behave roughly proportional and the instruction sets span a similar range.

The picture is different when the other cores produce memory I/O traffic. Figure 4.10(b) shows the profiles for this system load. These two profiles differ in the frequency setting. In the left profile, the frequency is set for the whole system. In the right profile it is only set for the NUMA node. A first observation, which is expected, is that the performance and *WEQ* are significantly lower than in profile (a). A second observation is that performance and *WEQ* do not scale linearly anymore. Instead, the profile shows an arc for each instruction set, similar to those, which were produced on the other test systems. Further, these arcs, are not as close as the configurations in profile (a). Peak Performance can only be reached by AVX2 and AVX512 in the profiles in Figure 4.10(b), where profile (a) also had well performing configurations using SSE and scalar processing. Especially when setting the frequency on a node level (right), there are two clusters of significantly differently performing instruction sets: one using SSE and scalar processing, and one using AVX512 and AVX2. Hence, if there is a system load, the choice of the instruction set does make a difference. Out of the two profiles, the one with the frequency setting on the system level reaches a slightly higher *WEQ*. This is because the system operates the other nodes in performance mode if we do not set the frequency manually. Hence, the cores on the other nodes potentially run faster than with our manually set frequency, but without reaching a higher performance due to the already saturated memory bandwidth.

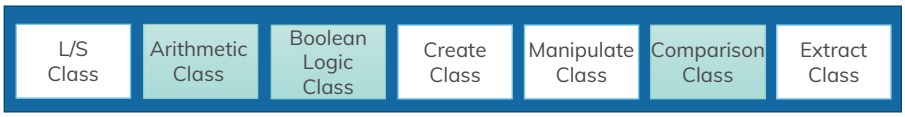
4.3 WORK-ENERGY-PROFILES FOR VECTORIZED QUERY PROCESSING

Benchmarking the *WEPs* for each operator and compression algorithm and each parameter, which could influence the profile, e.g. the selectivity, takes a large amount of time. For instance, our test system C offers 5928 configurations per instruction set and the power sensors update every 0.26 seconds. To get 20 power values, which proved to be enough to amortize for occasional outliers and operating system tasks, a single benchmark for one instruction set takes $5926 \cdot 0.26\mathtt{s} \cdot 20 \approx 8.56\mathtt{h}$. Test system B offers fewer configurations and the counters update more frequently. Hence, a single *WEP* on system B is done after half an hour. However, it would still take weeks to benchmark the *WEPs* for all operators, compressions, and parameters, which can occur during query execution. For this reason, we argue to benchmark only the primitives and approximate the profile for more complex tasks from these basic profiles. We aim to find the most energy-efficient configurations for different performance ranges by using the approximated profiles. The absolute performance or *WEQ* values are not required to be exact for this goal as long as their ratio is determined correctly. This allows us to break down our applications to the use of primitives and their ratios. In this section, we will explain, which basic profiles we are using, how the approximation works, and present selected examples.

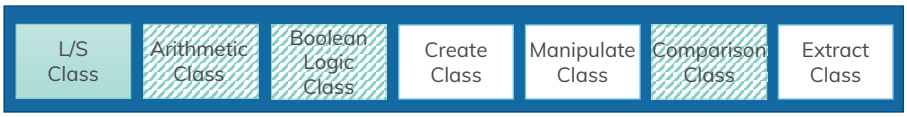
[6]Note that the operating system also produces occasional system load. To reduce this load, we disabled all unnecessary background processes.



(a) Load and Store Profiles



(b) Compute Profiles



(c) Processing Profiles

Figure 4.11: We use three different categories of basic profiles. Load and store profiles are retrieved from the primitives of the L/S class. Compute profiles are retrieved from the primitives of the Arithmetic, Boolean Logic, and Comparison classes. Processing profiles combine load primitives and primitives of one of the compute classes. The remaining classes, i.e. those working with single registers without memory access or computation, only play a minor role in our implementations and are therefore not specifically addressed.

4.3.1 Basic Profiles

Each operator or compression algorithm basically does 3 things: (1) read data from memory, (2) compute something from this data, and (3) write back a result. Reading and writing main memory is done with the primitives of the Load/Store class, while the compute part can be a combination of any primitives of the other classes. This results in three different classes of basic profiles. Load and store profiles, compute profiles, and processing profiles, which are a combination of load primitives and primitives of the compute intensive classes, e.g. the arithmetic or comparison classes. The classes, which access a register without doing a computation or accessing the memory, i.e. the extract, create, and manipulate classes, do not play a significant role in our implementations. They are never called within the main loop of any of our operators. For this reason, they are not specifically addressed here.

Load and Store Profiles Load and store profiles are retrieved from the primitives of the L/S-class of the *TVL* as highlighted in Figure 4.11(a). The primitives of the Load/Store class can show a sequential or a random memory access, which is the main factor influencing the performance and energy-efficiency. Hence, an approximation requires at least 2 basic profiles for reading access and 2 basic profiles for the write access, one for sequential memory access and one for random memory access each. For the sequential access, there are only the *load* and the *store* primitives, while the random access can be realized with multiple primitives, e.g. *gather*, *scatter*, *masked load*, and *compress store*.

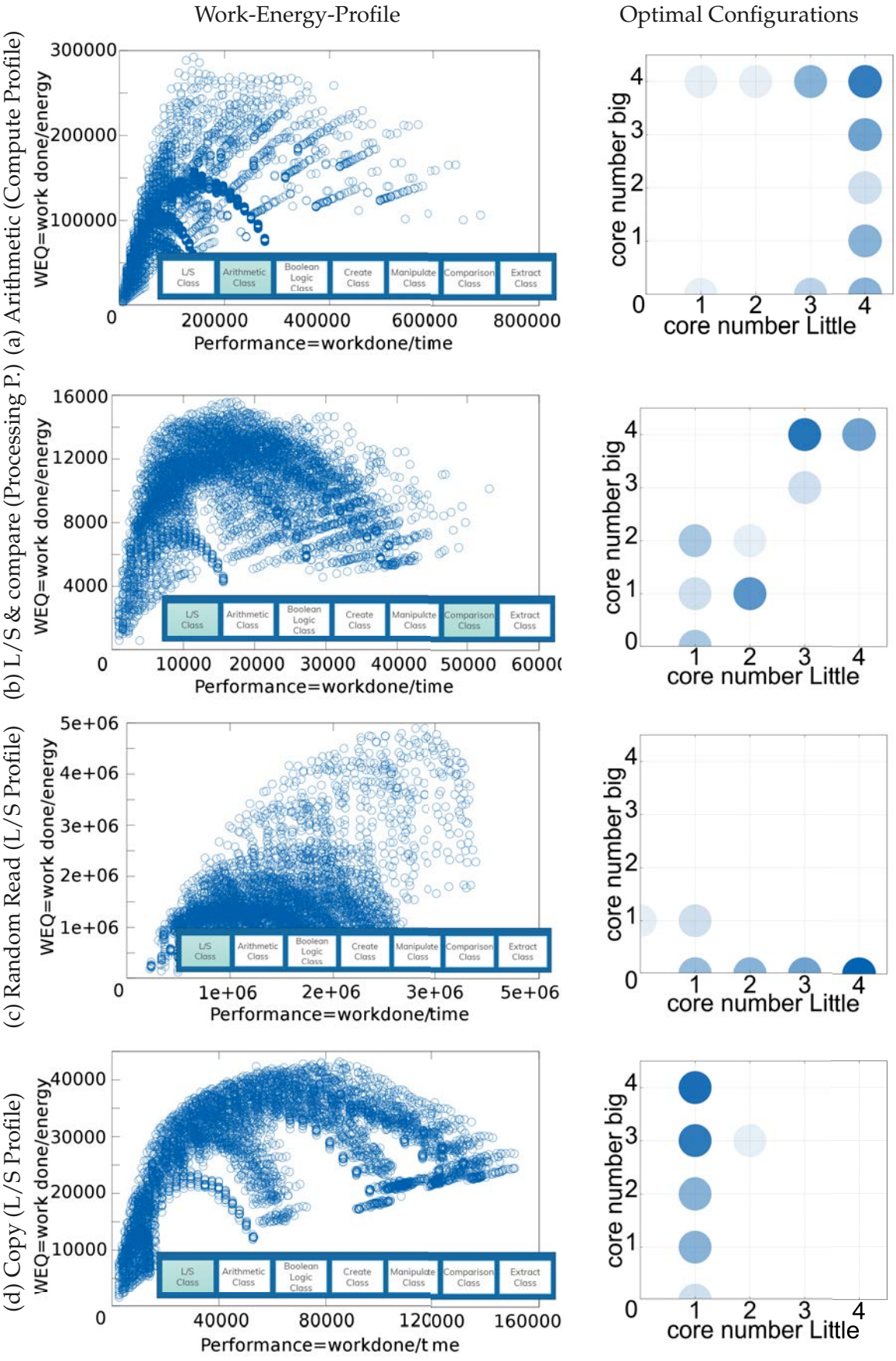


Figure 4.12: Examples for basic *WEPs*. The left side shows the *WEPs*. The right side shows the core configurations, which span the upper hull. In this example, *work done* refers to one iteration, which can include a different amount of processed values.

Compute Profiles Figure 4.11(b) highlights the classes of the *TVL*, which can be used to create compute profiles. Computing something from the data read from memory, can be arbitrarily complex. For instance, an aggregation might only require additions, while a join might implement a sophisticated hash function. However, each of these tasks is composed of primitives, of which some have the same performance and *WEQ*, e.g. a *bitwise AND* and a *bitwise OR* show the same behavior. This is a trivial finding, because on the circuit level, both operations can be implemented with a single logic gate per bit. There are more primitives, which show the same behavior because of obvious reasons, e.g. *min* and *max*, *less* and *greater*, *shift left* and *shift right*. In these cases, only one of the primitives require a *WEP*. This significantly reduces the time, which is necessary to build all required basic compute profiles. Moreover, all compute operations are a combination of low level operations, e.g. a subtraction is a combination of a sign change and an addition. The addition is again a combination of logical AND and XOR operations. Hence, if the complexity of an operation is known, the profiles of all compute operations can be reconstructed from a few compute profiles.

Processing Profiles For each set of data read from memory, some kind of predefined computation is triggered. If this wasn't the case, the application is either doing a copy operation, or the read data is not used anymore, which causes the compiler to optimize out the whole memory access. This is called dead code elimination. It is a basic feature of the gcc compiler among others, and it is by default activated using the lowest level of optimization (-O1)[7]. Hence, it does make sense to combine the read memory access and the compute operations into one profile. We call such profiles *processing profiles*. The classes used for creating such primitives are highlighted in 4.11(c). As long as there is no branching with heavily different complexities of the branches, the relation between read data and executed compute operations stays the same regardless of the properties of the incoming data. This approach shows a more accurate picture of the effects of limited bandwidth, frequency scaling across shared resources, and other side effects. However, the ratio between the read primitives and the compute primitives in a processing profile must be close to the actual ratio in an application. Therefore, a set of profiles with different ratios is required to match a variety of applications.

Example Basic Profiles

The approximation of a profile can either be done for a thread profile or for a system profile. Hence, basic profiles can be both: thread profiles and system profiles. We will show some examples for system profiles on our heterogeneous ARM system, and some examples for thread profiles on the Intel server system.

System Profiles on Test System C As already argued, main memory is the bottleneck for in-memory database systems and therefore, we use different basic profiles with significantly different memory access patterns. We chose basic profiles, which map directly on frequently used operations to facilitate the understanding of the corresponding tasks, which we implemented for our `Work Generator`. For some initial tests of our approach, we used scalar processing of the following four basic example profiles:

(1) **Arithmetic (Compute Profile)** This task simulates operations that do not involve any main memory utilization, e.g. the solution of mathematical equations. In our implementation, our task includes taking 512 square roots per iteration.

[7] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, accessed 25/05/2020

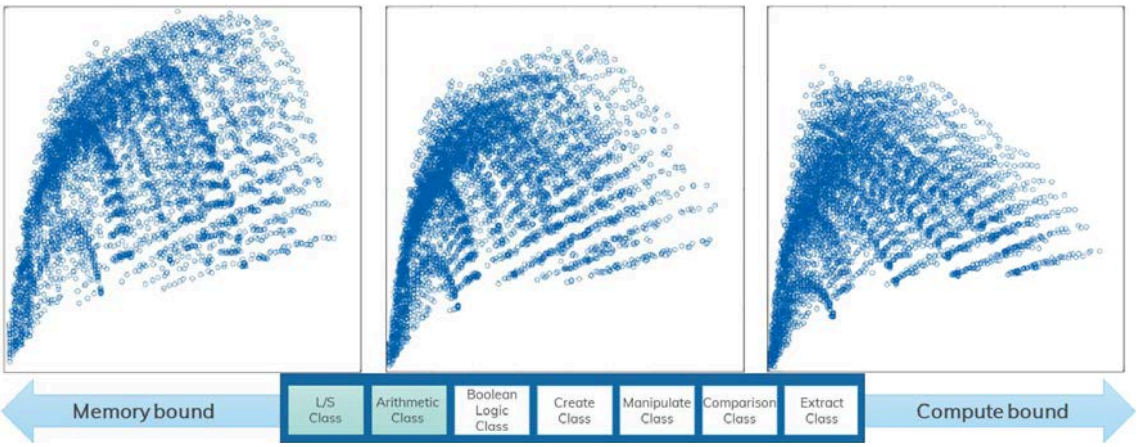
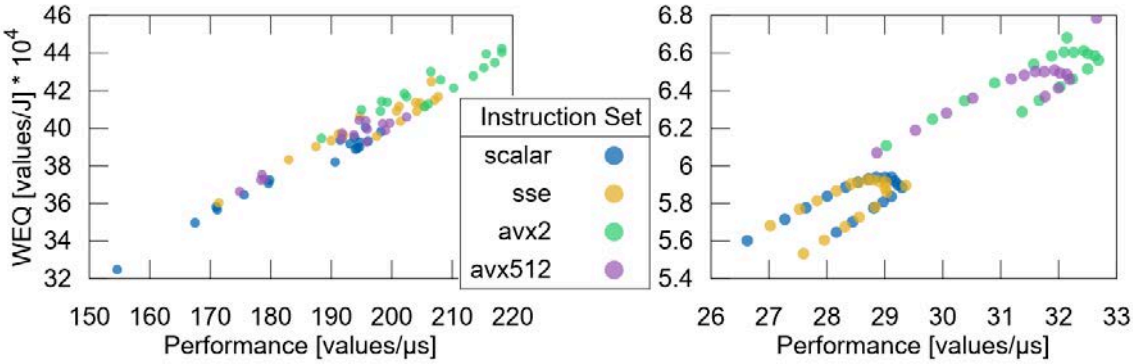


Figure 4.13: The shape of processing profiles with different read to compute ratios. The profile on the left side has a comparably high amount of memory accesses compared to the compute operations. In the profile on the right side, this ratio is inverted. The profile in the middle is neither solely memory bound nor compute bound.

(2) **L/S and compare (Processing Profile)** This task represents bandwidth-bound operations, like column scans exhibiting a sequential main memory access pattern and a limited computation overhead. The implementation requires loading data from memory and a compare operation. Each iteration used 128 KB.
(3) **Random Read (L/S Profile)** This task represents main memory-bound operations exhibiting a random main memory access pattern. It uses only primitives of the L/S class. This represents operations such as e.g. lookup. The lookup is done using 32B in each iteration
(4) **Copy (L/S Profile)** This task executes a data copy operation, which involves only sequential memory access. As test data, we used 128 KB per iteration.

While this list of basic profiles is not conclusive, it is sufficient to represent simple operator implementations, e.g. an aggregation, and some compression algorithms, e.g. classical RLE. The corresponding *Work-Energy-Profiles* for each operation are shown in Figure 4.12, where *work done* denotes to the number of performed iterations. As we can see in this figure, the profiles show a completely different shape. Again, each dot in each diagram represents a hardware configuration and for each hardware configuration, we determined a performance (x-axis) and a WEQ value (y-axis). That means, each basic profile shows a different behavior with regard to performance and energy efficiency. For instance, the significantly different shape between the Load and Store profiles show the importance of distinct basic *WEPs* for random and sequential memory access. All basic profiles have in common, that there are hardware configurations offering the same performance with a different energy efficiency value.

In the examples above, the second basic profile is a processing profile, because it requires reading data from memory and doing a comparison. Depending on the ratio between memory access and compute operations, these processing profiles look different. In Figure 4.13, we show the shape of processing profiles for scalar processing with different read to compute ratios. The memory access is sequential. From the left to the right, the amount of compute operations in relation to the read operations is growing. Thus, the left profile is memory bound, the right profile is compute bound. The profile in the middle is neither of them, similar to the scan profile. The high number of configurations in the memory-bound profile, which show a similarly high performance, are the CPU core frequencies above 1.4 GHz. As already explained, high frequencies are not always beneficial in memory bound scenarios. In the compute bound profile on the right side, a higher frequency does increase the performance. Hence, only the configurations with higher frequencies cover the highest performance ranges.



(a) store (b) gather

Figure 4.14: Basic *Work-Energy-Profiles* for the gather and store primitives on system A. The memory is also used by all other CPU cores. The frequency is set per NUMA node.

Thread Profiles on Test System A In the last section, we presented the *Work-Energy-Profiles* for a projection on System A. In this section, we show the basic profiles for the two main primitives, which are used by a projection: gather (random read access) and store. We decided to show the *WEP*s for a system, which is under an I/O load, because it is a rather unrealistic scenario that a multi-core system runs the threads, which require memory access, on only one core at a time. Figure 4.14 shows the *WEPs* of the store and gather primitives on thread level for different instruction sets. The frequency was set on a node level, because this is the most fine-grained reliable adjustment, which we can achieve with the used system.

By comparing the two profiles, a few expectations are met. First, the *store* primitive shows a higher performance and *WEQ* than the *gather* primitive. This was expected because the *gather* involves random memory access and an additional sequential access to load the offsets. The store primitive only involves a sequential memory access per iteration. Second, after a peak has been reached at 2.2 GHz, performance and *WEQ* decrease again. This effect is more significant in the *gather WEP*. This was also expected, because the gather is bound by the random memory access, which is slower than sequential memory access. Hence, this bottleneck is reached earlier, i.e. with a lower cpu frequency.

Another interesting insight is the shape of the gather *WEP*. It looks very similar to the *WEP* of the project operator shown in Figure 4.10(b). Since the store only takes a fraction of the time of the gather in our projection, the whole operator is heavily dominated by the gather. Therefore, the *gather* and *project* profiles look very similar.

Composition of the Upper Hull

When looking at a single profile, e.g. Figure 4.12(d), there are recurring arcs which differ in their length, height and width. Since they appear, more or less significantly, in all profiles, they are likely not to be a random pattern. Figure 4.12 also shows the core configurations which belong to the most energy-efficient configurations of the profile. A visualization of the configurations which belong to these optimal core configurations is shown in Figure 4.15. It shows how the arcs in the profiles are generated for the copy operation. Each core configuration spans an arc. Moreover, Figure 4.15 shows that only a few configurations serve the upper half of the hull of this profile. The right side of Figure 4.12 shows the optimal core configurations for the *Work-Energy-Profiles* next to

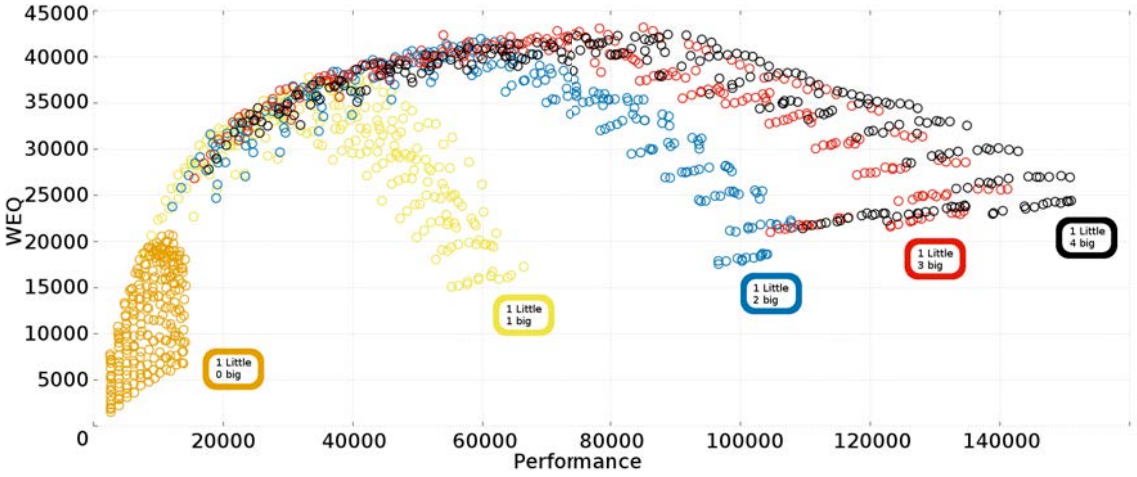


Figure 4.15: Reduced *Work-Energy-Profile* to only show the core configurations producing the optimum for a read/write operation.

them. While they differ for each test case, they are always only a subset of all possibilities. Hence, the upper part of the hull is made of a few combinations of active cores.

In our example in Figure 4.15, one configuration, namely four A15 and one A7 core, even covers a wide range of the optimum without ever falling significantly under the optimum of any performance range. Only very small performances cannot be reached. By staying at this core configuration and only changing the frequency depending on the requested performance, we always operate with a good energy efficiency while minimizing context switching. We call such configurations *robust configurations*. Using a robust configuration over a slightly more energy-efficient configuration with a different core usage, can be beneficial because it avoids context switching when adapting configurations on a relatively low cost.

4.3.2 General Approach

To approximate *Work-Energy-Profiles*, we propose to combine primitive profiles in a linear way. For this approximation to succeed, the primitive profiles have to be scaled to reflect their influence on a whole operator or compression as realistically as possible. This can be done in an automated way: At first, all *WEPs* are normalized to a comparable performance metric, e.g. values/s instead of loops/s. Ideally, the work generator already created the tasks in a way, that all profiles are already normalized. Then, the profiles are scaled to reflect their influence on performance and *WEQ* in the overall application. To do this, one profile, ideally the one with the highest performance and *WEQ* of all profiles, serves as a reference. The highest performance and *WEQ* of this profile are the reference values. All other profiles are scaled, such that their maximal values are the reference values multiplied with the fraction of their maximum values of the reference value. For instance, if profile A has a maximal performance value of 21 and profile B has a maximal performance value of 7, i.e. $1/3$rd of 21, the maximal performance value of profile B is scaled from 7 to $3 \cdot 21 = 63$. Since $63/7 = 9$, all performance values of profile B are scaled by a factor of 9. After an approximation, the new profile, containing (performance,WEQ)-tuples for i different configurations, is obtained from j scaled low-level profiles and the tuples of the new profile are determined by

$$(Performance, WEQ)_{i,new} = f^{-1}\left(\sum_{0}^{j-1} w_j \cdot f(Performance, WEQ)_{i,j}\right) \tag{4.3}$$

where w_j is a weighting factor which describes the influence of the profile j. The adjustment function f modifies the performance- and WEQ-values for the combination. This is necessary if the limiting factors, i.e. the basic profiles, do not scale linearly when the parameters of the operation are changed. Hence, there are 3 choices, which have to be made, before the approximation can be done:

1. *Basic profiles* The appropriate basic profiles have to be chosen, especially the kind of memory access, i.e. sequential or random, is important to get useful results. If a processing profile is chosen, the ratio between read and compute operations is required to select a matching processing profile.
2. *Adjustment Function* While most of our applications do not require an adjustment function, there are exceptions to this rule. We will present of these exceptions, classical vectorized RLE, in the examples below.
3. *Weighting Factors* The weighting factors depend on how many primitives, which can be represented by the same profile, are used in the application. In the cases, where we use a processing profile and a write profile, the ratio between read and write accesses is sufficient, while the ratio between read and compute operations is already reflected by the choice of the processing profile.

The basic profiles and the weighting factors can be gathered in an automated way. To do this, an application, an operator, or a compression is executed with a small amount of data, but all other parameters, e.g. the selectivity, is not changed. During this execution, all used primitives are counted. The reduced amount of data causes shorter runtimes but does not change the ratio between the used primitives. For example, a simple filter with a selectivity of 10% and n input values will always require reading n values, comparing n times, and writing out $n \cdot 0.1$ values. The total amount of values in a dataset does not change this relation. Since the amount of operators and compression algorithms in a database are finite and usually manageable. These runs can be done once during setup time for a representative set of parameters. The type of used primitives then defines the basic profiles, while their ratio defines the weighting factors. The adjustment function could also be generated automatically by doing a regression on the primitive count for different parameters, but since this is barely necessary, we chose to do it manually.

4.3.3 Example Operations

In the following, we will show two examples. First, we show a trivial operator, i.e. the project operator, which we already introduced in Section 4.2.4. Then, we show classical vectorized RLE, which requires an adjustment function and weighting factors depending on the run length.

Project Operator

In a column-store, a project operator is required to reconstruct tuples from a given index. Since this is required in all queries, which do not analyze only a single column, the project operator is used frequently. Additionally, it is an expensive operator, because it requires random memory access to gather the values from different rows. In this example we assume that the result of the operator is materialized, such that there is a memory I/O load caused by the remaining system, and that the frequencies are set per node.

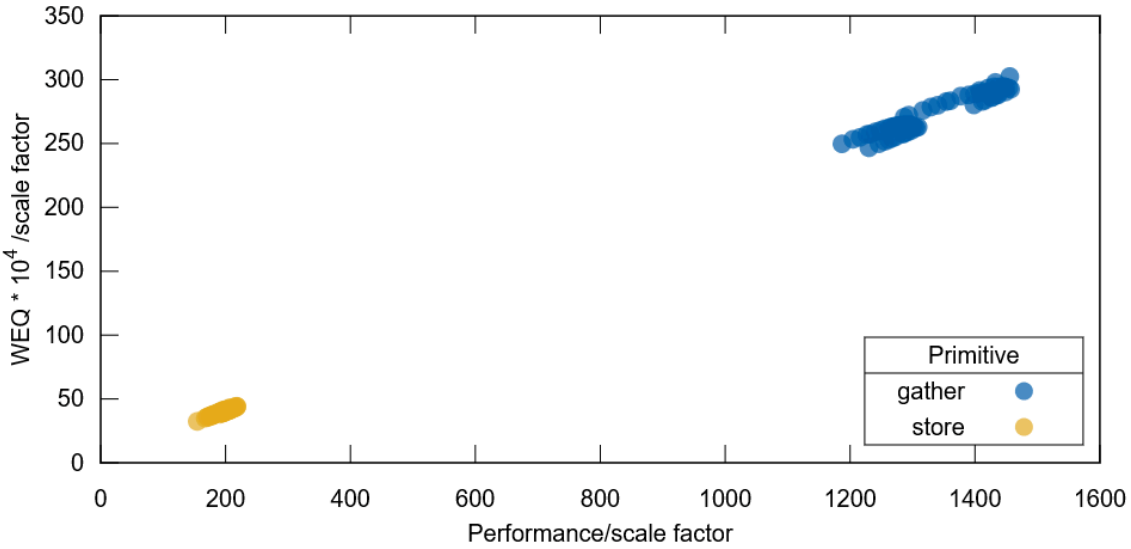


Figure 4.16: Scaled *WEP*s for *gather* and *store*.

Choice of Basic Profiles The main loop of a project operator does two things. First, a gather retrieves values from a column by using an index list, which was the result of a previous operator. Second, it stores the retrieved values sequentially into main memory. Hence, two basic profiles are required: one for the *gather* and one for sequential memory write access. Figure 4.16 shows both of these profiles for our test system A. They are already normalized and scaled as described in the previous section. The graph already shows that, on our test system A, the project operator is obviously dominated by the random memory access of the *gather* primitive.

Adjustment Function f The project operator itself has no parameters, which would change its behavior, i.e. there is neither branching nor masked operations. Thus, no adjustment function is required.

Weighting Factors w_j In each iteration of the main loop of the project operator, exactly one *gather* and one *store* operation is called. This results in the weighting factors $w_0 = 1$ and $w_1 = 1$.

Evaluation We executed the approximation and computed the upper hull of the approximated profile. Then, we checked where the configurations of this approximated optimum were in the measured profile. In Figure 4.17, the measured profile is shown. The optimal configurations, which resulted from the approximated profile are highlighted with red squares. Almost the whole performance range is covered by these configurations. Further, the configurations with higher cpu frequencies, which show a decreasing performance and *WEQ*, are not part of our approximated optimum. This is especially important, because increasing the cpu frequency is a common, but not always helpful, tool to increase the performance. The only limitation exposed by our approximated optimum is that it did not recognize the slightly higher performance of AVX512 over AVX2.

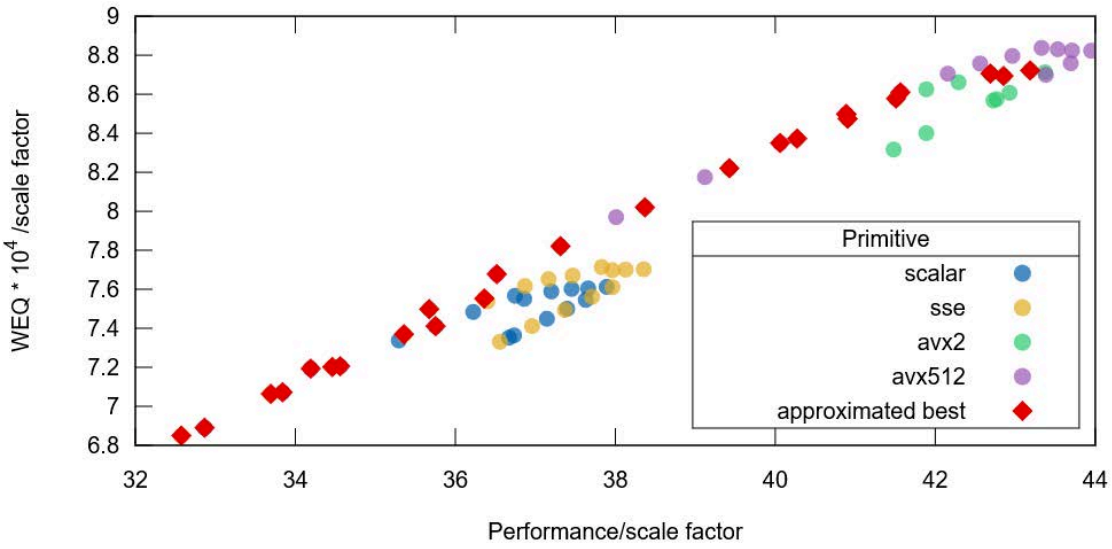


Figure 4.17: The measured *WEP* of a project operator. The configurations, which were optimal according to an approximated *WEP*, are highlighted.

Run Length Encoding

For this example, we will use the classical vectorized RLE compression without conflict detection as described in Chapter 3.1.2. We made this choice, because in the classical algorithm the amount of memory accesses scales non-linearly with the run length, which leads to the requirement of an adjustment function. The vectorized RLE algorithm itself is fixed, but the input data determines the actual execution behavior. In that regard, two extremes arise, whereas each execution is interposed there-between. One extreme is obtained if there is no run in the input data at all (run length equals 1). In this case, RLE-compressed data is twice as large as the original data, because for each array element, a run length of 1 is additionally stored. For the processing, this means that our vectorized compression algorithm performs essentially random reads and random writes with a ratio of 1:1. Random reads, because we always read 4 elements in each iteration, with 3 of them already being read in the previous iteration. The ratio is 1:1 because we always read, process and write once in each iteration. The second extreme occurs when each element in the uncompressed data array equals a single value (run length equals number of elements). In this case, RLE-compressed data consists of two values, the single value and the number of elements as run length and these two values are written once by the compression algorithm at the end of the last iteration. Thus, the read/write ratio approaches 1:0, while the read accesses are still random. Furthermore, between the reads and writes, there is also the actual compression which is a computation bound work. If this computation is slower than the I/O accesses, the memory access pattern is not the bottleneck for the performance anymore, but the computation itself. To summarize, the following three kinds of primitives are used within our vectorized RLE compression as well as decompression algorithms: (1) load, (2) store, and (3) compute (comparison primitives). Depending on the input data, these operations are composed differently. While the number of compute operations per read operation is constant, the ratio between read and write operations changes depending on the average run length. Hence, the profile for a specific average run length is within the spectrum between read-bound and write-bound operations.

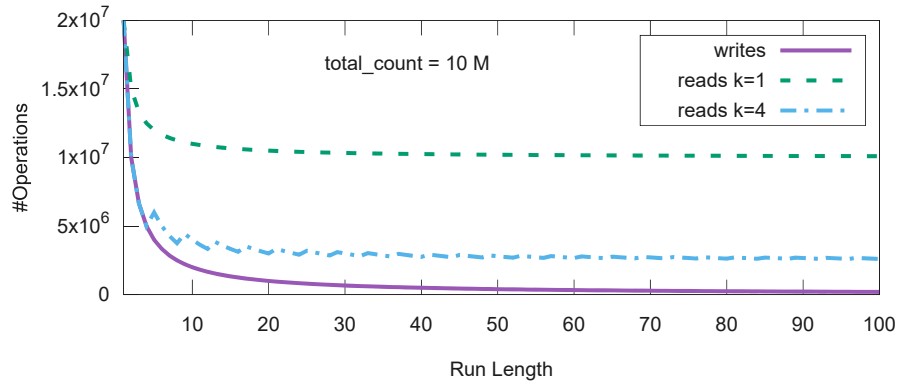


Figure 4.18: The number of read and write operations as function of the run length.

Choice of basic profiles We have already identified the three kinds of low-level operations used in the vectorized RLE compression. Now, we need to decide which *processing* profile represents these operations best. For every read operation, there is a fixed amount of compute operations. Thus, we use a processing profile for representing both operations. The cost of the operations per read access determines the exact processing profile, i.e. the read compute ratio. In our implementation for the ODROID-XU3, the main loop uses five operations in every step (additions, comparisons, and conversions), while every step loads two registers, one with the current value and one with the values which are compared. The cost of the compute and compare operations we use, are roughly the same. This sums up to five compute operations for every two read operations. Of course, it is a non-acceptable overhead to produce *WEPs* for each possible ratio. We built profiles with seven different ratios. The closest that one of these profiles gets to our preferred ratio of 5:2 is 6:2. The write operation is represented by a write primitive. The weight factor of the two profiles is determined by the ratio between read and write operations.

Adjustment Function f For getting an idea of how our profiles scale, we need to know how our limiting factors scale, i.e. the read and write accesses. Since the average run length of our input data defines the number of read and write accesses, and therefore the scaling of our profile-primitives, we need a description of these I/O accesses depending on the run length. We can then use this description for adjusting our primitive profiles according to the input data. For RLE compression, the functions for read and write accesses depending on the average run length can be built explicitly.

Assume a sequence of run lengths $RL = [rl_0, rl_1, rl_{|RL|-1}]$ with $count_{runs} = |RL|$ runs. For each run, there are two write operations, one for the value and one for the run length. Hence, the function for the write operations is

$$count_{writes} = 2 \cdot count_{runs} \tag{4.4}$$

The function for the number of reads depends on more parameters. One of these parameters is the vector width k. For a scalar c-implementation, k equals 1. For a NEON-implementation, which stores 4 values in one vector register, k equals 4. In every run, there is one scalar read for the value of the run followed by $\lfloor \frac{rl_i - 1}{k} \rfloor$ vector reads. The last vector read contains the end of the run and overlaps with the scalar read and possibly also the vector read of the following run. The final function for the number of reads is

$$count_{reads} = \sum_{i=0}^{|RL|-1} \left(2 + \left\lfloor \frac{rl_i - 1}{k} \right\rfloor\right) \tag{4.5}$$

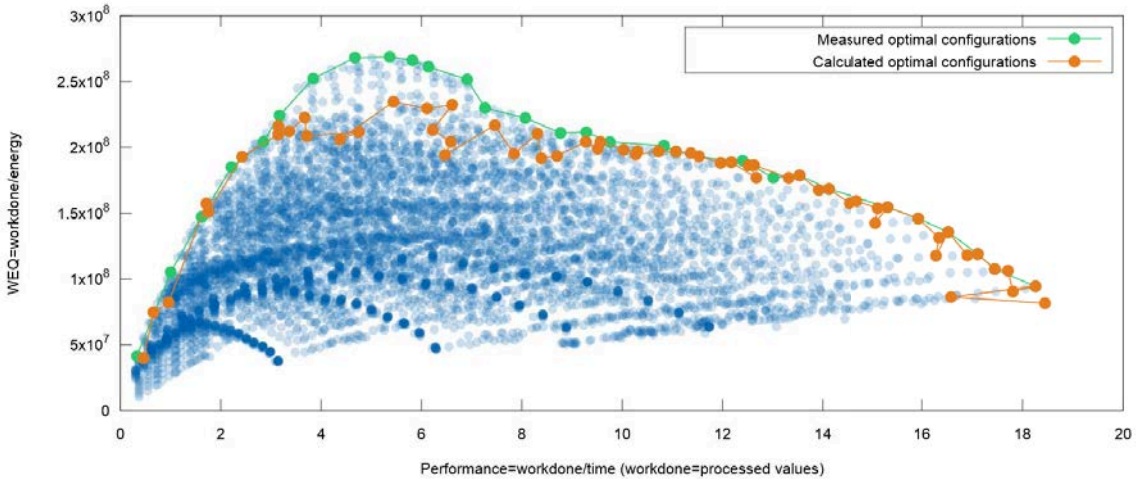


Figure 4.19: A benchmarked *Work-Energy-Profile* for RLE compression on the ODROID-XU3 with average run length of 45. The approximated optimal configurations are highlighted in orange, the actual optimum is highlighted in green.

For constant run lengths this iterative function can be rewritten as the explicit function

$$count_{reads} = (count_{runs} - 1) \cdot \left(2 + \left\lfloor \frac{rl_{const} - 1}{k} \right\rfloor\right) + \left(2 + \left\lfloor \frac{rl_{last} - 1}{k} \right\rfloor\right)$$

Note that we assume a fixed total number of data elements. Thus, the last run might be shorter than the other runs.

Figure 4.18 shows $count_{reads}$ and $count_{writes}$ for different run lengths. Both functions are rational. Hence, our function f must be rational, too. Since the exact weight factors are determined separately and there are no polynoms in either $count_{reads}$ or $count_{writes}$, we use the most simple rational function $f(Performance, WEQ) = (1/Performance, 1/WEQ)$.

Weighting Factors w_j The weighting factors w_j describe the influence of the primitive profiles. For RLE compression, we use two profiles and the weights w_0 and w_1, which are bound by the read and write operations. The ratio between the read and write operations are the factors w_0 and w_1. From Equation 4.4 and Equation 4.5 follows $w_0 : w_1 = 2 \cdot count_{runs} : \sum_{i=0}^{|RL|-1}(2 + \lfloor \frac{rl_i - 1}{k} \rfloor)$. For the decompression this ratio is inverted, since it has to read what the compression wrote.

Evaluation For being able to validate the results of the profile approximation, we also benchmarked the profiles for RLE compression. This way it is possible to see which performance and *WEQ*, the approximated configurations really produce. Most of the configurations in a profile will not be used since there are other configurations with the same performance but a higher *WEQ*. Hence, the useful configurations are the ones which are part of the upper hull. For comparing the quality of an approximated profile to a benchmarked profile, we filtered the configurations which are part of the upper hull of the measured and the approximated *WEP*. For implementing our approach, the weighting factors w_j, which depend on the average run length and the width of the vector registers k, must be found. For finding the weighting factors, we apply Equations 4.4 and 4.5. For the average run length of 45 and 128-bit vector registers (k=4) we get a read write ratio of $\approx$ 7:1. The ratio for the decompression is inverted, i.e. 1:7. Then we apply Equation 4.3.

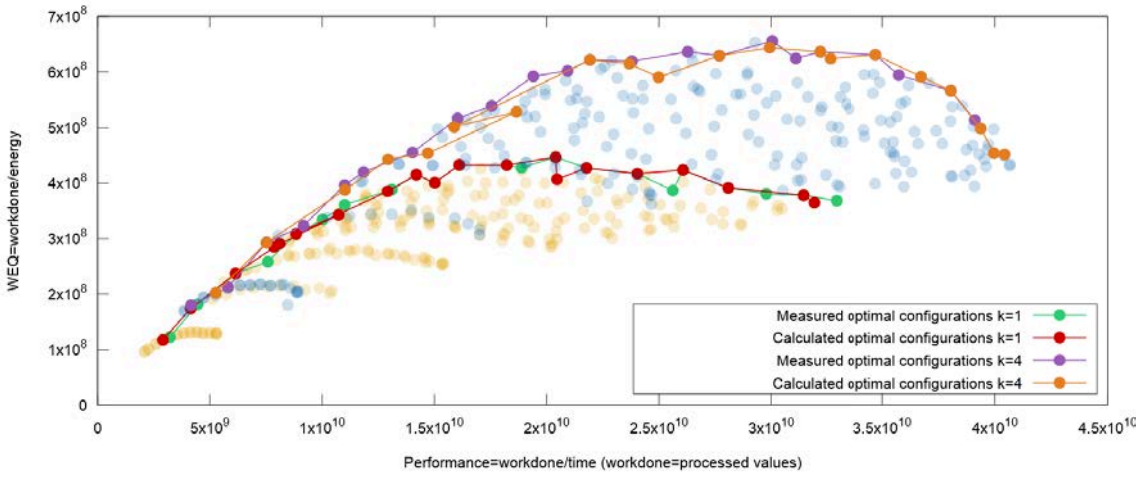


Figure 4.20: A benchmarked *Work-Energy-Profile* for RLE compression on test system B with an average run length of 500 and k={1,4}. The calculated optimal configurations and the actual optimum is highlighted.

The result is an approximated work energy profile. Afterwards, we select the configurations which are part of the upper hull of the newly found profile. Figure 4.19 shows the measured profile and the configurations of the approximated upper hull on our test system C. The graphic shows that the match is not exact but close to the optimal solution. This upper hull still contains many different core configurations which are often changing. But when applying a profile and changing hardware configurations, it is also necessary to consider the effects of context switching. This is why there is a clean-up phase after the upper hull is detected to find the robust configurations. In this phase, we eliminate configurations which interrupt a sequence of the same core configuration. The resulting configurations could now be used for configuring a system with regard to performance and energy balancing.

To show the general applicability of our approach, we also apply it on our test system B. The process was the same as for test system C. First, we benchmarked the basic write and the processing profiles. Then, we calculated the read/write ratio from Equations 4.4 and 4.5, applied the approximation Equation 4.3, and filtered the configurations which are part of the upper hull. For evaluating the result, the real profiles for compression and decompression have also been measured. Fig. 4.20 shows the benchmarked compression profile with and without vectorization (k=1 and k=4) for the average run length of 500. Additionally, the measured upper hull and the configurations of the approximated upper hull are illustrated. The graph shows a close match between the benchmark and the approximated configurations with only a few outliers.

4.4 BALANCING PERFORMANCE AND ENERGY FOR VECTORIZED QUERY PROCESSING

In the previous Sections, we explained how to benchmark primitive *Work-Energy-Profiles (WEPs)*, and how to use them to generate the *WEPs* for different and more complex operators. In this Section, we finally discuss different applications of these profiles to balance performance and energy-efficiency. We regard two fundamentally different scenarios:

1. System profiles are used to optimize a continuous but varying workload on a whole system.
2. Thread profiles are used for optimizing individual queries running on dedicated CPU cores.

We will describe the first case in Section 4.4.1 and the second case in Section 4.4.2. Finally, we will give a description of how to use the second approach to optimize query execution in Section 4.4.3.

4.4.1 Continuous System Workload

The first approach, which we applied, is the use of system profiles to determine the optimal core usage and frequency for a defined workload. In [UKM+16], we show this approach in a demonstrator on our heterogeneous test system C, which offers an especially large configuration space. Our workload consists of sequential memory reads and compute operations. The ratio between memory reads and computations as well as the amount of requests is user-defined. This workload shows the characteristics of some frequently used operations, i.e. scans with differently complex predicates. For the demonstration, all operations are executed in scalar mode.

The adjustment of the configuration is executed in a loop and is purely reactive. For performance reasons, the loop is not implemented as a busy waiting-loop but with system signal handlers, which are called after each power counter update. This loop, also called *energy control loop*, watches 3 quantities: (1) The read-compute-ratio, (2) the average query latency, and (3) the length of the query queue. Whenever the energy control signal handler is called, the following steps are executed: At first, a processing profile, which fits the current read-compute-ratio is chosen. Second, the real average query latency is compared to a user defined latency-threshold, which serves as a soft constraint. This means that it might be exceeded, but only temporarily. If this threshold is exceeded, a higher performing configuration out of the hull of the chosen processing profile is chosen. If the average query latency is below the threshold, a lower performing configuration can be chosen. The choice of the configuration is initially done in small steps, which always chose the next best resp. worse performing configuration. The length of the query queue is recorded after each configuration step. If it is increasing, the configuration steps are doubled, e.g. after three steps with an increasing queue length, the next 8 better resp. worse performing configurations are skipped. If it is decreasing, the configuration steps are decreased in the opposite way until the step width is down to one configuration. A constant queue length keeps the step width at the same level. However, the latter case rarely happens in our demonstrator. This approach causes the latency to level out around the user-defined soft constraint.

Figure 4.21 shows the user interface of the demonstrator. When the screenshot was taken, a latency-threshold of 1 s was defined, while the measured average latency is at 1.38 s. Hence, the measured latency is higher than the threshold and during the next iteration

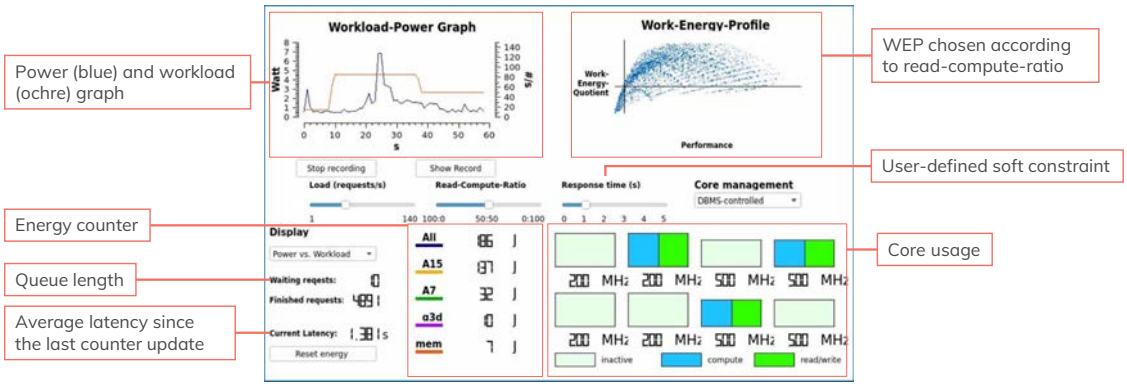
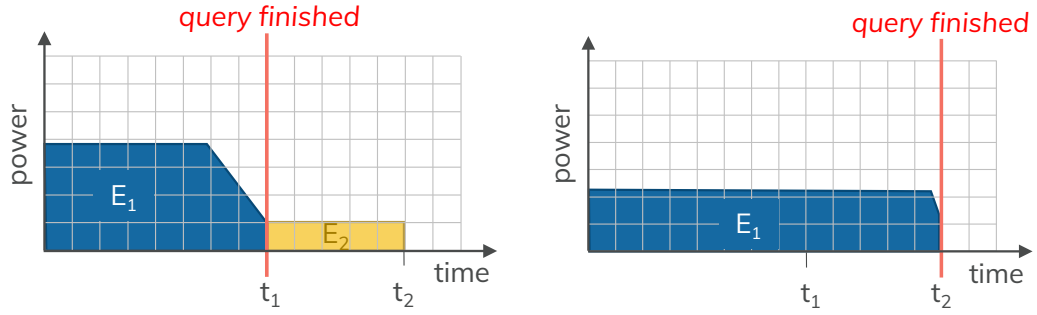


Figure 4.21: User interface of a demonstration of the use of *WEPs* for continuous workloads.



Configuration 1 (race-to-idle) Configuration 2

Figure 4.22: Comparison of power and energy of two different configurations with the same *WEQ* but different performance. Race-to-idle is not the best choice in this case if the goal is to save energy.

of the *energy control loop*, a higher performing configuration will be chosen. The current configuration is shown in the lower right quadrant of the user interface. In this example, the big cluster is running one core at 200 MHz and the little cluster is running two cores at 500 MHz. The remaining cores idle at the respective cluster frequency. The reticle in the *WEP* shows the position of this configuration in the profile. In this case, most other configurations perform better than the currently chosen one. As shown in most other profiles, this configuration is not the most energy-efficient one either. However, if the system is underutilized, i.e. if there are not enough requests to reach a higher performance, this configuration makes sense.

4.4.2 Individual Requests

A system, which is used exclusively by a DBMS is not the only possible scenario. Another scenario is a system, which processes individual requests, while other applications are running on the same hardware. In these cases, the DBMS can only use and control a subset of the system resources, i.e. not all cores. The configurations of a system *WEP* cannot be selected in such scenarios. However, each thread can still choose among the configurations of a thread profile. This includes the instruction set, and the CPU frequency, if the system offers this option. Like for the continuous workload, a threshold

can be defined by the user. Then, a configuration, which satisfies this threshold, can be selected. However, since the workload is not continuous, a reactive loop is not necessary. Instead, for each request, the configuration is set before running it. The required parameters to select the appropriate thread profile, e.g. utilization of the other cores, can be derived by using operating system tools. This step is important, because as we have shown in example 3 (project operator) in Section 4.2.4, the load on the remaining system influences the behavior of an observed thread significantly.

A common approach in such scenarios is called race-to-idle. In this approach, requests are finished as fast as possible, such that the processing cores can return into a sleep state or idle at a low frequency as often and as long as possible. This requires to always select the best performing configuration. However, this is not necessarily the most energy-efficient approach. Figure 4.22 shows an example explaining this statement. It illustrates the power graph during the execution of a request for two different configurations. Both configurations result in the same used energy E_1 for the query execution, i.e. the *WEQ* is the same, but their performance is different. Such configurations exist in each profile, which shows a plateau or a decreasing *WEQ* after a peak has been reached. Configuration 1, which finishes at t_1, performs better than configuration 2, which finishes at t_2. In a race-to-idle approach, configuration 1 would be chosen. This assumes that the overall energy consumption is lower than in configuration 2 because of the long sleep time of the used CPU. But even a CPU in a deep sleep mode still draws a small amount of power as long as the system is not switched off completely. The integral of this power over the idle time is the energy consumption E_2. This energy adds to E_1 for the overall energy consumption, while in configuration 2 the overall energy consumption equals E_1 only. If configuration 2 had a higher *WEQ* than configuration 1, the difference would be even higher. Only in the case that configuration 2 had a lower *WEQ* than configuration 1, there is a possibility, not a guarantee, of using less overall energy with configuration 1. For this reason, we argue to always choose a configuration, which only performs as fast as necessary, not as fast as possible.

4.4.3 Workflow for Query Execution

Despite being developed and tested for in-memory query processing engines, the scenarios mentioned above are very generally applicable to different applications. In this section, we describe the specific steps to optimize the execution of individual queries, which run on a dedicated CPU core. This is the approach, which we will also use for the end-to-end evaluation in the next chapter. As shown in chapter 2.3.1, there is a significant optimization potential, when the instruction set of a query is optimized on an operator level instead of the query level. In a combination with the optimal CPU frequency selection, this optimization potential increases even more. Both of these properties, the instruction set and the CPU frequency, can be determined using *WEPs*. To do this, several steps during setup time and query compile time are required. For the sake of simplicity, we will focus on operators in our explanations. However, the same techniques are applicable for compression algorithms.

Setup time To minimize the overhead during query compilation and runtime, the profile approximation should be done during setup time, or whenever a new operator is added. For the approximation, the weights and basic profiles have to be determined for each operator. Fortunately, the basic profiles do not depend on the amount of data, which has to be processed. They only depend on the used primitives and the location of the data in the memory hierarchy. Since we are developing for in-memory query execution, we expect our data to be in main memory. The used primitives can easily be

counted by running each operator with a low amount of input data. If there are parameters, e.g. the selectivity, each operator is run multiple times with input data reflecting these parameters. During these runs, each primitive counts how often it is called. When these operators are run with larger input data sets, only the absolute amount of primitive calls changes, but not the ratio between them. For instance, a simple filter with a selectivity of 10%, which sequentially scans the input data, will always show a read-write ratio of 10:1 and a read-compare ratio of 1:1 unregarded of the amount of input data. This means that there is one write call for 10 read calls, and one comparison for each read call. With these weights and the specification of the used primitives, the approximation of the *WEPs* for the operators can be done. After the approximation is done, the upper hull is filtered from the retrieved *WEPs*, because the remaining configurations are never optimal anyway. This procedure is done for different system loads.

Query compile time We assume, that the preferences of the user are known at query compile time, i.e. if he wants to optimize for performance, energy-efficiency, or a defined performance threshold. Additionally, a rough estimate of the system load is gathered by using tools of the operating system. Depending on this load and the user preferences, an according configuration is selected from an approximated profile for each operator. For instance, if the user wants a performance optimization, the configuration with the highest performance is selected. This configuration contains the instruction set and the CPU frequency. Our query is then translated into C++ code from an intermediate representation, e.g. MAL [8]. During this translation, the processing style and the CPU frequency for each operator is defined in the code. Finally, the c++ code is compiled into an executable.

4.5 RELATED WORK

The importance of optimization for energy-efficiency for data management systems has already been identified in 2009[HSMR09]. The authors show that the most energy-efficient configuration is not always the one with the highest performance. For instance, they show that a scan on compressed data performs better than a scan on uncompressed data. While this was expected, the energy consumption is higher for the scan on compressed data. Although the experiment reads the input data from disk, while we read data from main memory, they observe the same counter-intuitive behavior as we observed on our test systems. Following from these observations, the authors argue, that hardware approaches alone are not sufficient to optimize for energy-efficiency. However, to develop a software-driven approach, two main questions have to be answered:

1. Where lies the largest optimization potential, i.e. which hardware components have knobs that can be used to save energy?
2. How can these knobs be tuned in a meaningful way?

These questions have been answered differently depending on the exact application scenario and the available hardware. The partitioning of data across disks is a powerful tool to vary the number of active disks, and therefore the energy consumption. This approach has been applied in the system E²DBMS [TWZX14]. However, we are considering in-memory systems in this work, but conventional main memory is not persistent. Hence, we cannot switch it off like disks to save energy. On the CPU side, E²DBMS adjusts the CPU frequency in a feedback loop, which checks for the compliance with a user-defined performance constraint. This approach is similar to our *energy-control-loop*. However,

[8] https://www.monetdb.org/Documentation/MonetDBInternals/MALReference, accessed 03/08/2020

we are using *WEPs* to adjust the CPU frequency, while E²DBMS assumes a monotonic relationship between the power level, which defines the CPU frequency, and the performance. As we have shown, for in-memory query processing, this assumption does not hold true for all systems and workloads. Hence, out of the two proposed components, we can only consider the CPU for optimization, but must find a different way than proposed in [TWZX14].

Tsirogiannis et al. analyzed the energy consumption of a database server and also found that the CPU is one of the three components, which make up the main share of a server's idle power [THS10]. The other two components are the main memory and the mainboard itself. While the power level of the main memory was fixed in the system, which was used in [THS10], the power level of the CPU increased significantly from 48 W to 160 W, when they were not idle anymore. Again, this shows the huge optimization potential of CPUs. This potential has been exploited in the system ERIS, which uses a variant of our *WEPs* in a hierarchical *energy-control-loop* [Kis17]. This control-loop runs on different levels of the system, i.e. there is a system-wide control loop, another one per node, and finally, one per core. While this approach worked well for workloads on a system level, it assumed the hardware to run only the database system, which has full control over all cores of the system. Additionally, it did not consider vectorization.

In the recent years, hardware development brought up main memory with control knobs similar to those of CPUs, e.g. frequency scaling. This sheds new light on its optimization potential. Appuswamy et al. showed that main memory has become a promising component for energy-aware optimization [AOA15]. Since the properties, which can be adjusted, are mostly the same as for CPUs, we expect that the strategies for CPU optimization are well transferable to memory. However, this mainly concerns large server-grade systems, because a number of memory DIMMs is required to account for a mentionable part of the overall energy consumption. Smaller systems, which could perspectively be used for ubiquitous computing, e.g. in edge clouds, will hardly benefit from main memory optimization. For instance, our test system C shows the lowest energy consumption for the main memory out of all measured components, as can be seen in Figure 4.21.

The energy consumption of such heterogeneous single-board systems, like our test system C, has been analyzed and optimized by Mühlbauer et al. [MRS+14]. The authors used HyPer as a database system on an ARM® big.LITTLE™and generated an estimation of the performance and energy consumption out of microbenchmarks for different queries. Their main control knobs were the selection of the CPU cluster and its frequency. Unlike in our test system C, the system of the authors was only able to apply cluster switching, i.e. only one cluster could be used at a time. Further, the approach is specific for the used hardware. It is unclear if it shows the same promising effects on a different hardware.

Summarizing, there are two main components, which show a significant optimization potential: the CPUs and the memory, may it be disc or main memory. Discs are not relevant in our in-memory applications and the control knobs of the main memory are similar to those of the CPU usage. Therefore, we focus on optimizing the CPU.

The question of how to optimize the CPU requires a more differentiated answer. Generally, there are three different approaches: (1) analytical, (2) benchmark- or monitoring driven, and (3) analytical models based on benchmarks. An example for the first approach was proposed by Xu in 2010 [Xu10]. This approach used the power specifications of hardware vendors to build an analytical model estimating the power consumption of a workload. However, these specifications turned out to be unreliable, which complies with our explanations in Section 4.1. Hence, when E²DMBS was proposed, which Xu was a co-author of, a feedback loop was applied. This makes the frequency scaling of

E²DMBS an example for the second approach. Another example for the second approach is ERIS, which uses benchmarked profiles to adjust the CPU knobs while monitoring the actual performance. These profiles are either generated during an installation phase or dynamically when the system is running. The benchmark-based model in [MRS+14] is an example for the third approach.

Hence, the analytical approach does not seem feasible because of a lack of necessary information. The benchmark- or monitoring-driven approach requires either extensive benchmarking or fails to portray the behavior of current hardware correctly. To reduce the amount of necessary benchmarking, a combined approach makes sense. However, to the best of our knowledge, none of the existing approaches considers vectorization on arbitrary hardware platforms.

In this sense, we combine benchmarks of primitive profiles, which can contain vectorization as a configuration parameter, with an approximation model. These two components can be derived independently of each other. This means, that the benchmarks do not depend on the approximation, while the parameters of the approximation do not depend on the hardware-specific benchmarks.

4.6 SUMMARY

In this chapter, we explained the requirement for a benchmark based model to estimate energy-efficiency and performance. We introduced such a model, and call it *Work-Energy-Profile (WEP)*. It enables a mapping between performance and energy-efficiency. Further, we proposed an approach to estimate the profiles of complex applications by combining a number of basic *WEPs*. This efficiently reduces the time required for initial benchmarking. An evaluation using two fundamentally different examples showed that our approach is close to the optimal solution. Then, we gave an overview of how our *Work-Energy-Profiles* can be used in practice. Finally, we closed with a discussion of related work concerning energy-efficiency on database systems.

END-TO-END EVALUATION

So far, we have examined the challenges and chances of vectorization. We learned, that the chosen instruction set and the vector width are a crucial choice, but not a trivial one. To enable this choice in a way that the code base can stay the same, we developed a library, the *Template Vector Library* (*TVL*), which allows to set the instruction set, vector size, and base data type as a template parameter. Based on this *TVL*, we presented an optimization technique, which allows optimizing for performance and energy-efficiency likewise. The evaluation of this technique was done on an individual operator and on an individual compression algorithm. In this chapter, we optimize analytical queries combining a number of operators. First, we introduce the setup for this end-to-end evaluation. Then, we show the results for the applied *TVL* and use this as a basis to explore the optimization potential when choosing the instruction set on an operator granularity. Then, we apply our optimization technique to differently used systems and to uncompressed and compressed data and discuss selected results. Finally, we will compare our results to MonetDB, a state-of-the-art column store.

5.1 EVALUATION SETUP

Software Setup For this evaluation, we use MorphStore[DUP+20]. MorphStore is an in-memory query execution engine for analytical queries, which uses the *TVL* to implement all its operators and compression algorithms. Therefore, the same code base can be used for all available instruction sets. So far, query execution in MorphStore is single-threaded. However, any changes of the CPU frequency are initiated by a separate thread on a separate NUMA node to not interfere with the query execution. On test system A, we compiled our code with g++ version 8 with the optimization flags -O3 and -flto. On test system D (see below), we used g++ version 10.0.1 with the same optimization flags.

The Star Schema Benchmark To show the applicability of our approach to complex analytical queries, we apply the well-known Star Schema Benchmark[OOC07], which consists of 13 analytical queries. In order to do this, we generate synthetic data using a data generator provided by Lemire[1]. The size of the test data is determined by the scale factor (SF), where a scale factor of 1 generates about 800 MB of data and a scale factor of 10 generates about 7.6 GB. To enable numeric processing of the whole data set, we use dictionary encoding on all non-integer columns. Further, all non-integer predicates in the queries are replaced by their respective dictionary keys. If not mentioned otherwise, we do not apply other compression techniques. Since MorphStore is solely an execution engine, i.e. it does not provide a query optimizer or a parser, we use MonetDB [IGN+12] for these preceding steps. We retrieve the query execution plans from MonetDB in the form of MAL plans [2] and translate them into MorphStore C++ code. For this translation, a few changes to the original plan are necessary. First, we omit the `Order By`-clause because MorphStore does not support sorting, yet. Second, we introduce projections where candidate lists would have been used in MonetDB.

Hardware Setup As the first test hardware for the end-to-end evaluation, we have chosen our test system A, which offers all Intel vector instruction sets. The second system we chose is an ODROID-C2, which we will denote as test system D. This switch away from the ODROID-XU3 (test system C) was necessary, because the Star Schema Benchmark, which we will be running, requires instructions on 64-bit, which is not provided for all necessary operations in the NEON version offered by the ODROID-XU3. The ODROID-C2 features four ARM® Cortex®-A53 cores running at a maximum of ≈1.5 GHz and 2 GB DDR3 main memory. The main features of this system are summarized in Table 5.1. If

[1] https://github.com/lemire/StarSchemaBenchmark, accessed 13/07/2020
[2] https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference, accessed 13/07/2020

Core Description	ARM-Cortex-A53
Number of Cores	4
Hyperthreads per core	1
Frequencies	0.1 GHz - 1.536 GHz
Frequency Steps	150 MHz, 250 MHz, 500 MHz, 296 MHz, 240 MHz
Number of Freq. Steps	6
SIMD Extensions	NEON
L1, L2, L3 Cache	32 kB + 32 kB (instruction + data), 512 kB (shared), -

Table 5.1: Configuration options of the ODROID-C2 (test system D).

not mentioned otherwise, the thread executing a query is the only thread running on a system. Hence, there are no resource conflicts caused by concurrent threads. For all experiments, which do not explicitly apply our optimization, we use the default CPU driver of each system in performance mode to care for the frequency scaling. As we have shown in Chapter 4, the powersave governor of the *acpi* CPU driver does not provide acceptable performance. Hence, we do not consider the *powersave* governor for test system D, which applies the *acpi* driver. The *powersave* mode of the *pstate* driver behaves roughly like the *schedutil* and *ondemand* governors[3]. While we did not have the chance to test the *schedutil* governor, the *ondemand* governor failed to provide sufficient energy-efficiency in our test in Chapter 4. Therefore, we also do not consider the *powersave* mode of the *pstate* driver for test system A.

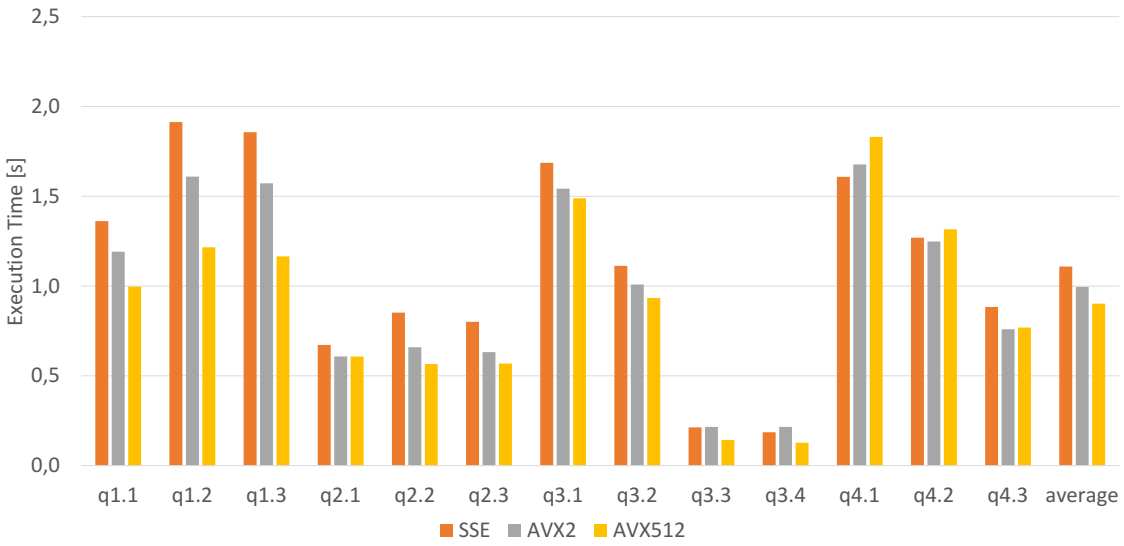
Optimization Knobs In Section 5.3 we will show selected results of our optimization. This optimization is done per operator, i.e. the code for a query execution is constructed from several individually optimized operators. This means that each operator of a query can be executed with an individual instruction set and vector length. Further, the CPU frequency can be changed on node or on system level before each operator. The choice of the instruction set and the CPU frequency is done according to our optimization technique as presented in Section 4.4.3.

5.2 OPTIMIZATION POTENTIAL OF VECTORIZATION

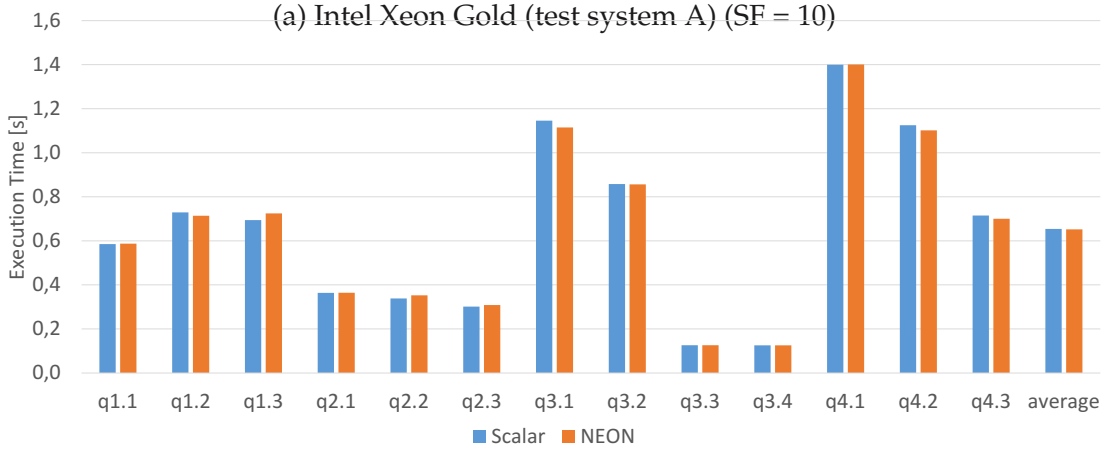
In a first series of experiments, we use the *TVL* to execute the SSB in MorphStore with different instruction sets. Our two test systems (A and D) offer different instruction sets. While test system D offers NEON as the only vector instruction set, test system A offers SSE, AVX2, and AVX512. To keep it simple, we chose to only use the native vector sizes of AVX2 and AVX512, although they also offer instructions on smaller vector registers. Hence, we consider the following processing styles:

- **System A:**
  - scalar < v64 < uint64_t > >
  - sse < v128 < uint64_t > >
  - avx2 < v256 < uint64_t > >
  - avx512 < v512 < uint64_t > >
- **System D:**
  - scalar < v64 < uint64_t > >
  - neon < v128 < uint64_t > >

[3] https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html, accessed 27/07/2020



(a) Intel Xeon Gold (test system A) (SF = 10)



(b) ARM Cortex-A53 (test system D), (SF = 1)

Figure 5.1: Star Schema Benchmark results on different systems for fixed processing styles.

We first used the same processing style on all operators of a query. Then, we used different processing styles for operators working on base data and for operators working on intermediates only. In both cases, the CPU frequencies are managed by the CPU driver.

5.2.1 TVL for Analytical Queries

In our first runs of the SSB, we use the same instruction set for each operator of a query. We run the benchmark on test system A with a scale factor of 10. On test system D we could only run it with scale factor 1, because there is only 2 GB of main memory, which could not fit all base data and intermediates for larger scale factors. The execution time of the individual queries of the SSB are pictured in Figure 5.1.

Figure 5.1(a) shows the results for the vectorized execution of test system A. These are the results, which we already partially used in Section 2.3.1 to motivate the importance of the right choice of the vector register size. The depiction of all queries of the SSB show, that the largest vector register size, achieved with AVX512, is often beneficial for the performance, but not in all cases. In the queries 4.1 and 4.2, AVX512 even shows the worst performance. A closer look at the individual operators of these queries shows the reason for this behavior. These queries are heavily dominated by a join and the performance of

base data \ intermediates	Scalar	SSE	AVX2	AVX512
Scalar	1	0.96791	0.95683	0.94941315
SSE	0.99605	0.92731	0.94743	0.9367778
AVX2	1.08536	1.03665	0.99002	1.00316438
AVX512	1.51896	1.44266	1.40313	1.3630008

(a) SSB query 1.1

base data \ intermediates	Scalar	SSE	AVX2	AVX512
Scalar	1	0.93786435	0.95021031	0.94617176
SSE	0.85047707	0.86999579	0.85613411	0.8728028
AVX2	0.8183232	0.81304241	0.78575679	0.83494329
AVX512	0.79608686	0.77981009	0.77083946	0.81790197

(b) SSB query 4.1

Figure 5.2: Speedup over scalar execution for different instruction sets for operators on base data and operators on intermediates.

this join makes up the major part of the overall performance. However, the specific hash join used in MorphStore only profits from AVX512 on test system A if the cardinality of the result set is only a fraction, i.e. smaller than 1/3rd, of the cardinality of the input. For higher cardinality ratios, vectorization is not beneficial at all, but scalar processing shows the highest performance. The exact value of this ratio can differ between systems, which is just one reason, why we argue for our benchmark-based optimization. Another interesting insight is the behavior of the queries 3.3 and 3.4. These queries show a similar performance for SSE and for AVX2, although the latter offers wider vector registers. Only with the introduction of new instructions in AVX512, the performance increases. This supports our claim that wider vector registers also require new instructions to make full use of the increased data parallelism.

Figure 5.1(b) shows the execution times of the SSB queries for test system D. The NEON instruction set is limited when it comes to functionality. For instance, the result of a comparison between two vectors is always another vector. This introduces an additional overhead for creating a bitmask out of these vectors. For this reason, the use of NEON hardly shows any benefit over a plain scalar execution. A new vector instruction set by ARM, called SVE, is announced. It will feature a wider range of instructions and a scalable vector size. There is already a prototype implementation of the *TVL* for SVE [4], but because the according hardware is not yet available, it cannot be tested for performance. Since the optimization potential for vectorization is very small with the system, which is physically available for us at the time this thesis is written, we are not considering test system D for our optimization technique. However, test system A offers more instruction sets, which show a varying performance as we have shown. Hence, we will use this system for the following evaluation.

5.2.2 Combination of Instruction Sets

As explained in Section 2.3.1, it can be beneficial to use different instruction sets on different operators in the same query. To see if the use of different instruction sets also shows a different performance in reality, we combined two vector instruction sets in query 1.1 and 4.1 of the SSB. One instruction set is used for all operators working on base data and another instruction set is used for all operators working only on intermediates. The results for different combinations of instruction sets on test system A are shown in Figure 5.2. Although the intermediates are small compared to the base data, there is still a difference in the execution time when they are processed by different instruction sets. If the instruction set, which works on the base data is changed, this difference increases. Among our

[4] Available in a dedicated branch of the *TVL* repository: https://github.com/MorphStore/TVLLib/tree/simd-arm_sve, accessed: 15/07/2020

tested instruction sets, the highest performance for query 1.1 is achieved by a combination of AVX512 for operators on base data and scalar processing for operators on intermediates. Again, this can be explained with a closer look at the individual operators. Query 1.1 spends most time on selections on base data and on intersections on intermediates. While the selections involve mainly sequential read access on one column, the intersection requires access on two columns, i.e. it requires reading from two different memory addresses. Like for the join, this causes a performance loss when using large vector registers if the input-output cardinality-ratio passes a certain threshold. Query 4.1 does not profit from vectorization. Most of the execution time for evaluating this query is spent on a join, which performs best with scalar processing. Additionally, a second join operator, which requires less runtime, does also not benefit from vectorization.

Our experiments show that the choice of the instruction set and the according vector register size can influence the performance positively or negatively, whereas the best choice is not in all cases the trivial one. This is why we presented an approach to choose the appropriate vector extension, which we will evaluate in this chapter. As we will show, it is a feature of our optimization to detect counter-intuitive behavior without having to actually run all possible combinations of register sizes and cardinality ratios.

5.3 OPTIMIZATION APPLICATION

In order to show the universal applicability of our approach, we consider 3 exemplary scenarios for our optimization:

1. A query is executed in a single thread on an otherwise idle system. The CPU frequencies can be set for the whole system, because no resources are required by any other threads.
2. A query is executed in a single thread. There is no other thread running on the CPU core, the query is executed on, but the remaining cores of the same NUMA node and the cores on the neighboring NUMA node produce a constant memory I/O load. This simulates the effects of other queries running on the same system. The CPU frequencies can be set per NUMA node. Frequency setting itself is triggered by a separate thread running on another NUMA node, which is otherwise idle.
3. We consider our first scenario, but with compressed base data, which is decompressed on-the-fly during query execution, i.e. it is decompressed per vector register before this register is handed to the actual operator.

For all of these scenarios, we will show the results of the performance optimization and discuss selected noticeable details. Then, we have a closer look at the energy optimization for scenario 1. We will show that the best performing configuration is not in all cases the most energy-efficient one, and that our optimization is able to find such configurations at a minimal performance loss.

5.3.1 Performance Optimization

To optimize the performance of the SSB, we execute the steps explained in Section 4.4.3. In detail, the following steps are done offline: First, we create primitive profiles for each primitive class and subclass, which we use in our operators. There are two sets of profiles created: one for an otherwise idle system and one with a bandwidth saturated system. Then, we count the number of called primitives in each operator with a scalar processing style for small input sets of 10.000 values. For all operators, where the output cardinality

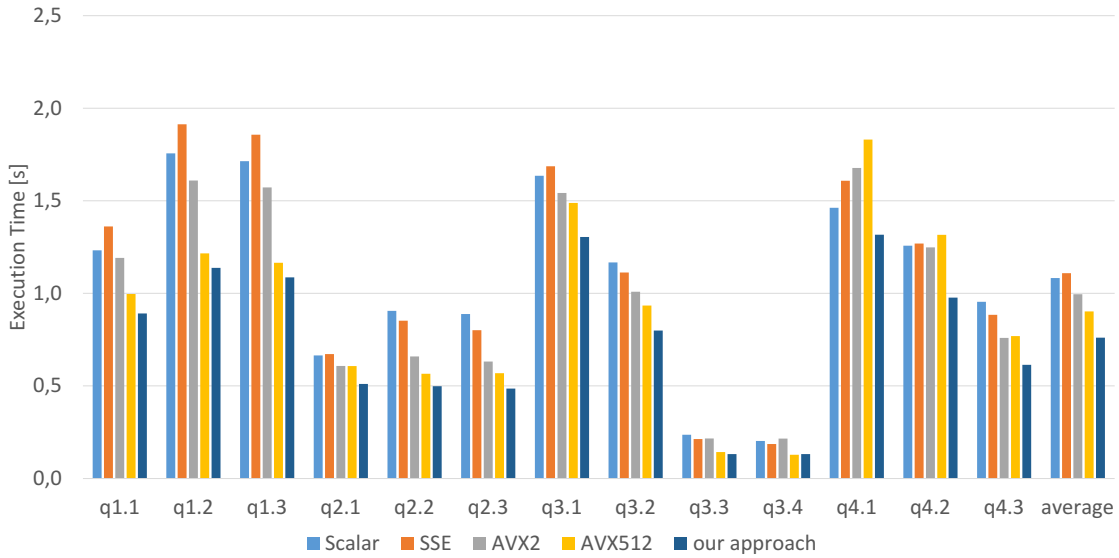


Figure 5.3: The results of our optimization for the SSB compared to the results for different fixed instruction sets (SF=10, Scenario 1).

can vary and be different from the input cardinality, e.g. joins and selections, we count the called primitives for different input-output-cardinality ratios. For instance, for the join we considered 5 different ratios between 1:0 and 1:1. We did not consider larger ratios, because the query plan optimization, which happens before our optimization, should and does avoid such constellations wherever possible. For masked I/O operations, we counted the number of actually read or written values. These counted calls serve as the weights for our profile approximation. We approximate the operator profiles for all used operators and for all considered cardinality ratios. Again, this is done for an idle system and for a bandwidth saturated system. Finally, we extract the upper hull of these operator profiles and sort them by performance, such that the last configuration in the list is the one with the highest expected performance. These last configurations are used during query translation to set the processing style and define the optimal CPU frequency for each operator. We assume that the cardinalities are already estimated for the purpose of query plan optimization and we can use this estimation. In the following, we will show the results of this process.

Scenario 1: Idle System

In the first scenario, there are no resource conflicts with other threads and we can set the frequency on a system level. We apply or optimization approach to the SSB with a scale factor of 10. Hence, we set the processing style and the CPU frequency according to the result of our operator profile approximation. Figure 5.3 shows the resulting execution time of the individual queries. The graph also compares them to the execution times of different fixed instruction sets with the *pstate* CPU driver managing the CPU frequencies. We reach a shorter execution time in most queries and a similar execution time in one query (query 3.4). In no case was our approximation significantly slower than the best-performing variant with a fixed instruction set.

Generally, for most operators, our approach selects AVX512 as the appropriate instruction set. Only for 2 to 3 operators per query, i.e. never more than 15% of all operators, scalar processing is selected as shown in Figure 5.4. However, these operators include some joins, which take up most of the processing time of the queries. Therefore, this selection is important, and we can observe a speedup of up to 1.4 compared to pure AVX512. Especially the queries 4.1-4.3 profit from our instruction set selection, although less than 10%

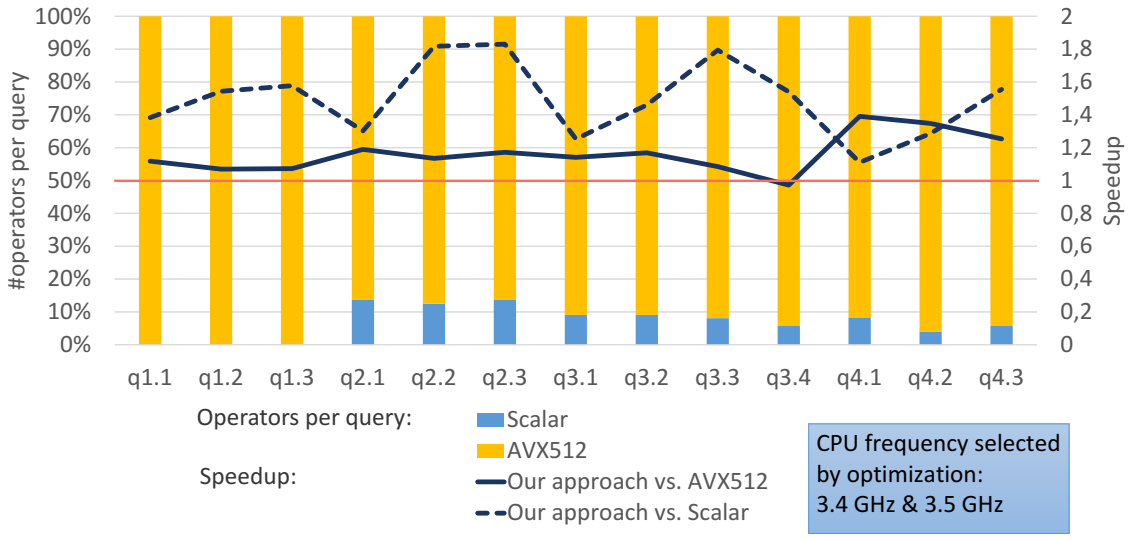
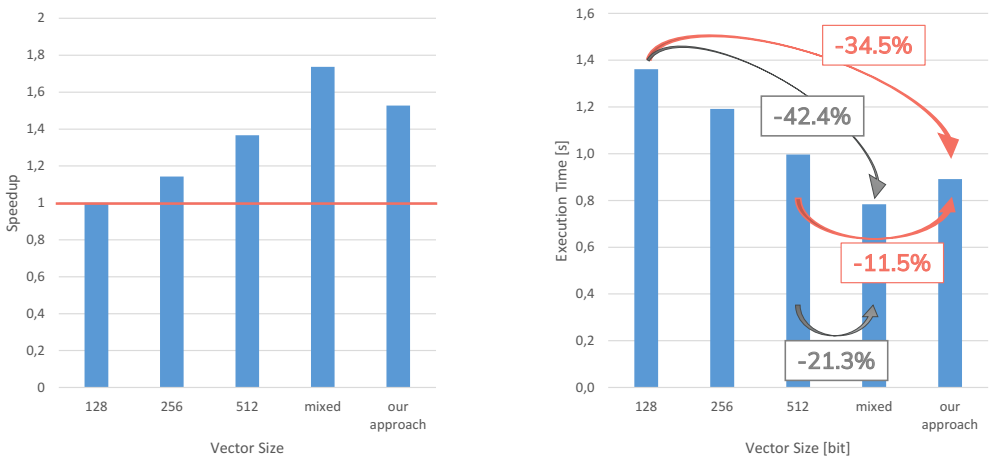


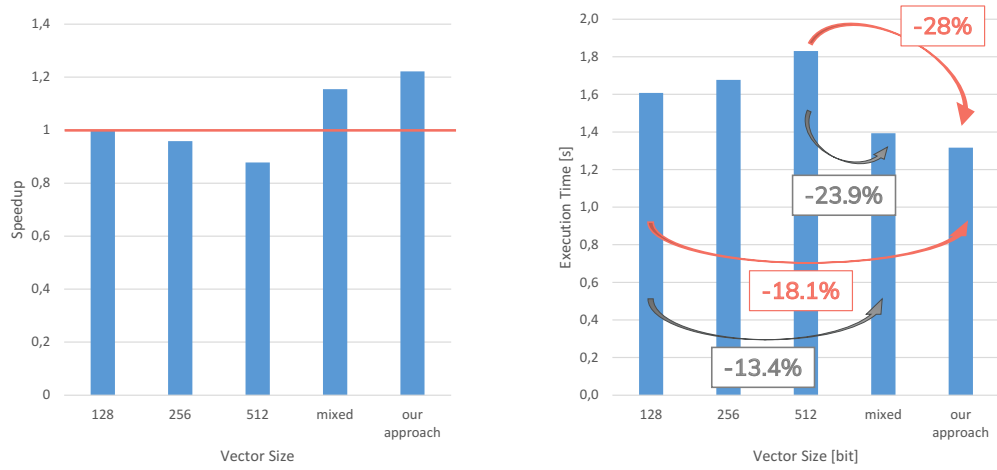
Figure 5.4: The instruction sets as chosen by our optimization. SSE and AVX2 were never chosen. The optimal CPU frequency as proposed by our approach was at 3.4 GHz resp. 3.5 GHz. (SF=10, Scenario 1). We observe a speedup compared to scalar execution and to AVX512 when applying our optimization.

of the operators of these queries are executed in scalar mode instead of using AVX512. Additionally, the frequency is set to 3.4 GHz or 3.5 GHz for all operators. This leads to a further speedup, which we can especially observe in the queries 1.1-1.3, where only AVX512 is selected by our optimization. There are also queries, where AVX512 with automatic frequency scaling by the CPU driver is already optimal. In the SSB, query 3.4 shows this behavior. In chapter 2, we found that a few operators perform best using SSE or AVX2. Our results show that these instruction sets play only a minor role to optimize for performance in scenario 1. We suppose that the explicit setting of the CPU frequency is responsible for this game change.

In Figure 5.5 we show the results of query 1.1 and query 4.1 in more detail. The graphs show the speedup compared to SSE's 128-bit processing on the left side, and the execution time on the right side. The "mixed" bar shows the theoretical optimum if the fastest vector size and the according instruction set is chosen per operator. Unlike our approach, manual CPU frequency scaling is not considered for the "mixed" scenario. In query 1.1, a wider vector register size is beneficial for the performance. However, the mixed scenario shows that there is still some optimization potential when selecting the vector size per operator. Our approach finds such a configuration, which is performing even better than the plain AVX512 approach. We could achieve a reduction of the execution time by 11.5% compared to AVX512. Compared to SSE, the reduction is at 34.5%. In query 4.1, AVX512 shows a lower performance than AVX2 and SSE. Hence, larger registers are not efficient in this query. As already explained, this is mostly because of a join dominating the performance of the query, which performs best when scalar processing is used, while the remaining operators profit from AVX512 when a high CPU frequency is set. Our approach successfully detects this and even sets the CPU frequency for the join to 3.5 GHz without any intermediate steps like the CPU driver. For this reason, the execution time is not only below the execution time of AVX512, but also below the execution time of scalar processing.



(a) Query 1.1



(b) Query 4.1

Figure 5.5: Our approach compared to a static choice of the vector size and the theoretical optimum if the best performing size is chosen for each operator. Additionally, in our approach we do not rely on the *pstate* cpu driver to set the frequency. Instead, we set it according to our approximated operator profiles (SF=10).

Scenario 2: Bandwidth Saturated System

As already argued, the optimal configuration might change when other threads on a system are claiming shared resources, e.g. memory bandwidth. This is the setting of our scenario 2. Additionally to the thread executing our query, all other CPU cores on the same NUMA node and the neighboring NUMA node run their maximum number of hyperthreads and continuously request data from main memory. This reduces the efficiently usable bandwidth for our query thread. We are able to set the frequency on a NUMA node level, which is the most fine-grained reliable setting possible with our system. Since the CPU driver setting is a global one, we can not rely on the *pstate* driver to manage the frequencies on the remaining nodes. Thus, we set them to the base frequency. The approach is the same as for the idle system with two exceptions. First, the primitive profiles for the bandwidth saturated system are used for the approximation of the operator profiles. Second, MorphStore was given an update, which introduced a range selection, such that the approximation also has to be done for this new operator.

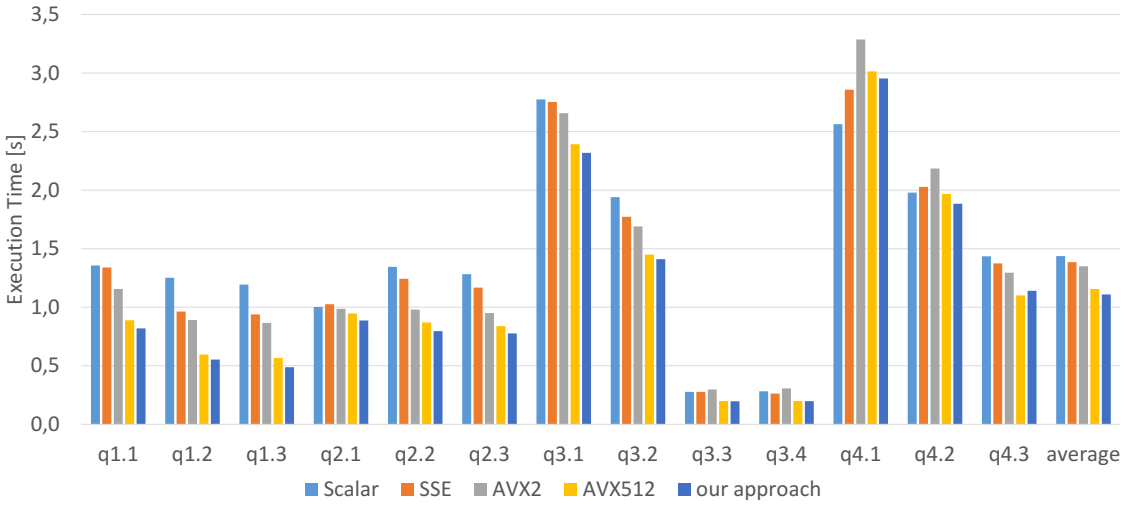


Figure 5.6: Results for the SSB while there is a continuous memory I/O load produced by the cores, which are not running the queries (SF=10, Scenario 2).

Again, we also measured the execution time for fixed instruction sets and vector register sizes. The results for all SSB queries are shown in Figure 5.6. For this scenario with resource conflicts, AVX512 is never the worst performing choice. Except for query 4.1 it is even always the best performing choice. Our approach is able to detect that AVX512 is the most reasonable choice in most operators. Scalar processing is only selected once per query in the queries 2.1 to 4.3. Moreover, this does not include the join operators anymore. However, the previously optimal CPU frequencies of over 3 GHz have decreased. In Figure 5.7, we show how often a certain CPU frequency is selected during query execution. The highest frequency our optimization suggests, is at 2.2 GHz while the most frequently suggested frequency is only at 1.4 GHz. This seems to be a reasonable choice, because we can achieve a similar or lower execution time than without our frequency setting in most queries. The exceptions are queries 4.1 and 4.3, where we still perform close to AVX512 with CPU frequency scaling.

Scenario 3: Compressed Data

In our third scenario we repeat scenario 1, but with compressed base data. Compressing the base data is common practice in column stores, although a very recent thesis by Patrick Damme shows that the compression of intermediates can reduce the memory footprint and the execution time of queries significantly[Dam20]. MorphStore offers a few compression techniques including static vertical bitpacking (static VBP). Static VBP is an algorithm for the Null Suppression technique. The number of bits used to store a value is static for a whole column. This number is chosen to use the smallest amount of bits, which can still accurately store the largest value of the respective column. The decompression of the data is done on-the-fly, i.e. when a register with compressed data is read from memory, it is directly decompressed and handed over to the operator. This avoids the decompression of the whole column, and therefore writing the decompressed data to memory, before an operator can be called.

To apply our approach to compressed data, we automatically counted the amount of called primitives for data compressed to different bit widths. We did this for the aggregation operator and compared the result to the primitive calls of an aggregation on uncompressed data. This provides us with the difference in primitive calls for each bit width,

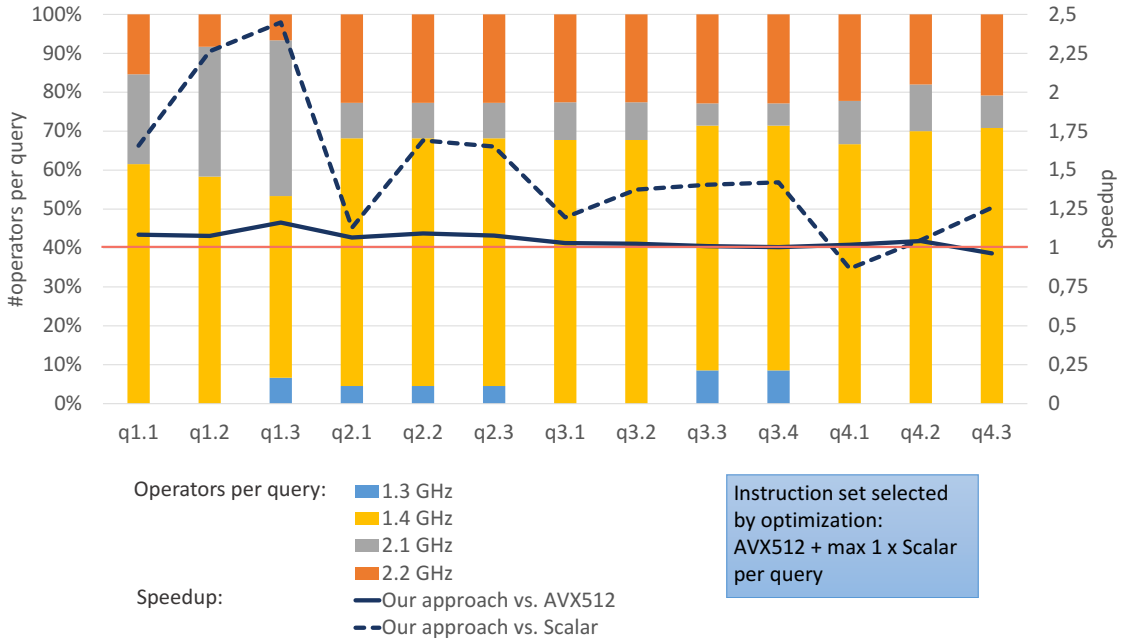


Figure 5.7: The CPU frequencies proposed by our optimization for scenario 2 and the speedup compared to scalar processing and to AVX512 with frequency scaling by the CPU driver. The instruction set chosen by our optimization was mainly AVX512. Scalar processing was only selected for one operator per query in the queries 2.1-4.3. (SF=10)

which is mainly a reduction in memory read accesses and an increase in shift operations and bitwise logic. Since the in-register decompression is always the same unregarded of the operator the result is handed to, we can apply the same results for all operators. The number of primitive calls represents the weights for the profile approximation. Thus, the changed number of primitive calls requires a new approximation for all operators, cardinality ratios, and bit widths. The SSB requires 16 different bit widths, one for the uncompressed intermediate data and 15 for the differently compressed columns of the base data. As a result, we approximated more than 600 operator profiles to choose from during query translation. The remaining steps during query translation and compilation stay the same as in the previous scenarios.

Figure 5.8 shows the query execution times for different instruction sets and for our approach. As already described by Damme, only compressing the base data has a rather moderate effect on query performance. Queries 4.1-4.3 benefit the most from the compression. The compressed data decreases the influence of the memory read access and increases the influence of computation. The used computation mainly consists of element-wise comparisons and bitwise logic, which do not show any dependencies between the elements of the vector register. Such operations profit from wide vector registers. Hence, the performance of AVX512 increases. Except for query 4.1, AVX512 is even the best performing instruction set now. However, even in query 4.1, it is still the best performing vector instruction set. Our approach successfully detects that AVX512 is now the reasonable choice for the runtime dominating joins for all cardinality ratios. Additionally, the CPU frequency is again set to 3.4 GHz resp. 3.5 Ghz, depending on the exact cardinality ratio, which is slightly below the maximum but more than 1 GHz above the base frequency. This reduces the execution time further by a small amount.

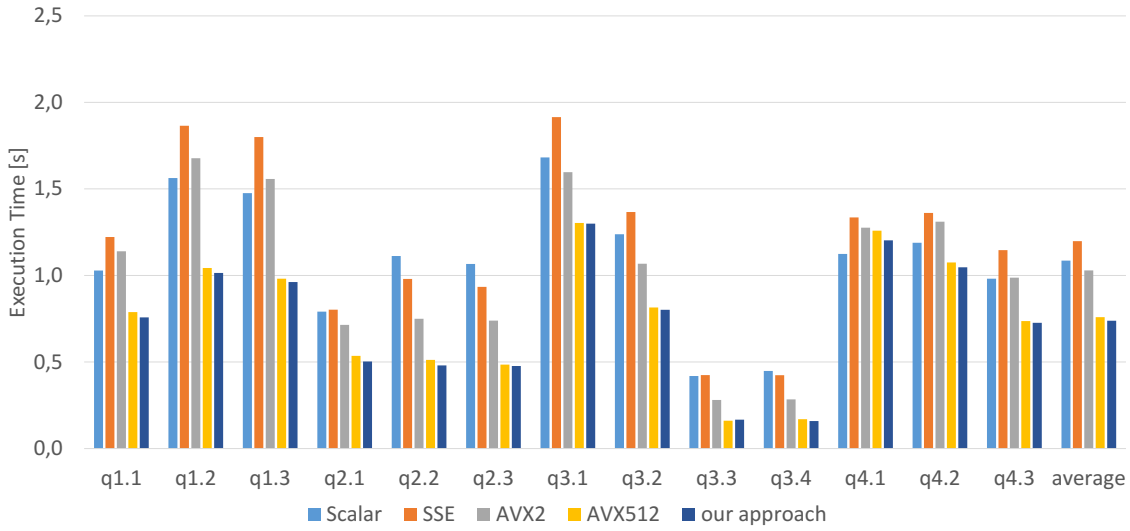


Figure 5.8: The execution times of the SSB queries with our optimization approach compared to fixed instruction sets for compressed base data (SF=10). The base data is compressed using static vertical bitpacking.

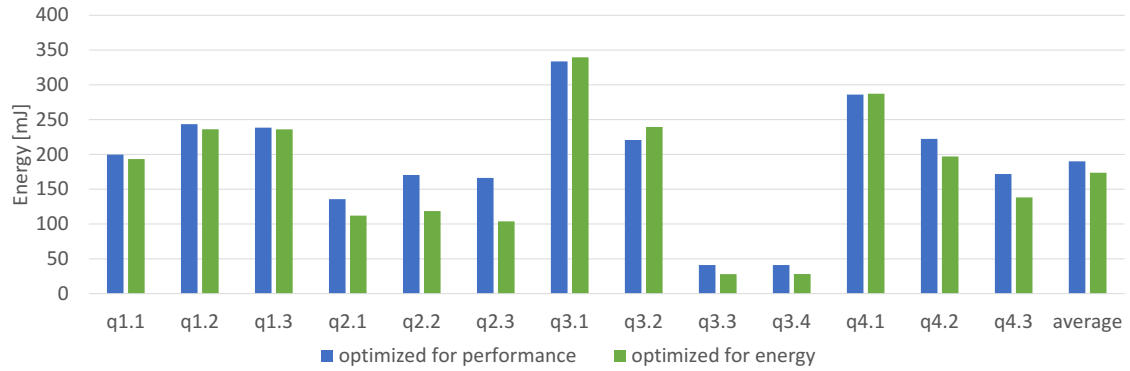


Figure 5.9: The energy consumption of configurations with a different optimization goal: performance and energy-efficiency (SF=10).

5.3.2 Energy Optimization

For showing the optimization for energy-efficiency, we take up scenario 1 again. Our optimization should already have chosen a configuration, which is as energy-efficient as possible for the highest achievable performance, because we select the configurations from the upper hull of the operator profiles. However, we can also explicitly use our approach to optimize for low energy consumption. To do this, we change a single step in our described approach: The upper hull of the operator profiles are sorted by *WEQ* instead of performance. This way the optimal configuration for each operator is the most energy-efficient one according to our approach. If performance and energy-efficiency are related to each other, this configuration should be the same. Hence, energy consumption should be equal regardless of our optimization goal being performance or energy-efficiency. In Figure 5.9 we compare the energy consumption of all power domains, i.e. all NUMA nodes, for both optimization objectives for all SSB queries. The graph shows that we could successfully save energy in all but three queries by changing our optimization goal. The slightly higher energy consumption in 3 queries is most likely caused by our coarse grained consideration of cardinality ratios, such that these queries contain operators, which are in a transition region between two configurations of which our approach chose the wrong one.

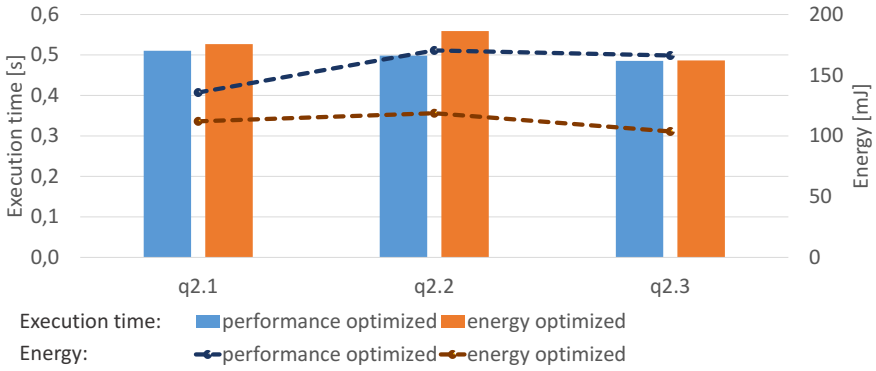


Figure 5.10: Runtime and energy consumption of the second set of SSB queries on test system A (SF=10). We compare an optimization for performance with an optimization for *energy-efficiency*. The graph shows, that performance does not equal *energy-efficiency* in every case. In this example, the energy consumption can be reduced on the cost of a minimal performance loss.

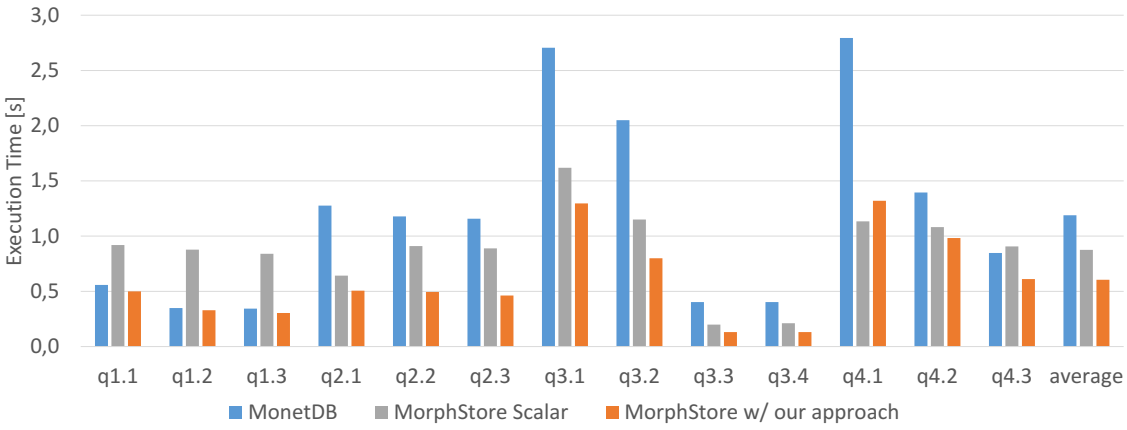


Figure 5.11: Execution time of MonetDB as compared to scalar execution in MorphStore and to our approach (SF=10).

However, especially for the queries 2.1-2.3 there is a significant difference of the energy consumption. A reason could be that our approach did not actually find the best performing configuration and the change of the optimization goal accidentally led to the desired result. To check this, we show the execution time and the energy-consumption of these queries in Figure 5.10. As shown, we did find better performing configurations with the optimization goal set to performance, although the difference is insignificant for query 2.3. On the cost of this insignificantly longer execution time, we gained about 1/3rd of energy savings. Moreover, this configuration still performs better than AVX512 on all operators with CPU driver frequency scaling. When looking at the individual operators, the queries spend most of the execution time on a join again. For query 2.2, our optimization suggested dropping the CPU frequency for its join operators from 3.5 GHz to 1.9 GHz to be most energy-efficient instead of showing the best performance. In query 2.3, only one of the 3 join operators uses this lower CPU frequency, while the other join operators run at 3.5 GHz.

5.4 COMPARISON WITH MONETDB

So far, we have compared our optimization approach to different configurations within MorphStore. Now, we compare our approach to MonetDB [IGN+12], a state-of-the-art column-store, which processes uncompressed data in a scalar manner. We used MonetDB v. 11.31.13 and configured it with the `-enable-optimize` flag. We build and run it on our test system A in a single thread and in read-only mode. Hence, we use the already described scenario 1 for this comparison. For a fair comparison, we also omit the `GROUP-BY` clause, use the same dictionary-encoded data as for MorphStore, and define all columns as a 64-bit data types (`BIGINT`). Since the query execution plans we use for MorphStore are a result of the MonetDB optimization, we can safely assume that MonetDB uses the same plans as MorphStore. We only measure the query execution time in MonetDB excluding query parsing and optimization. We use a scale factor of 10. Each query is repeated 12 times, where we omit the first run, because during this run, the data is loaded from disk into main memory. This is also necessary to ensure a fair comparison with MorphStore, which works entirely in main memory.

In a first step, we compare the execution time of the SSB queries in MonetDB to its according configuration in MorphStore. This configuration is scalar processing of 64-bit uncompressed integer data[5]. As we have shown, scalar processing can outperform vectorized processing in some queries. Thus, the choice of scalar processing is not made to produce an intentionally low baseline, but to show a fair comparison. In the second step, we add our optimization with performance as optimization goal. The results are shown in Figure 5.11. MorphStore with scalar processing and MonetDB outperform each other in different queries. There is no system, which is generally faster than the other one. Hence, MorphStore can keep up with a recent and widely used execution engine. When applying our optimization, the execution time decreases in all but one query (4.1). However, our approach outperforms MonetDB in all queries, even in those where MorphStore was initially significantly slower, e.g. in query 1.1.

[5]Note that this comparison uses the mentioned updated MorphStore, which includes a range selection.

SUMMARY AND FUTURE RESEARCH DIRECTIONS

In the final chapter, we summarize the challenges we tackled and our solution. Further, we discuss work, which is only loosely related to this thesis but provide a bigger picture and partially preceded the work presented in this thesis. Finally, we will give an outlook on possible future research directions.

6.1 SUMMARY

This thesis investigated the chances and challenges of vectorization for in-memory column-store engines. In Chapter 2, we gave an overview on the current state of vectorization in column-stores. Besides the plain operators, this also includes lightweight compression and specialized operators working on compressed data. We have also shown that the traditional Volcano iterator model is not suitable anymore for vectorized processing. Further, we gave a short introduction into the growing landscape of different instruction sets for vectorization. We experimentally show that large vector register sizes alone are not always beneficial. Moreover, they can also be counter-productive and decrease the overall performance. Additionally, the use of vectorization is bound to physical constraints, which influence not only performance, but also energy-efficiency, where this relationship differs between hardware. Since energy-efficiency has become an increasingly important optimization goal while the demand for performance has not decreased, we argue to not treat performance and energy-efficiency as independent optimization goals.

In Chapter 3, we investigated the reasons for the counter-intuitive behavior of large vector registers. We found that there are inherent and algorithmic reasons. While the inherent reasons can be solved by the introduction of new instructions, the algorithmic reasons require rethinking the existing operations and compression algorithms. Then, we provided a selected solution for Run Length Encoding, which uses the recently introduced conflict detection instructions, and remodels the compression algorithm completely. However, since the available instruction sets differ between hardware, several implementations for different targets are necessary. Even when these implementations exist, a meaningful choice of the instruction set and register size is crucial for both of our optimization goals, i.e. performance and energy-efficiency. To enable this choice on a fine-grained level without the requirement of individual implementations, we proposed the Template Vector Library (*TVL*). The *TVL* separates the algorithm implementation from the underlying hardware. It does this by introducing so-called primitives, abstract data types, and derived constants, which can be used for implementation. Each of these primitives and types has a back-end for different instruction sets, vector sizes, and base data types. The mapping between the implementation and the back-end is defined by a nested template, which we call *processing style*, hence the name Template Vector Library.

In Chapter 4, we used the *TVL* as a basis for a micro-benchmark-based optimization approach, which treats performance and energy-efficiency as interdependent user-defined optimization goals. This means, that the user can set a preference on performance, or energy-efficiency, or on an acceptable threshold of either of these goals. Our optimization then finds a configuration consisting of an instruction set and a CPU core count and frequency, which fulfills the user-defined primary optimization goal while being as optimal as possible for the other optimization goal. To reach this, we first explained why a purely analytic optimization strategy will not be feasible. Then, we introduced the concept of *Work-Energy-Profiles* (*WEPs*), which is the model we use for the mapping between performance and energy-efficiency. We presented a benchmark concept to create such profiles and discuss selected results. To create *WEPs* for more complex applications, we developed an approximation approach, which combines several primitive *WEPs*. We

apply this approach to our operators and compression algorithms and present selected results. Finally, we show different ways of using these *WEPs*.

In our end-to-end evaluation in Chapter 5, we applied our optimization strategy to the popular Star Schema Benchmark (SSB). We used MorphStore as an evaluation environment. Besides the performance optimization for different scenarios, we have also shown that the best performing configuration is not necessarily the most energy-efficient one. For the majority of SSB queries, we were able to find configurations, which required less energy than our best performing configurations, by using our optimization approach. A comparison with MonetDB wraps up the evaluation. We have shown that MorphStore provides a competitive performance, which we can significantly increase by applying our optimization.

6.2 BEYOND THIS THESIS

In this thesis, we presented software-based optimization approaches to make the best use of the available hardware. However, performance and energy-efficiency for database systems is not only targeted by software developers, but also by hardware developers. In the best case, both sides provide each other feedback and work together for a hardware/software co-design. In this sense, hardware can be optimized and built for the software it will be running instead of software trying to use whatever hardware is available. This section shall provide some insight into this topic[1].

Hardware can be designed in two fundamentally different ways: (1) as a design for field programmable gate arrays (FPGAs), which can be reconfigured multiple times, and (2) as application-specific integrated circuits (ASICs), which are hard-wired and cannot be changed after being built. In the first case, development cycles can be almost as short as the participating developers want them to be, because no hardware has to be built. The only restriction is the time required for the synthesis of a logical design into a physical one. In the second case, either hardware has to be built or simulators have to be written or configured, which leads to development cycles taking years. Aside from these differences, both approaches have their advantages and disadvantages, which we will briefly discuss in the following.

FPGAs

For prototyping or for very specialized functionality, which is only requested by a small user group, FPGAs are the tools of choice. FPGAs are also useful when the same hardware should be able to perform different but very specialized tasks, such that the hardware configuration has to be changed during runtime. The core concept of FPGAs is

[1]Parts of the material in this chapter have been developed jointly with Tomas Karnagel, Eric Mier, Dirk Habich, Wolfgang Lehner, Nils Asmussen, Marcus Völp, Sebastian Haas, Benedikt Nöthen, Gerhard Fettweis, and further members of the Vodafone Chair Mobile Communications Systems and the Chair for Highly-Parallel VLSI-Systems and Neuro-Microelectronics, TU Dresden. The chapter is based on [UHK+17, LUH18a, UHK+15, HAS+16]. [UHK+17] is published under a Creative Commons BY-SA license; the original publication is available at https://dl.gi.de/20.500.12116/641. The copyright of [UHK+15] and [HAS+16] are held by the Institute of Electrical and Electronics Engineers (IEEE); the original publications are available under https://doi.org/10.1109/ICDEW.2015.7129569 and https://doi.org/10.1109/NORCHIP.2016.7792904. A revised and extended version of [HAS+16] is also available at https://doi.org/10.1016/j.micpro.2017.10.002. The copyright of [LUH18a] is held by Springer Nature; the original publication is available under https://doi.org/10.1007/s13222-018-0276-y.

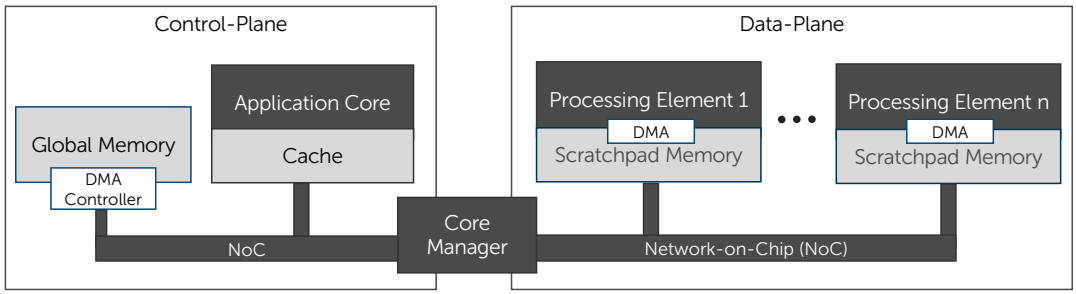


Figure 6.1: An overview of the Tomahawk architecture.

based on an array of configurable logic blocks each implementing a binary function, which can be changed by modifying the truth table of the particular function. This table is written to the SRAM cells of the logic block and can be overwritten, thus yielding a reconfigurable processing unit. The number of these logic blocks as well as other specifications and the price vary heavily between different FPGAs. Simple development kits cost less than 100 euro while more powerful boards cost up to 7000 euro. Besides differences in the interfaces, technology, and additional general-purpose cores, the applied FPGA chip is responsible for this huge price range. For example, expensive Intel/Altera boards use Stratix FPGAs with up to 5.5 million logic elements versus a MAX 10 FPGA with only up to 50k elements on low-priced boards[2]. Xilinx covers a similar range starting with the Spartan-6-boards in the entry-level segment and going up to the UltraScale+ for complex tasks. There are already some approaches to integrate high-end FPGAs into database systems. For example, [ZBB+16] presents concepts and implementations for hardware acceleration for almost all important operators appearing in SQL queries. Here, an SQL query is analyzed and divided into single operators and modules, which are subsequently concatenated to a data path and loaded onto the FPGA. Another approach to the fusion of databases with FPGAs is taken by Jahan et al. They implemented different scans, analyzed their performance, and compared them to a CPU approach, which makes use of SIMD and multithreading [LUH+18b]. As memory access is still the main bottleneck in both approaches, they also developed a high-throughput BitPacking compression for FPGAs [LNH+19] to reduce the memory footprint at minimal computation cost. A more general description of challenges for database developers working with FPGAs can be found in [Teu17].

The ability to be reconfigurable obviously comes with some limitations compared to ASICs. First, the resources on an FPGA are limited, i.e. the specific task to be supported by an FPGA-based implementation has to be carefully selected; second, building an application takes some time. For example, the synthesis of an application for an FPGA can take between several minutes and multiple hours, depending on the complexity of the application and the available computing power. This is why the main part of debugging must be done in simulations. Finally, the size of an FPGA exceeds the size of most hard-wired solutions by several orders of magnitude, which also increases the time a signal takes to traverse the whole circuit. For comparison, a device package with the Xilinx Spartan-7 FPGA covers between 64 mm² and 729 mm²[3], while the Titan3D (see next section), uses only 4.95 mm² [HSH+17].

[2] https://www.altera.com/products/fpga/max-series/max-10/overview.html, accessed: 03/01/2018
https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html, accessed: 03/01/2018

[3] https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, accessed: 01/01/2018

Figure 6.2: The Tomahawk DBA module including the processor and two 64 MB SDRAM modules.

	Bitmap Compression and Processing (AND, OR, XOR)			Hashing						Sorted Set Operations				
Extension / Processor	WAH	PLWAH	COMPAX	Hash + Lookup	Hash + Insert	Hash Keys	Hash Sampling	CityHash32	Merge Sort	Intersection	Union	Difference	Sort-Merge Join	Sort-Merge Aggregation (SUM)
BitiX	X	X	X											
HASHI				X	X	X	X	X						
Titan3D						X			X	X	X	X	X	X
Tomahawk DBA	X					X	X		X	X		X		

Figure 6.3: Overview of the extensions and the processors they are implemented on. The processors which have actually been manufactured at this point are highlighted.

Low-Power ASICs

An application specific integrated circuit (ASIC) is basically a hard-wired circuit, which is designed to perform a certain domain-specific task. However, a database also requires some general purpose computing abilities. Therefore, it makes sense to extend general purpose computing units with selected ASICs. To also save energy, extensible low-power processors like the Tensilica LX5 can be used for this purpose. This requires the domain experts on all levels from the circuit design to the software development to work together, which is exactly what was done as a part of the large-scale research project Center for Advancing Electronics Dresden (cfaed) [CLK+17]. As a result, several generations of the Tomahawk System-on-Chip (SoC), which features different ASICs on its processing units, were designed, built, and tested.

Generally, the Tomahawk platform [AMN+14] consists of two subsystems called control-plane and data-plane as illustrated in Fig. 6.1. The control-plane subsystem comprises a CPU (traditional fat core), a global memory and peripherals, whereas this CPU is also called application core (App-Core). The App-Core is responsible for executing the application control-flow. The data-plane subsystem consists of a number of processing elements (PEs), each equipped with a local program and data memory. PEs are not able to access the global memory directly, instead a data locality approach is exploited using

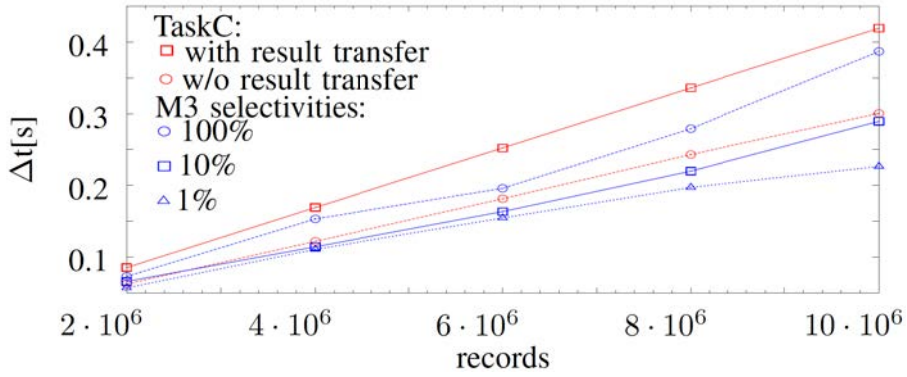


Figure 6.4: Runtime comparison between TaskC and M3. TaskC runtimes are not differentiated by selectivity since the amount of transferred data is always the same.

scratchpad local memory. That means, the PEs are explicitly isolated from the control-plane subsystem. The data-plane subsystem is used as an accelerator for the control-plane. Therefore, this subsystem can be seen as slave unit in the overall system architecture. Both subsystems are decoupled logically and linked together through a controller called Core Manager (CM). The CM is responsible for the task scheduling of the PEs, the PE allocation, and data transfers from global memory to the PEs and vice versa. Additionally, the CM can perform frequency scaling of the PE cores to minimize the power consumption.

Based on the overall collaborative setting in the cfaed project, this Tomahawk platform could be tailored for database requirements. A number of modifications has been added to enhance data processing. This includes minor and uncritical changes like an enlarged scratchpad memory of the PEs as well as more sophisticated enhancements like specialized instruction set extensions for the PEs, realized by ASICs. To further increase the performance, the DMA controller of the global memory has been enhanced to push down data intensive operations, like filtering or pointer chasing. The objective of this DMA enhancement (or intelligent DMA, iDMA) is to reduce the necessary data transfer between the control and data-plane which is clearly a bottleneck for processing large amounts of data.

The overall development of the Tomahawk is conducted in cycles, where the first version to feature a specialized instruction set for database operations is the Tomahawk 3. Therefore, it is also called the *Tomahawk Database Accelerator (DBA)* (Fig. 6.2). The Titan3D [HAS+16] was a prototype processor representing a single PE. Further, there are processor prototype designs, which were not built in reality, namely BitiX [HKA+16] and Hashi [AHF+14]. An overview of the different extensions is shown in Figure 6.3. A more detailed explanation of the layout of these instructions and of the iDMA can be found in [UHK+17].

Application of Low-Power ASICs

During several projects, which were preceding this work, we were investigating two challenges of the Tomahawk:

1. Since it is bare metal without any operating system, how can it be used in the most efficient and convenient way?

2. How can the offered instructions be applied beyond their intended use to gain maximum performance at minimal energy consumption?

For the first challenge, we considered two different concepts: TaskC, which is a programming interface introduced alongside the hardware concept, and M3, a minimalist microkernel-based operating system developed at the operating system group of TU Dresden.

Generally, the TaskC concept is similar to CUDA kernels or OpenCL kernels. Therefore, the implementation of query operators is straightforward. A database query is a data flow graph represented as host program and within this program, several tasks can be called, whereas we have to explicitly specify the parallelism by defining the number of tasks for each operator based on the data size. The size of input and output arrays must be stated when a task is started. The CM is responsible for the initialization of any data transfers and for dispatching the tasks to the PEs. Unfortunately, query processing with TaskC using the proposed Core Manager suffers from the limitation that the resulting set of data has to be estimated before the task is executed producing unnecessarily large return arrays. In typical database systems, this size is only known at runtime, after processing the input data. Furthermore, the whole output array is always sent back to the DRAM, regardless of the number of results that have actually been written. This leads to many expensive transfers which is especially bothering when the resulting dataset is rather small, compared to the input dataset.

The concept of M3 is to run a microkernel on a dedicated core and remote-control the other cores. That is, both the microkernel and the applications run alone on their cores, whereas the applications get linked against a library that provides them with abstractions for application creation, communication, and memory management. In contrast to TaskC, where tasks are isolated by definition because they cannot communicate with other PEs or the DRAM, a central point of M3 is to increase the flexibility for applications by allowing them to communicate. However, allowing communication requires isolation, i.e. M3 needs to ensure that applications cannot influence or even destroy each other, because otherwise the system cannot be used by multiple applications at the same time. Since the microkernel is running on a dedicated core, it is not involved in a communication between applications, but data is directly exchanged between the cores the applications run on. For that reason, M3 builds upon a small hardware component for each core, that allows to establish communication channels between cores or between a core and a memory. This hardware component can be used by the application, but its configuration, i.e. where data can be sent to, for example, is not accessible by the application, but only by the microkernel. Thereby, the communication capabilities of applications are in complete control of the microkernel. The M3 kernel and library allows event driven data transfer, i.e. data is only sent when necessary and the receiver implements an event handler for processing incoming messages. It may be expected that the application can profit from the absence of unnecessary transfers.

To test this hypothesis, we applied the evaluation of a selection with 3 predicates. In TaskC, 4 tasks, one per PE, is started and each task evaluates every predicate. This eliminates any memory accesses for intermediate materialization between predicate evaluations. However, the amount of data written back to memory is as large as defined when the tasks were spawned, which is the same amount as the input size. This is necessary to not lose any results, because we do not know the size of the result set at the beginning of a task. In M3, two PEs evaluate the first predicate. The result of this evaluation is sent to two other PEs, which evaluate the remaining two predicates. Only the result of the last evaluation is written back to memory. Figure 6.4 shows the execution time for our query for M3 with different selectivities and for TaskC, where there is no runtime difference depending on the selectivity. Hence, the TaskC concept initially developed

for mobile communications, which has a fixed sender-receiver-model, is not suitable for query processing, whereas the microkernel approach is promising.

The second challenge was the use of the specialized instructions. What seemed like an easy task, which only requires some cooperative knowledge exchange, turned out to be more tricky than expected. For instance, instructions on the Tomahawk like intersection, union, sort, and difference, perform this operation on a number of sequentially read values and return the result, which is optimal for working on columnar data. What they do not return is an additional position or index list, or a bitmap. Thus, a projection to retrieve the remaining parts of a tuple is not possible, which reduces the applicability of the operations to index lists. However, if an operator is executed in two phases, the instructions can still be used in the first phase. The idea is to determine the selectivity or cardinality of a result set of an operator in the first phase. The actual result is not written during this first phase. As we have learned, it can be useful to know the size of a result set, e.g. to not write unnecessary data, not allocate unnecessarily large amounts of memory, or optimize the order of operators. The second phase executes the operator again and writes the final result, this time without the specialized instructions, which do not provide the desired output. In [HAS+16] we have shown exemplarily that this does indeed increase the average overall query performance including both phases.

6.3 FUTURE RESEARCH TOPICS

The development of SIMD instruction sets is as vivid as the database community presenting new systems and architectural concepts every year at the established conferences. For this reason, the topic of this thesis stays exciting and can serve as the starting point for further research. In the following, we will briefly present a few ideas for this future research and development.

Integration of Novel Architectures and Instruction Sets As we have shown, the landscape of hardware for vectorized processing is manifold and developing. For instance, the most recent SIMD instruction set of ARM, SVE, provides scalable vector sizes. This creates new challenges for our *TVL* and for vectorized programming in general, e.g. constants like the vector size are not known at compile time anymore. It is also crucial to examine any new instruction set for its ability to tackle the inherent challenges of large vector register, which is often a prerequisite to overcome the algorithmic challenges, as we have shown for the RLE example. This new variety in vector sizes and the fact that current software is not developed for the new instruction set, makes an abstraction of SIMD programming more important than ever before.

Moreover, there is a project integrating the instructions of the vector engine SX-Aurora into our *TVL*. This will give us the opportunity to use a common codebase to investigate the interplay of the vector engine with its Intel Skylake host processor for efficient query processing. That means, we want to leverage the wide vector registers as well as the high memory throughput of the vector engine while compensating the shortage of processing logic on the SX-Aurora with the Intel Skylake.

Optimization Push-Down As well as our optimization works, we still have to use tools provided by the operating system to change CPU frequencies, and we have to know our processing styles at query compile time. To use the tools of the operating system, we spawn a separate thread, which runs on an otherwise idle NUMA node. We do this to ensure a flawless use of the *CPUfreq* interface even when the remaining system is heavily used by I/O intensive threads. This additional step of interface usage could be avoided if the operating system was configuring its frequency by itself according to the specific hardware, current overall system usage, the used instruction set, and the kind of recent memory accesses pattern. The case for the choice of instruction sets is different, since this is defined at compile time. However, a compiler could use primitive profiles to narrow down the selection of instruction sets and produce several variants, which can be chosen from depending on the system usage. Moreover, a global database containing the primitive benchmark results for different systems can eliminate the requirement for individual benchmarks on each instance of a system. Such databases already exist for graphic benchmarks. Thus, given a userbase, which is large enough, and risking less accuracy due to shortened benchmarks, this is a realizable goal.

Query Prioritization With the physical constraints of vectorization in mind, the prioritization of queries gets another dimension. For instance, cores executing low-priority queries can be advised to use only scalar code or SSE, while high-priority queries are allowed to use AVX-512. This allows for less frequency down-scaling on the whole system, because AVX2 and AVX-512 scale down the core frequency if multiple cores are used, where this effect is less significant with scalar or SSE processing.

Hardware Development As briefly shown in Section 6.2, hardware development does not have to be a one-sided process. In a collaborative environment, feedback from software developers can influence hardware design decisions. For query processing, there are a few features, which are worth considering. Since AVX512 has turned out to not be the final answer for all memory intensive workloads, the optimization potential may lie somewhere else. A few of these things could be an upgrade of the memory buses in size and number, and moving the computation closer to memory. The latter conforms to the electrical engineering definition of in-memory processing, similar to the previously mentioned iDMA approach. Especially automatic compression and decompression in memory and in cache could be extremely beneficial to overcome the memory wall. However, as our experience has shown, academic research is only able to provide prototypes. Large scale field tests and production requires industrial engagement.

BIBLIOGRAPHY

[AHF+14] Oliver Arnold, Sebastian Haas, Gerhard P Fettweis, Benjamin Schlegel, Thomas Kissinger, Tomas Karnagel, and Wolfgang Lehner. Hashi: An application specific instruction set extension for hashing. In *ADMS@ VLDB*, pages 25–33, 2014.

[AMN+14] Oliver Arnold, Emil Matus, Benedikt Noethen, Markus Winter, Torsten Limberg, and Gerhard Fettweis. Tomahawk: Parallelism and heterogeneity in communications signal processing mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(3s):1–24, 2014.

[AOA15] Raja Appuswamy, Matthaios Olma, and Anastasia Ailamaki. Scaling the memory power wall with dram-aware data management. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, pages 1–9, 2015.

[B+18] Tobias Behrens et al. Efficient SIMD vectorization for hashing in opencl. In *EDBT*, 2018.

[BHC12] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. Mcjoin: a memory-constrained join for column-store main-memory databases. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 121–132, 2012.

[BMK+99] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[CLK+17] Jeronimo Castrillon, Matthias Lieber, Sascha Klueppelholz, Marcus Völp, Nils Asmussen, Uwe Assmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Froehlich, et al. A hardware/software stack for heterogeneous systems. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):243–259, 2017.

[D+17] Patrick Damme et al. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, 2017.

[D+19] Patrick Damme et al. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *TODS*, 2019.

[Dam20] Patrick Damme. *Analytical Query Processing Based on Continuous Compression of Intermediates*. PhD thesis, Technische Universität Dresden, 2020.

[DGR+74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[DHHL17] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[DHL15] Patrick Damme, Dirk Habich, and Wolfgang Lehner. A benchmark framework for data compression techniques. In *TPCTC*, 2015.

[DUP+20] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. Morphstore: Analytical query engine with a holistic compression-enabled processing model. In *VLDB*, 2020. accepted.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[EFGL14] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. Boost. simd: generic programming for portable simdization. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pages 1–8, 2014.

[FLKX15] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 31–46, 2015.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[Gra94] Goetz Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[H+13] Max Heimel et al. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9), 2013.

[HAS+16] Sebastian Haas, Oliver Arnold, Stefan Scholze, Sebastian Höppner, Georg Ellguth, Andreas Dixius, Annett Ungethüm, Eric Mier, Benedikt Nöthen, Emil Matúš, et al. A database accelerator for energy-efficient query processing and optimization. In *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, pages 1–5. IEEE, 2016.

[HDVH12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[HKA+16] Sebastian Haas, Tomas Karnagel, Oliver Arnold, Erik Laux, Benjamin Schlegel, Gerhard Fettweis, and Wolfgang Lehner. Hw/sw-database-codesign for compressed bitmap index processing. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 50–57. IEEE, 2016.

[HNZB07] Sándor Héman, Niels Nes, Marcin Zukowski, and Peter Boncz. Vectorized data processing on the cell broadband engine. In *Proceedings of the 3rd international workshop on Data management on new hardware*, pages 1–6, 2007.

[HSH+17] Sebastian Haas, Stefan Scholze, Sebastian Höppner, Annett Ungethüm, Christian Mayr, René Schüffny, Wolfgang Lehner, and Gerhard Fettweis. Application-specific architectures for energy-efficient database query processing and optimization. *Microprocessors and Microsystems*, 55:119–130, 2017.

[HSMR09] Stavros Harizopoulos, Mehul Shah, Justin Meza, and Parthasarathy Ranganathan. Energy efficiency: The new holy grail of data management systems research. *arXiv preprint arXiv:0909.1784*, 2009.

[IGN+12] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[Int16] Intel. *Intel® 64 and IA-32 ArchitecturesSoftware Developer's Manual, Volume 2B*, 2016.

[Int20] Intel. *Intel® Xeon® Processor Scalable Family, Specification Update*, 2020. Available at https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf, accessed on 30/03/2020.

[JPG04] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, pages 275–280, 2004.

[K+18] Timo Kersten et al. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13), 2018.

[KHL18] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Adaptive energy-control for in-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 351–364, 2018.

[Kis17] Thomas Kissinger. *Energy-Aware Data Management on NUMA Architectures*. PhD thesis, Technische Universität Dresden, 2017.

[KKL+09] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[KL12a] Matthias Kretz and Volker Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 42(11), 2012.

[KL12b] Matthias Kretz and Volker Lindenstruth. Vc: A C++ library for explicit vectorization. *Softw., Pract. Exper.*, 42(11):1409–1430, 2012.

[KM17] P Karpiński and J McDonald. A high-performance portable abstract interface for explicit simd vectorization. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 21–28, 2017.

[KN11] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[L+14] Roland Leißa et al. Sierra: a simd extension for c++. In *WPMVP*, 2014.

[L+15] Daniel Lemire et al. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1), 2015.

[LFV+12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.

[LNH+19] Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, Dirk Habich, Akash Kumar, and Wolfgang Lehner. High-throughput bitpacking compression. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 643–646. IEEE, 2019.

[LP13] Yinan Li and Jignesh M Patel. Bitweaving: fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300, 2013.

[LUH18a] Wolfgang Lehner, Annett Ungethüm, and Dirk Habich. Diversity of processing units. *Datenbank-Spektrum*, 18(1):57–62, 2018.

[LUH+18b] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan Duy Anh Nguyen, and Akash Kumar. Fpga vs. simd: Comparison for main memory-based fast column scan. In *Data Management Technologies and Applications: 7th International Conference, DATA 2018, Porto, Portugal, July 26–28, 2018, Revised Selected Papers*, volume 862, pages 116–140. Springer, 2018.

[M+17] Prashanth Menon et al. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1), 2017.

[MRS+14] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, pages 1–10, 2014.

[Neu11] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.

[OOC07] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. The star schema benchmark (ssb). 2007.

[P+15] Orestis Polychroniou et al. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, 2015.

[P+16] Holger Pirk et al. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14), 2016.

[P+19a] Johannes Pietrzyk et al. First investigations of the vector supercomputer sx-aurora TSUBASA as a co-processor for database systems. In *BTW Workshops*, 2019.

[P+19b] Orestis Polychronious et al. Towards practical vectorized analytical query engines. In *DaMoN*, 2019.

[Pla09] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2, 2009.

[S+17] Nigel Stephens et al. The ARM scalable vector extension. *IEEE Micro*, 37(2), 2017.

[SAB+18] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.

[SBB+17] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. The arm scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary search on modern processors. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 52–60, 2009.

[SGL10] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, pages 34–40, 2010.

[Teu17] Jens Teubner. Fpgas for data processing: Current state. *it-Information Technology*, 59(3):125–131, 2017.

[THS10] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242, 2010.

[TWZX14] Yi-Cheng Tu, Xiaorui Wang, Bo Zeng, and Zichen Xu. A system for energy-efficient data management. *ACM SIGMOD Record*, 43(1):21–26, 2014.

[U+18] Annett Ungethüm et al. Conflict detection-based run-length encoding - AVX-512 CD instruction set in action. In *ICDE Workshops*, 2018.

[UDP+17] Annett Ungethüm, Patrick Damme, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. Balancing performance and energy for lightweight data compression algorithms. In *ADBIS Short Papers*, pages 37–44, 2017.

[UHK+15] Annett Ungethüm, Dirk Habich, Tomas Karnagel, Wolfgang Lehner, Nils Asmussen, Marcus Völp, Benedikt Nöthen, and Gerhard Fettweis. Query processing on low-energy many-core processors. In *2015 31st IEEE International Conference on Data Engineering Workshops*, pages 155–160. IEEE, 2015.

[UHK+17] Annett Ungethuüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, and Wolfgang Lehner. Overview on hardware optimizations for database engines. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[UKHL16] Annett Ungethüm, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Work-energy profiles: General approach and in-memory database application. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 142–158. Springer, 2016.

[UKM+16] Annett Ungethüm, Thomas Kissinger, Willi-Wolfram Mentzel, Dirk Habich, and Wolfgang Lehner. Energy elasticity on heterogeneous hardware using adaptive resource reconfiguration live. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2173–2176, 2016.

[UPD+20] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. Hardware-oblivious simd parallelism for in-memory column-stores. CIDR, 2020.

[WFXS11] Jun Wang, Ling Feng, Wenwei Xue, and Zhanjiang Song. A survey on energy-efficient data management. *SIGMOD*, 40(2), 2011.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[Xu10] Zichen Xu. Building a power-aware database management system. In *Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research*, pages 1–6, 2010.

[ZB12] Marcin Zukowski and Peter A Boncz. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, 35(1):21–27, 2012.

[ZBB+16] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. Fpga-based dynamically reconfigurable sql query processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(4):1–24, 2016.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.

[ZR02] Jingren Zhou and Kenneth A Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156, 2002.

[ZZL+15] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. A general simd-based approach to accelerating compression algorithms. *ACM Transactions on Information Systems (TOIS)*, 33(3):1–28, 2015.

LIST OF FIGURES

LIST OF TABLES

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, October 6, 2020