Novel Vector Architectures for Data Management

Timothy Hayes
Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors

A thesis submitted for the degree of
Doctor of Philosophy in Computer Architecture
May, 2016

Director: Prof. Mateo Valero
Codirector: Dr. Oscar Palomar

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola de Doctorat

Assessment results for the doctoral thesis

Academic year:

Full name

Doctoral programme

Structural unit in charge of the programme

Decision of the committee

In a meeting with the examination committee convened for this purpose, the doctoral candidate presented the topic of his/her doctoral thesis entitled

____

____.

Once the candidate had defended the thesis and answered the questions put to him/her, the examiners decided to award a mark of:

☐ UNSATISFACTORY ☐ SATISFACTORY ☐ GOOD ☐ VERY GOOD

(Full name and signature) Chairperson	(Full name and signature) Secretary

(Full name and signature) Member	(Full name and signature) Member	(Full name and signature) Member

_________________________, ____

The votes of the members of the examination committee were counted by the Doctoral School at the behest of the Doctoral Studies Committee of the UPC, and the result is to award the CUM LAUDE DISTINCTION:

☐ YES ☐ NO

(Full name and signature) Chair of the Standing Committee of the Doctoral School	(Full name and signature) Secretary of the Standing Committee of the Doctoral School

Barcelona, __________

Per a la Carla.

Abstract

As the rate of annual data generation grows exponentially, there is a demand to manage, query and summarise vast amounts of information quickly. In the past, frequency scaling was relied upon to push application throughput. Today, Dennard scaling has ceased, and further performance must come from exploiting parallelism. Vector architectures offer a highly efficient and scalable way of exploiting data-level parallelism (DLP) through sophisticated single instruction-multiple data (SIMD) instruction sets. Traditionally, vector machines were used to accelerate scientific workloads rather than business-domain applications. In this thesis, we design innovative vector extensions for a modern superscalar microarchitecture that are optimised for data management workloads. Based on extensive analysis of these workloads, we propose new algorithms, novel instructions and microarchitectural optimisations.

We first profile a leading commercial decision support system to better understand where the execution time is spent. We find that the hash join operator is responsible for a significant portion of the time. Based on our profiling, we develop lightweight integer-based pipelined vector extensions to capture the DLP in the operator. We then proceed to implement and evaluate these extensions using a custom simulation framework based on PTLsim and DRAMSim2. We motivate key design decisions based on the structure of the algorithm and compare these choices against alternatives experimentally. We discover that relaxing the base architecture's memory model is very beneficial when executing a vectorised implementation of the algorithm. This relaxed model serves as a powerful mechanism to execute indexed vector memory instructions out of order without requiring complex associative hardware. We find that our vectorised implementation shows good speedups. Furthermore, the vectorised version exhibits better scalability compared to the original scalar version run on a microarchitecture with larger superscalar and out-of-order structures.

We then make a detailed study of SIMD sorting algorithms. Using our simulation framework we evaluate the strengths, weaknesses and scalability of three diverse vectorised sorting algorithms—quicksort, bitonic mergesort and radix sort. We find that each of these algorithms has its unique set of bottlenecks. Based on these findings, we propose VSR sort—a novel vectorised non-comparative sorting algorithm that is based on radix sort but without its drawbacks. VSR sort, however, cannot be implemented directly with typical vector instructions due to the irregularity of its DLP. To facilitate the implementation of this algorithm, we define two new vector instructions and propose a complementary hardware structure for their execution. We find that VSR sort significantly outperforms each of the other vectorised algorithms.

Next, we propose and evaluate five different ways of vectorising GROUP BY data aggregations. We find that although data aggregation algorithms are abundant in DLP, it is often too irregular to be expressed efficiently using typical vector instructions. By extending the hardware used for VSR sort, we propose a set of vector instructions and novel algorithms to better capture this irregular DLP. Furthermore, we discover that the best algorithm is highly dependent on the characteristics of the input.

Finally, we evaluate the area, energy and power of these extensions using McPAT. Our results show that our proposed vector extensions come with a modest area overhead, even when using a large maximum vector length with lockstepped parallel lanes. Using sorting as a case study, we find that all of the vectorised algorithms consume much less energy than their scalar counterpart. In particular, our novel VSR sort requires an order of magnitude less energy than the scalar baseline. With respect to power, we discover that our vector extensions present a very reasonable—if any—increase in wattage.

Keywords: Computer Architecture, Microarchitecture, Data-Level Parallelism, Vector Processors, Single Instruction-Multiple Data, Database Management Systems, Decision Support Systems, Online Analytical Processing, Hash Join, Sorting, Aggregation

Acknowledgements

First and foremost, I would like to thank my immediate thesis director—Dr. Oscar Palomar. You have been an outstanding mentor for me. I am very grateful for your encouragement, demeanour and wealth of knowledge. I feel that I have learned a great deal from you and hope that one day I can give to another what you have given to me. It has been a privilege to work with you.

I would like to express my appreciation to my three supervisors at the Barcelona Supercomputing Center—Dr. Osman Ünsal, Dr. Adrián Cristal and Prof. Mateo Valero. You have collectively brought an immeasurable amount of wisdom and support to my research. You have always given me the benefit of the doubt, encouraged my ideas and pushed me to achieve my very best. I appreciate the great opportunity that you've given to me.

A big thank you to the members of my thesis defence committee—Prof. Ramón Beivide, Prof. Víctor Viñals, Dr. Roger Espasa, Dr. Rubén Titos and Dr. Juan Manuel Cebrián. I would like to thank Dr. Enric Morancho for participating in my thesis pre-defence and providing great feedback and suggestions. I would also like to give a shout-out to the excellent staff at the Barcelona Supercomputing Center and UPC's Departament d'Arquitectura de Computadors, especially Joana Munuera and Dr. Xavier Masip.

My sincere gratitude to all of my colleagues—both current and former—at the Barcelona Supercomputing Center. We've had a lot fun times together. In particular, I want to thank the good folks who shared the "vector office" with me. I can't think of a single day where our collective diligence didn't break down resulting in a cacophony of incoherent ramblings about politics, religion and history (this nonsense went so far as to getting DNA tested for ancestry composition to determine whose opinion held more sway on European matters). If we weren't arguing about something topical, chances are we were complaining about work. This was certainly a pleasant distraction from the toil of all those paper deadlines.

A heartfelt thank you to all of my wonderful friends, especially those I've had the pleasure of meeting in Barcelona. You've shared your culture, customs and languages with me and have made my stay in this city such a special one. After living here for six

years and experiencing many great moments together, I feel like an honorary Catalan. A special thanks goes to Asier Roa who helped induct me into Barcelona and is once again drafting me to my next destination.

Last but not least, I wish to thank my partner Carla and both of our families. Carla has been a constant source of strength and inspiration. You supported me through thick and thin and have been unimaginably patient waiting for this time to arrive. We've shared countless great moments together and I know we will share many more in the next phase of our life. Words cannot express how grateful I am to you.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under the AXLE project (GA nº 318633), the RoMoL ERC Advanced Grant (GA nº 321253) and has been supported in part by the European Union (FEDER funds) under contract TTIN2015-65316-P. From 2013 to 2016 I was supported by a *formación de profesorado universitario* research grant from the Spanish *Ministerio de Educación, Cultura y Deporte*.

Contents

CHAPTER 1

Thesis Overview

In this chapter, we define our premise and convey the overall goal of this thesis. We first motivate the idea of accelerating database management systems (DBMSs) with vector extensions. After, we outline our objectives and then summarise the contributions of the thesis.

1.1 Motivation

DBMSs have become an essential tool for industry and research and are often a significant component of data centres. They can be used in a multitude of scenarios including decision support, data mining, e-commerce and scientific analysis. DBMSs can be broadly categorised into two main categories—(1) online transaction processing (OLTP), and (2) online analytical processing (OLAP). OLTP is characterised by small to medium sized databases, frequent updates and short queries. OLAP, in contrast, is characterised by very large databases, infrequent updates—usually done in batch—and long compute-intensive queries. Typically, a DBMS will be implemented and optimised targeting either OLTP or OLAP and its design and architecture will be quite different depending on the area chosen. OLAP is the fundamental part of a decision support system (DSS) which allows users to ask complex business-oriented questions over huge amounts of data.

As the amount of information to manage grows exponentially each year [MCB^{+}11, CML14], there is a pressure on software and hardware developers to create data centres that can cope with the increasing requirements. Advances in both hardware and software have moved OLAP databases further away from disk storage and relocated its contents instead to main memory for real-time operation [BMK99]. This has shifted the problem from being IO-bound to being CPU/memory-bound. It is therefore necessary to revisit the strategies used to accelerate data management techniques and find new methods that lie closer to the hardware-software interface. At the same time, there is now also an additional demand to provide greener and more power-efficient data centres while simultaneously pushing for better performance [HB09].

Moore's Law [Moo65] famously predicted that the number of transistors on an integrated circuit would double approximately every two years. For a long time, Moore's Law had two fundamental outcomes—(1) more features and functionality in an integrated circuit, and (2) higher operating frequencies with the same power density. Both of these outcomes contributed to the processor's overall performance. Frequency scaling was generally transparent to the programmer and algorithms were expected to execute faster with every new generation of processor. In the beginning of the twenty-first century in what is seen as the breakdown of Dennard scaling [DGR+74, Boh07], thermal and power issues made it infeasible to continue increasing the processor's operating frequency. While Moore's Law still held true, the free performance scaling that we had grown accustomed finally came to an end [Sut05]. The industry has now had to shift their focus on using the extra available transistors to achieve better performance through explicit parallelism.

Parallel techniques can be broadly categorised as instruction-level (ILP), thread-level (TLP) and data-level (DLP) [Fly66]. When it is possible to exploit it, DLP is by far the most efficient form of parallelism [HP12]. DLP is defined as applying the same operation to more than one element of (typically homogeneous) data. Strict definitions may require that the operations be independent whereas more lax definitions may allow for semi-independent operations. DLP can be exploited in a variety of ways. Recent developments in processor architectures have pushed a focus on multi-core acceleration. While it is generally straightforward to transform DLP to TLP in order take advantage of a multi-core architecture, single instruction-multiple data (SIMD) instruction sets offer a much more efficient way to capture and accelerate DLP [GP07, LAB+11].

SIMD instructions are concise, expressive and scalable. One of principal advantages of SIMD lies in the fact that many operations can be expressed in an ultra-compact form with much less encoding overhead than techniques used to exploit ILP and TLP. This succinct form in itself allows for lightweight hardware implementations which can infer and exploit the independence and homogeneity of the data as well as the repetitiveness of the operation. SIMD also reduces meta-work of an algorithm by curtailing the number of control flow instructions allowing the processing unit to instead focus on the operations of interest.

The SIMD paradigm goes back as early as the 1940s [HJ88]. One of the earliest examples of a machine using SIMD is the English Electric DEUCE [Hal56] based on the earlier Pilot ACE [Wil53]. The DEUCE used an instruction set architecture (ISA) with a limited SIMD facility. The machine's memory consisted of twelve mercury delay lines each holding thirty-two 32-bit words. To access a particular value in memory, the processing unit needed to wait for that word to circulate through the delay line into the circuitry. Due to the limited amount of memory as well as the long latency when accessing individual locations, the ISA included a modifier called a 'characteristic' which allowed a set of consecutive words within a single delay line to be used as operands instead of individual words. The DEUCE was an accumulator machine and this restricted the type of SIMD instructions to unary operations. It was also possible to sum all of the elements of a mercury delay line together using a single instruction. This kind of pipelined sum reduction operation would reappear in later SIMD architectures.

The 1960s saw substantial advances in SIMD architectures. This was principally motivated by the realisation that scientific problems—which require substantial processing power—are abundant in DLP. The SIMD paradigm fit nicely with the `do loop` construct of FORTRAN—the language of choice at the time for developing scientific and engineering applications. Two principal schools of SIMD design emerged—array architectures and pipelined architectures [Fly72], the latter becoming better known as vector architectures [Kog81, Sch87a, Sch87b, Cra96, EVS98].

There are many design similarities and differences between array and vector architectures, and also many distinctions between the implementations of the architectures within each of these design categories. An exhaustive list of these differences and similarities is beyond the scope of this thesis, however, from this author's personal perspective there is one fundamental difference between early array and vector architectures—the former were typically designed with isolated local memories per processing element whereas the latter used a unified memory space. This is a very significant difference because it directly influenced the programmability of the machines and determined the types of problem that could be accelerated. The vector machines were a natural extension to an already familiar scalar Von Neumann design. This led to a powerful programming model—a sequential metaphor with implicit concurrency—and was simpler to program and debug over contemporary array architectures.

Pioneering work on vector architectures began with Senzig and Smith in their 1965 proposal for VAMP [SS65], an architecture which targeted scientific problems such as global weather prediction. According to Cragon [Cra96], it was the first work to use the term 'vector' in this context. In VAMP, Senzig and Smith identified the problem of disjoint local memories in state-of-the-art array architectures such as the SOLOMON [SBM62] (the basis of the ILLIAC IV [BBK$^+$68]) and proposed a global unified memory space instead. Although it was not a pipelined design, a follow-up work by Senzig [Sen67] suggested the use of high-speed pipelined functional units instead of a networked array of functional units. Senzig even suggested using replicated functional units to decrease the latency of vector instruction execution without exposing this redundancy to the ISA, i.e. decoupling the vector instruction's definition from its microarchitectural implementation. This idea later became better known as vector lanes.

Although VAMP was never built, its related publications surely influenced the first commercial vector architectures, namely the Control Data Corporation STAR-100 [HT72] and the Texas Instruments Advanced Scientific Computer [Wat72]. Regrettably, both of these machines ended up being commercial failures, mainly due to their memory-to-memory design as well as the poor performance of their non-vector, i.e. scalar, functional units. These shortcomings were later addressed by Seymour Cray who developed a high-performance vector machine called the Cray-1 [Rus78] which used a register-to-register design and included fast non-vector functional units. The Cray-1 was highly successful in terms of sales and performance and set the stage for an era between the mid-1970s until the early 1990s where vector architectures were *the* design choice for high-performance machines [EVS98]. Several other companies joined

Cray in the vector supercomputer market including NEC, Fujitsu and Convex. The ideas found in vector architectures matured and evolved over time. We now list some of the features which are characteristic of typical vector architectures. This list is by no means exhaustive; we simply use it to convey a very general idea of what we mean by the term 'vector architecture'.

- A Von Neumann architecture with the ability to execute both scalar and SIMD instructions, often interlaced together.
- A single global memory that is shared by the scalar and SIMD operations.
- SIMD operations typically pipelined through functional units, sometimes with redundant functional units to reduce latencies, i.e. parallel vector lanes.
- One control unit (per logical processor) to govern all concurrent or parallel operations generated by a single SIMD instruction.
- Optionally—but typically—a vector register file in which a single register can hold many values.
- A programmable vector length, i.e. a variable user-controlled number of elements associated with a single SIMD instruction. This is often achieved through a dedicated vector length register.
- Support for non-trivial vectorised memory access patterns, e.g. unit-stride, strided and indexed loads/stores.
- Fine-grained control flow achieved through SIMD predication, i.e. masking, although coarse-grained control flow, i.e. branching, can be used too. Often a complementary vector mask register file is provided to aid this.
- High bandwidth between the processor and memory system.

In the 1990s, vector architectures began to lose their dominance in the high-performance computing market. Since the 1970s, there had been tremendous advances in the microprocessor, i.e. a complete processor contained on a single silicon die. The general-purpose microprocessor had become both sophisticated and inexpensive due to (1)—the opportunities afforded by Moore's Law and Dennard Scaling, and (2)—a high demand from the personal computer market. In contrast, vector machines were typically built in small quantities using exotic and expensive components and, therefore, had fewer economies of scale. Demand for expensive specialised vector supercomputers gradually reduced in favour of highly-parallel machines built from inexpensive off-the-shelf microprocessors. As of 2016, all but one the big players in the vector market have ceased to develop new vector machines. Only NEC continues to produce vector machines through their SX line of supercomputers.

Around the same time as the decline in high-performance vector architectures, there was a simultaneous adoption of very simple vector-like capabilities into several general-purpose microprocessors. Originally proposed by Lee in 1995 as extensions to the PA-RISC architecture [Lee95], these extensions offered simple SIMD functionality with the aim of accelerating multimedia algorithms. Instead of adding a dedicated vector unit to the processor, Lee proposed repurposing the existing architecture's scalar registers and functional units and selectively partitioning them in order to operate on multiple subwords of data in parallel. For example, if the baseline architecture supported integer arithmetic on 32-bit registers, this could be used to operate on two 16-bit integers or even four 8-bit integers. Lee also demonstrated the benefits of having multiple SIMD instructions execute concurrently in a superscalar pipeline thus obtaining ILP and DLP simultaneously. The industry quickly followed suit and these SIMD extensions— commonly dubbed multimedia extensions—began to appear in commercial microprocessors such as Intel's MMX [PW96], AMD's 3DNow! [OFW99] and IBM's AltiVec [DDHS00].

Initially these extensions were very simple, especially compared with contemporary vector architectures. Where vector architectures could process dozens of double-precision floating-point values with one instruction, multimedia extensions were typically limited to a much smaller number of values with less precision. Multimedia extensions typically did not pipeline their operations like vector architectures; instead, the values were operated on in parallel or else broken up into smaller μops, each of which could pass through the functional units as a single unit of work as in the Intel Pentium 4 [HSU+01]. There was limited support for conditional execution and separate mask registers were not provided. The memory access patterns were generally limited to contiguous locations, i.e. unit-stride loads or stores. Finally, there was typically no programmable vector length and the user was forced to operate on the maximum number of elements possible with each SIMD instruction.

Eventually, multimedia ISA extensions became commonplace enough to justify their own register file and dedicated functional units within the architecture's principal pipeline. Although multimedia extensions started out relatively simple, successive generations became more sophisticated offering wider SIMD registers to process more elements per instruction as well as more intricate instructions to operate on them. As an example, Intel's AVX-512 [Int14b] increases the width of its multimedia registers to 512 bits with respect to the previous generation and also includes mask registers, full gather/scatter support and many non-trivial SIMD instructions. This trend is anticipated to continue in the future and the SIMD register width and instruction sets are expected to grow further. For example, AVX-512 was designed with provisions to grow to 1,024 bits. We therefore predict that the SIMD support found in commodity microprocessors will eventually resemble the instruction sets of the classical vector architectures formerly found in the high-performance computing market.

We see a vector architecture as a good candidate for DBMS acceleration for a variety of reasons. Vector architectures are deterministic with self-synchronisation and consequently require less hardware than multicore designs as there need be only one control unit for many concurrent operations. Communication between processing elements and data movement typically requires fewer cycles than in multicore designs. Performance scalability of a vector architecture is generally not inhibited by associative hardware structures as is the case with out-of-order superscalar mechanisms [PJS97]. One the strongest advantages of vector processors is their ability to tolerate long latency instructions, above all, memory operations. As memory latency and bandwidth have become a significant issue for both computer architects and software developers, vector support could be instrumental in optimising databases which are often bottlenecked by the memory system [BMK99].

Vector architectures are known to be energy-efficient [LSCJ06, LAB+11] and can be implemented in microprocessors using simple and efficient hardware [Asa98]. Energy efficiency is achieved through the large reduction of activity in the pipeline as a whole due to the compact representation of many operations with one instruction. Additionally, there is less requirement for aggressive speculation since execution patterns are fully encoded into a single instruction. Vector memory instruction also implicitly encode their access patterns, e.g. one vector load instruction can convey that many consecutive elements in main memory will be accessed together; this can help reduce the reliance on memory prediction hardware structures. Using a vector instruction set also reduces an application's memory footprint as algorithms can be coded with fewer instructions. Leveraging this kind of architecture could be instrumental in building the future generations of green servers for data management where performance and energy consumption are equally important concerns.

Vector architectures have traditionally been used for scientific applications abundant with floating-point code, however, their applicability to business domain, i.e. integer, applications has yet to be analysed. Some former research in DBMSs has shown that various operations do contain some DLP [ZR02, GBY07, HNZB07, IMKN07, CNL+08, KKL+09, PR13]. All of this research, however, was done outside the sphere of computer architecture. This is important because it implies that researchers were working with limited ISAs and hardware and tried to solve their problem within these constraints. We feel that looking at this area from a different angle could be fruitful. In this thesis, we aim to answer the question—*what needs to happen to the algorithms, instruction sets and hardware in order to effectively exploit data-level parallelism in DBMSs?* There is potential to discover new algorithms—previously impossible to implement—by defining new instructions as well as optimising existing algorithms by tailoring the hardware to the available DLP.

We are mindful that there exist other means to exploit DLP; data management performance has also been improved with dedicated devices like FPGAs [MTA09, AANS+14] and GPUs [GGKM06, HLY+09, SHG09]. It is possible that using these accelerators could achieve results more favourable than those presented in this thesis, however, it would not be correct to make a direct comparison for two reasons. (1) FPGAs and GPUs haven't strictly superseded SIMD extensions and there are still advantages with tightly integrated DLP support in out-of-order superscalar processors [PM12], especially when it is preferable not to offload to an external device [GH11] or when there is a fine-grained mixture of scalar and vectorisable code as is common in DBMSs [BZN05]. (2) This thesis evaluates the relative gains of SIMD acceleration within a *single* core. There has been a lot of work done accelerating data management with TLP [RGAB98, LBE+98, KKL+09, PA11] including some of the algorithms examined in this thesis. The work of Lee et al. [LKC+10] shows that TLP on multicore performs comparably with GPUs when the code is optimised for the platform. The purpose of our research is to improve the DLP capabilities within the core itself, nevertheless, TLP and DLP are not mutually exclusive and there is no reason to believe TLP could not be leveraged as well, however, this is beyond the scope of the thesis.

1.2 Objectives of Research

The primary goal of this thesis is to find new, interesting and efficient ways of accelerating OLAP data management operations with vector microprocessor extensions. To achieve this, the overall goal can be broken down into a series of individual objectives. Achieving these objectives will form the contributions of this thesis.

The first objective is to take a set of relevant operations—typically used in data management applications—and transform them to expose DLP. Some algorithms will have straightforward translations. In other cases, existing algorithms may not be the most effective—even when transformed to expose a lot of DLP. For example, it is possible to use sorting network algorithms [Bat68] to expose a lot of DLP to the processor. However, by making this transformation we actually increase the number of dynamic operations to a value larger than would be produced from a more simple scalar sorting algorithm. In cases like these, new algorithms must be designed that can outperform existing DLP-enabled equivalents. In many cases new algorithms may not be possible to implement on existing architectures, therefore, this objective is bound to subsequent objectives and may require new instructions and custom hardware to execute.

The second objective is to take the DLP-enabled algorithms created in the first objective and find the most appropriate instructions to vectorise them. This objective has two parts. (1) To draw from past vector ISAs and cherry pick the optimal instructions in order to effectively capture the available DLP. Many of these instructions will have been seen before, however, it is important to determine which ones are the most effective when working with data management operations. Unlike previous vector incarnations, our focus will be on integer operations rather than floating point. (2) To define novel instructions that have not been encountered before in previous instruction sets. Some of the algorithms from the first objective will not be trivial to implement with existing instruction sets. To efficiently vectorise these algorithms, new instructions should be defined that can capture an algorithm's DLP as concisely as possible. The instructions should be elementary enough to lend themselves to multiple scenarios, however, if an instruction with very specific semantics is found to be useful in the context of data management, it should at least be explored.

The third objective is to create appropriate hardware that efficiently implements the vector instruction set defined in the second objective. The target is to create hardware that has modest power dissipation and low energy consumption as well as being able to execute the vector instructions at a high speed. Furthermore, it is necessary to understand how this hardware scales with respect to both the algorithms and the area/energy/power overheads. Not only do we want to execute the vector instructions in a speedy manner, we also need this hardware to be unobtrusive to the rest of the processor and play nice with a DBMS as a whole. The algorithms are expected to be a fine-grained mixture of scalar and vector code, therefore, it is important to be able to integrate the vector support in such a way that the processor can have intercommunication between vector and scalar parts with low latencies and overhead. A second part of this objective is to discover new microarchitectural techniques to be used in conjunction with this hardware. For example—what are the benefits of using techniques such as register renaming, superscalar execution or a cache hierarchy with our vector ISA? Are there benefits to executing vector instructions out of order and how could such a technique be implemented?

1.3 Thesis Contributions

In this section, we summarise the principal contributions of the thesis. We present four self-contained works that—when combined—complete all of the objectives discussed in Section 1.2. Each work forms the basis of a different chapter and every chapter covers multiple objectives and contains numerous contributions. This section also serves as an outline of the document.

In Chapter 2, we demonstrate that vectors can offer significant performance benefits to existing DBMS solutions. We first characterise a state-of-the-art OLAP DBMS called Vectorwise using an Intel Westmere based system. We profile this application with the TPC-H decision support benchmark and discover that the hash join operator accounts for 61% of its total execution time. We demonstrate that this operator has disproportionate performance returns when scaling or increasing the out-of-order and superscalar parameters in a cycle-accurate simulator. We then take the most significant part of the operator—the probe phase—and find an abundance of DLP that cannot be captured by the system's SIMD multimedia extensions.

Based on these observations, we design our own vector extensions for a modern out-of-order superscalar x86-64 commodity microprocessor. These extensions are inspired by the ISAs of classical vector supercomputers but are tailored and optimised for data management support. We implement these extensions with a custom simulation framework based on PTLsim and DRAMsim2. Our results show significant speedups with good scalability for medium to long vector register lengths and future memory bandwidths.

In Chapter 3, we make a detailed study on sorting—an elementary building block of OLAP DBMSs. There are several known techniques to vectorise and accelerate a handful of sorting algorithms by using vector SIMD instructions. We implement three existing vectorisable algorithms—quicksort, bitonic mergesort and radix sort—to run with our vector extensions defined in Chapter 2. We show the strengths, weaknesses and scalabilities of each algorithm when run on our simulation framework.

Based on these findings, we propose our own non-comparative vectorised sorting algorithm—VSR sort. VSR sort is inspired by radix sort, however, the algorithm circumvents the drawbacks that we identify in the former. To facilitate the execution of this algorithm, we define two new vector instructions and propose a complementary hardware structure for their implementation. The new instructions target DLP which is too irregular for a typical SIMD ISA to vectorise. We discover that VSR sort has very significant speedups over all prior work.

In Chapter 4, we explore different vectorisation techniques applied to GROUP BY data aggregation—another important operator in OLAP DBMSs. Using the infrastructure developed in Chapters 2 and 3, we propose and evaluate five different ways of vectorising data aggregation. We find that although data aggregation is abundant in DLP, it is often too irregular to be expressed efficiently using typical vector SIMD instructions.

In Chapter 3, we discover that exploiting DLP in its irregular form leads to very good results. We now build upon this idea and propose a set of novel algorithms and vector instructions to better capture this irregular DLP in data aggregation. We discover that the best algorithm is highly dependent on the characteristics of the input. Our proposed solution can dynamically choose the optimal algorithm in the majority of cases and achieves significant speedups over a scalar baseline in all cases.

In Chapters 2, 3 and 4, we develop and evaluate vector extensions with performance as the principal focus. In Chapter 5 we look at the area, energy and power costs of these extensions. Using McPAT, we model our baseline architecture and vector additions in order to measure the total area of the microprocessor as well as the area overhead of the vector extensions. After, we use various event counters generated by our simulation framework to feed into McPAT and estimate the energy and power consumption of the scalar and vectorised executables. As a case study, we focus on the sorting algorithms of Chapter 3.

Our results show that our proposed vector extensions come with a very reasonable area overhead, even with a large maximum vector length and lockstepped parallel lanes. We also find that all of the sorting algorithms consume much less energy than their scalar counterpart. In particular, our own novel VSR sort requires an order-of-magnitude less dynamic energy than the scalar baseline. With respect to power, we discover that our vector extensions present a reasonable increase in wattage. Furthermore, we demonstrate that there are configurations of our vector hardware that consume *less* power than the scalar baseline and still achieve very good speedups.

In Chapter 6, we give a recap of these contributions as well as some reflections on the outcomes of this thesis. In Chapter 7, we list the various publications resulting from this work. Finally, Appendix A provides extra runtime statistics for the algorithms evaluated in Chapters 2–4 and Appendix B lists the final ISA developed progressively throughout this thesis.

CHAPTER 2

A Study on Hash Join

2.1 Introduction

In this chapter, we take a top-down approach to accelerating decision support systems (DSSs) on x86-64 microprocessors using purpose-built vector ISA extensions. We first analyse a leading DSS DBMS for potential DLP and propose an instruction set reminiscent of classical vector architectures to capture it effectively. We implement this instruction set using unintrusive modifications to a modern x86-64 microarchitecture that are tailored for the workload. We introduce our cycle-accurate simulation framework—also used in later chapters—which we use to evaluate the ISA and microarchitecture. We find a single operator—hash join probing—is responsible for 41% of the total execution time of the TPC-H DSS benchmark. Our results show performance speedups between 1.94× and 4.56× for an implementation of this operator run with our proposed hardware modifications.

Vectorwise [Act11] is a modern OLAP database management system designed for DSSs. It is a commercial product based on the work of MonetDB/x100 [BZN05]—a block-at-a-time query engine which is hardware-conscious and highly optimised for modern superscalar microprocessors. Vectorwise identifies the bottlenecks of previous database solutions and structures their own software to exploit the full capabilities of modern commodity hardware. Vectorwise algorithms are designed to reduce branch misprediction penalties and function call overheads. Where possible, it uses block partitioning to optimise its algorithms for the data cache. Its functions are designed to be data-level parallel in order to expose independent loop iterations to the underlying microarchitecture. Vectorwise can store tables in a columnar fashion [CK85] meaning that columns of a table are stored as arrays in memory. When the algorithms access data like this, it can help to expose DLP and generate more regular memory access patterns.

In order to keep the processor fully utilised, Vectorwise transforms a lot of potential DLP into ILP. This is principally due to the simplicity and limitations of multimedia

SIMD extensions which cannot effectively capture the available DLP. While there are still performance gains achieved through the DLP⇒ILP transformation, optimising software this way is neither the most efficient nor scalable solution. Modern microprocessors found in servers generally have out-of-order superscalar microarchitectures and can cope with ILP to some extent. The problem is that the hardware complexity and power consumption of finding more independent instructions this way increases quadratically [PJS97] making this an unscalable solution which is not suitable in the long term.

In this chapter, we use a top-down methodology and profile Vectorwise in order to find opportunities to use vector technology. From this, we identify software bottlenecks caused by unscalable superscalar hardware structures. We propose new integer-based vector ISA extensions for x86-64 that can capture the available DLP in a concise and effective manner. These ISA extensions are implemented using simple and scalable hardware which we evaluate using a cycle-accurate microprocessor simulator fused with a highly-detailed memory simulator. We compare major design decisions against alternative options both qualitatively and quantitatively. Our vectorised implementation achieves performance speedups between 1.94× and 4.56× for hash join probing—the most significant part of the DBMS. In contrast, our experiments show that increasing the out-of-order superscalar resources offers very little benefit to the same algorithm.

This chapter is structured as follows—Section 2.2 characterises the application and provides motivation for this work. In Section 2.3, we discuss the design and implementation of the proposed vector extensions. Section 2.4 describes the experimental setup and introduces our simulation framework. In Section 2.5, we present the results of various experiments related to the design space and performance. Section 2.6 compares and contrasts our proposal with related work. Finally in Section 2.7, we conclude the chapter.

2.2 Software Characterisation

To get an idea of the application's characteristics, we run Vectorwise v1.0 with TPC-H [Tra11]. TPC-H is the standard benchmark for evaluating decision support systems. Unlike other benchmarks such as SPEC, TPC-H is not a set of individual applications but instead a set of queries that stress different aspects of an OLAP DBMS implementation. The benchmark defines the database tables and their relations (schemas), the values contained in the database, and the queries to be evaluated over the database. The DBMS software has the freedom to store the database in its preferred way and evaluate the queries in a manner that it sees fit. Therefore, what we present here is one *particular* evaluation of Vectorwise.

Figure 2.1 shows the CPU time of the 22 [1] TPC-H queries executed on a database of 100 GB on an Intel Westmere system with 16 GB of DDR3-1333 memory. The results show that a significant amount of time is spent in the hash join operation. If all the queries are evaluated together and their total execution time is accumulated, hash joins then account for 61% of this time. This has motivated us to focus our

[1]Query 21 was not run due to a larger memory requirement than what was available.

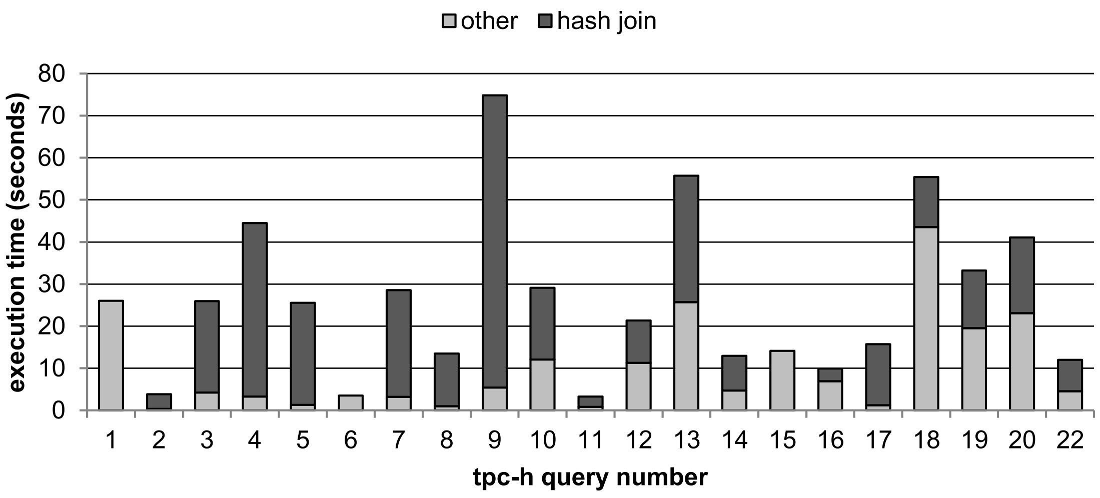


Figure 2.1: Breakdown of the TPC-H benchmark with 100 GB database.

initial evaluation on this operation and—in particular—the probe phase of the join which constitutes 67% of the time spent in hash joins and 41% of the total execution time. Although this chapter focuses on one particular operation, we expect many of our findings to be applicable to other aspects of the DBMS. This is due to the way that Vectorwise has been implemented using a block-at-a-time column-oriented query engine. This software organisation helps expose more DLP than is possible with traditional volcano-style DBMS designs [GM93].

2.2.1 Hash Join Probing

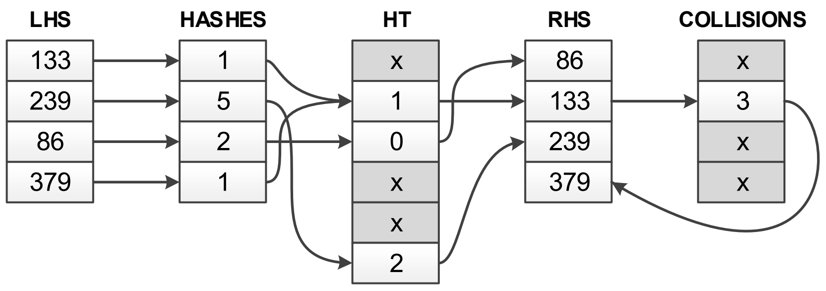


Figure 2.2: Probe phase of hash join.

Vectorwise's hash join probe—illustrated in Figure 2.2—works in several stages and rounds. The ultimate goal is to match and join the keys from the left-hand side (LHS) with those on the right-hand side (RHS). A portion of the LHS—called a block—is processed together as a compromise between cache locality and function call overhead. First, the keys of the block are hashed to create indices into the hash table (HT) structure. The corresponding values in the HT are—in turn—used as indices into the RHS. The values must be retrieved and compared against the keys from the LHS for matches. It is possible for distinct keys to hash to the same index thus leading to bucket collisions which must be handled appropriately; it can be seen in the example that keys 133 and 379 cause such a collision. If the match fails, an auxiliary structure named collisions is checked. If there was a bucket collision, the entry will chain to the next colliding key, otherwise the entry is empty which implies there are no collisions and potentially no possible matches. For a more detailed explanation of the Vectorwise hash join implementation, we refer the reader to [Ż09, SZB11].

Vectorwise's hash probing uses block partitioning to expose more independent operations to the compiler/microarchitecture and amortise function-call overheads. In contrast to other operations in Vectorwise, it is more difficult to achieve data locality and cache residency. It is necessary that the hash table be fully built before it can be probed, thus, the entire right-hand side must be evaluated before any work on the left hand side can begin. If the right-hand side has many values, the hash table becomes large and reduces the opportunity to effectively use the cache. For example, the tables used in one particular hash join found in query 9 of TPC-H amount to 86 MB; these tables are queried by over 600 million keys leading to performance issues.

Each row of the join is independent with respect to another, thus making the algorithm potentially data-level parallel. The structures are stored contiguously as arrays in memory, however, due to the random access nature of the algorithm, indexed memory operations are necessary, i.e. `gather/scatter`. Mask registers could be used to optimise many of the operations. For example, when only a subset of keys from the LHS block are matched with the RHS, these could be masked out of subsequent iterations to avoid redundant checks in the collisions table. Vectorwise's hash join probe implementation frequently rearranges its block partitions to filter out data unnecessary for subsequent rounds. The vector `compress` instruction coupled with a mask population count and programmable vector length could very useful when vectorising these parts.

2.3 Design and Implementation

In this Section, we design a vector instruction set based on our analysis of the hash join probe algorithm in Section 2.2.1. We then discuss the various design choices of the architecture and implementation details of the microarchitecture.

We choose x86-64 as a base ISA to build upon; this was chosen for several reasons. It is the leading ISA in the server market, having roughly 60% of the market share [Int11]. It is a universal ISA with mature optimising compilers and toolchains. x86-64 is also a large improvement over the archaic IA32 ISA. Many improvements have been made, e.g. the number of general purpose registers is doubled and several legacy features—such as memory segmenting—have been removed. Vectorwise, although not exclusively written for x86-64, has several optimisations made for x86-64 and the Intel Xeon 5500 series [Ing09].

The baseline microarchitecture is not taken from any specific incarnation of x86-64. Instead, the features available from PTLsim [You07], a cycle-accurate x86-64 simulator, are used. PTLsim models an aggressive superscalar out-of-order microarchitecture with instruction-to-μop translation; multistage pipelines; speculation and recovery; and a multi-tiered cache hierarchy. We use Intel's Westmere microarchitecture [Int14a] as a reference when choosing particular configuration parameters—the same microarchitecture used to make the initial evaluations in Section 2.2.

2.3.1 Instruction Set Proposal

Here we outline our ISA extensions for true vector SIMD support. The ISA offers eight vector registers—$vr0 \rightarrow vr7$. Each vector register can store up to the same number of elements defined by the maximum vector length (MVL) constant. The actual number of elements that a given instruction operates on depends on the value of the vector length (VL) control register which is managed explicitly by get/set instructions. The ISA also provides an instruction that sets the VL to the MVL. Retrieving the MVL at runtime allows for transparent scaling of the microarchitecture; if the vectorised functions are written using loop strip mining, they may be able to take advantage of larger vector register lengths without the need of rewriting or recompiling. This is precisely how we made our vector scalability experiments.

To vectorise hash join, unit-stride and indexed memory access patterns are needed. Each pattern is supported with its own *load*, *store* and *prefetch* instructions. Strided memory instructions are not necessary and are omitted from the ISA in this chapter, however, these would be useful if the baseline DBMS were row-oriented instead of column-oriented. Individual elements that comprise a vector memory operation are assumed to be independent of one another; stores/scatters always write to unique locations and scatters with conflicting indices are left semantically undefined.

Instructions are classified and listed in Table 2.1. Complete definitions of each one can be found in Appendix B. The ISA includes vectorised integer arithmetic and bitwise logical instructions. For these, one of the source operands must be a vector register and the other may be another vector register or a scalar register. There is a class of initialisation instructions which can set all of the elements within a vector register to a specified scalar value. A very useful variant of this—known as `iota` [SFS00]—is also included. This instruction generates a vector of consecutive integers starting from a specified value. `iota` is useful for dynamically generating indices used to access the hash join structures.

Table 2.1: Overview of added vector instructions.

class	instructions
integer arithmetic	`add`, `subtract`, `multiply`
bitwise logical	`and`, `xor`, `shift right`
comparison	`not equal`
initialisation	`set all`, `clear all`, `iota`
mask	`set mask`, `clear mask`, `and`, `or`, `not`, `popcount`
permutative	`compress`
vector length	`set`, `set MVL`, `get`
memory fence	`scalar-vector`, `vector-scalar`, `vector-vector`

Many of the instructions can take an optional vector mask specified by one of four available mask registers—$mr0 \rightarrow mr3$. Vector masks are updated in three ways—(1) with initialisation instructions, e.g. `set mask` and `clear mask`. (2) With vector comparison instructions that write their boolean results to mask registers. (3) With mask-mask logical instructions.

The ISA also includes a position manipulation instruction called `compress` [Kog81], which condenses non-masked elements from one vector register contiguously into another vector register. To complement this, a mask population instruction—`popcount`—is included which counts the number of set bits in a mask register. These instructions are useful for eliminating rows that have no potential entry in the hash table as well as shortening the vector length when checking candidate matches.

In order to achieve better performance, the vector memory instructions have been made weakly ordered with respect to one another. This way, the execution order of vector loads and stores is not deterministic and ordering must be achieved through explicit fence instructions. This allows for more aggressive scheduling in the microarchitecture as well as reduced hardware complexity due to the absence of memory aliasing checks. Although this puts more pressure on the programmer, we find that the vast majority of hash join's memory accesses are independent of one another. Weak ordering guarantees can also be found in Cray's NV-2 ISA [ABS+07].

The proposed ISA is reminiscent of classical vector ISAs used in supercomputers except with emphasis on integer support over floating point. A true vector ISA is already known to be useful for scientific computing and multimedia processing [Asa98, EVS98] as well as other areas [EAE+02] thus broadening the scope of applicability of our work. Additionally, there has been related work [HLY+09, MK00, ZR02] that shows DLP opportunities in DBMS software beyond hash join that could also be exploited with an ISA like this. In subsequent chapters, we use this newly created ISA as a basis to explore other algorithms.

2.3.2 Design Decisions

In the following subsection, we describe and justify a list of key design decisions of the implementation of our vector extensions. Figure 2.3 provides a block diagram of the microarchitecture. The shaded area on the left represents hardware additions to the baseline x86-64 architecture.

2.3.2-a Out-of-Order Execution

One of the biggest design decisions we made is to allow vector instructions to issue out of order. The work of Espasa et al. [EVS97] showed that additional performance can be gained by using register renaming and out-of-order execution. An out-of-order execution engine can begin memory operations early and utilise the memory ports much more efficiently hence hiding long memory latencies. Vectors are already tolerant of long memory latencies in their own right; combining them with an out-of-order core can further enhance this quality.

There are also drawbacks to an out-of-order microarchitecture. The structures used to achieve out-of-order execution don't scale well and are very power hungry [PJS97].

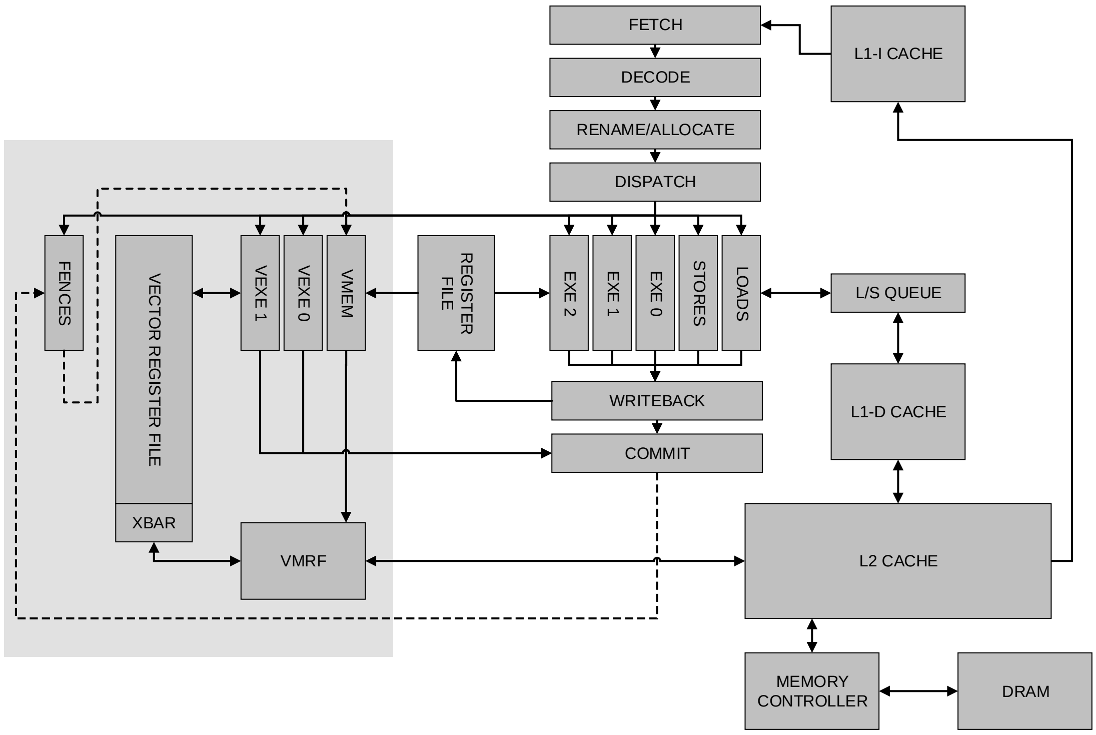


Figure 2.3: Block diagram of the baseline microarchitecture extended with vector support.

Fortunately, a single vector instruction can represent a lot of work and reduce the need to scale these structures more than what already exists in current commodity out-of-order microprocessors. Our decision to allow issuing vector instructions out of order affects many of the subsequent design decisions. In Section 2.5.1-a, we evaluate the benefit of using the out-of-order mechanism against a simpler decoupled design.

2.3.2-b Cache Integration

The block-at-a-time processing technique used by Vectorwise is very conscious of the cache hierarchy. As mentioned in Section 2.2, large structures like hash tables often have trouble fitting in the cache hierarchy, however, many other structures can still reside there comfortably. Of particular importance are the block partitions which flow through various data operators and create intermediate results in cache-resident arrays. For this reason, it is highly desirable to take advantage of the cache hierarchy when possible.

A solution to integrating vector support into an existing superscalar processor was proposed by Quintana et al. [QCEV99]. A major part of this work was integrating the vector units with the existing cache hierarchy. Their novel solution involved bypassing the level 1 data (L1D) cache altogether and going directly to the level 2 (L2) cache. The main motivation behind this was that adding the logic necessary to support vector loads at the L1D cache could compromise its access time as seen by the scalar units.

This idea was later used in Tarantula [EAE$^+$02] which had a 4 MB banked L2 cache directly accessible by its vector memory units.

Accessing the L2 directly introduces potential coherency problems with the L1D cache. In the same article, Quintana et al. describe a simple approach to resolve this. This involves adding an extra bit to each line to mark if its data is exclusively owned by the scalar units or the vector units.

Since unit-stride loads and cache lines match quite well, this solution can pull many elements from the cache at once and hide the additional latency incurred by the L2. The L2 cache also has a larger capacity than the L1D cache so there is the added benefit of having a larger cache-resident working set. For these reasons, we choose to use L1D bypassing for our vector extensions. In Section 2.5.1-c, we evaluate and compare L1D bypassing against an alternative approach that instead accesses the L1D cache directly.

2.3.3 Microarchitecture Implementation

It is desirable to reuse as much as possible from the base microarchitecture so the additions necessary to implement the vector ISA can be minimal. One of the key design decisions we made has been to integrate the vector units into the core itself. This way, the new vector instructions can make use of the existing pipeline and supporting hardware structures. Nevertheless, we have had to introduce some new hardware structures in addition to modifying several existing ones.

The decode units had to be modified to incorporate the new vector ISA. These changes were minimal as the new instructions all have a fixed length and begin with the same prefix (described in Appendix B). The register rename tables had to be changed to accommodate the new vector, mask and vector-length architectural registers. A new physical register file was added in order to support vectors and masks. The vector length register was simply mapped to the existing integer physical register file.

Three new clusters have been added—vmem, which executes vector memory instructions, and vexe 0 and vexe 1, which handle non-memory vector instructions. Each cluster contains an issue queue with eight entries and various pipelined integer functional units. The existing issue queues can handle up to four operands which is sufficient for existing x86-64 instructions; the new vector instructions need two extra operands, i.e. a total of six. This is for two reasons—(1) the vector length register is allowed to be renamed and thus it is necessary to have it as an operand in the issue queue. (2) The destination register is also a source register. This is because it is possible for the vector instruction to overwrite part—but not all—of its destination register. This occurs when the VL is shorter than the MVL or if the instruction masks out operations on some of the vector's elements. The scalar issue queues using four operands can coexist with the vector issue queues using six operands.

Misspeculation recovery piggybacks on the existing infrastructure of the out-of-order core. Vector registers are renamed using the same mechanism as scalar registers. On branch mispredictions, the register rename table is restored to a stable state before fetching from the correct path. This is similar to the approach used in [EVS97]. Vector stores can generate their addresses when issued but don't modify the memory state until they are the oldest instruction ready to commit.

2.3.3-a Fence Mechanism

Since the new vector memory instructions are weakly ordered, it is mandatory to use fence instructions to enforce ordering when required. To implement these vector fences, it is necessary to continually log the youngest, i.e. most recent, store instruction in the pipeline. This way, when a fence instruction is decoded it can be made dependent on this store assuming it has not already retired. Subsequent load instructions can use this fence as a dependent operand. The fence instruction waits in its issue queue until its source operand—the store instruction—has successfully committed to memory. At this point, the fence instruction proceeds to issue and wakes up any of the dependent load instructions waiting in the memory issue queues. To achieve this functionality, we add three new structures.

The first two structures are placed and used between the rename/allocate and dispatch stages of the pipeline. The first of these is a small table with two entries used to record the tag, i.e. reorder buffer id, of the most recent vector and scalar stores. The second structure is a queue which we use to initialise and manage the fence instructions. Similar to the reorder buffer, entries need to be allocated and deallocated in this queue when decoding and committing fence instructions. Each entry of the queue contains the tag of the associated fence instruction as well as the type of fence it implements, i.e. *vector store → vector load*, *vector store → scalar load* or *scalar store → vector load*. When a scalar or vector load instruction passes through the pipeline's front end, the queue is checked for the presence of a fence which matches the criteria, e.g. scalar loads need only check for *vector store → scalar load* fences and can safely ignore other types. If there is one or more matching fences present in the queue, the memory instruction is made dependent on the youngest one. With respect to hash join probing, we have found that a queue with four entries is sufficient. This requirement could increase for other algorithms, especially if the fence instructions were to occur more frequently.

The third structure is a new issue queue dedicated entirely to fence instructions. In our baseline microarchitecture, instructions broadcast their tags to dependent instructions after they have completed execution, i.e. passed through the functional unit. This is problematic for store instructions since their execution only achieves address generation. We don't want a fence to issue until the store it depends on has been fully committed to memory. To solve this, we add a path from the commit stage of the pipeline to our custom fence issue queue. This is illustrated with the dashed lines in Figure 2.3.

In Section 2.5.1-a, this fence mechanism is compared against a completely fenceless approach and in Section 2.5.1-b, against a more a naïve hardware implementation.

2.3.3-b Vector Memory Request File

Vector memory requests have to be handled differently from all of the other operations. Scalar memory loads can be executed out of order, but to achieve this, a complex associative hardware structure called the load/store queue (LSQ) must be used. The

LSQ detects memory aliases, i.e. loads and stores that go to the same address which may have incorrect behaviour when issued out of order. Using the LSQ for vector memory instructions would limit the number of in-flight memory operations in the microprocessor. Additionally, such a structure presents a significant design problem for handling indexed memory operations in which a single instruction may access a large number of disjoint memory locations. Vector memory operations are known to be data independent at the element level and in many cases are also data independent with respect to one another. Since the infrequent case of vector memory aliasing is handled explicitly using fence instructions, it does not need transparent resolution in hardware.

It is also important to take advantage of the regular patterns found in vector memory operations. Unit-stride loads and stores access consecutive locations in memory and thus have a lot of spatial locality. It is therefore preferable to work with whole cache lines when possible. The LSQ as it exists does not take advantage of this locality and each entry ultimately refers to a single scalar value. For indexed memory operations with less spatial locality, it is also important to reduce the penalties that may be incurred from transferring unnecessary data.

We have designed a structure called the Vector Memory Request File (VMRF) to effectively manage vector memory instructions while avoiding the complex associative hardware found in the LSQ. This structure is optimised for data transfers at the granularity of a cache line and can take advantage of the fact that the majority of vector memory requests are independent. Here we outline the VMRF's structure and mechanism.

The VMRF contains three non-associative tables—the Load Table (LT), Store Table (ST) and the Cache Line Table (CLT). Every vector load instruction is allocated a single entry in the LT; similarly, each vector store instruction is given one entry in the ST. Since we anticipate a moderate number of in-flight vector instructions, these tables can be kept reasonably small. The CLT is used by both vector loads and stores and is responsible for tracking and managing cache line transfers to and from the vector register file. Since a single vector memory instruction can generate many cache line requests, this structure will be significantly larger than both the LT and ST.

When a vector memory instruction is decoded, one entry in the LT or ST is allocated. An entry contains fields that point to the reorder buffer entry of the memory instruction as well as the physical register of the source or destination. Additionally, each entry contains two bitmasks—clt-waiting and clt-resolved. Both of these bitmasks contain a number of bits equal to the number of entries in the CLT.

When the vector memory instruction is executed, i.e. its addresses are generated, the VMRF will allocate an entry in the CLT for every L2 cache line accessed. For a unit-stride memory instruction with a VL of 64, this may generate four or five CLT entries assuming a datatype of 32 bytes. For indexed memory instructions, up to MVL CLT entries could be used. Each CLT entry has an implicit identifier which is simply its offset in the table. When the VMRF allocates an entry in the CLT, it uses that entry's identifier to mark the equivalent bit in the clt-waiting bitmask in the LT or ST.

After the address generation, the reorder buffer entry of the vector load or store is placed in a waiting state. The VMRF then completes the load or store without stalling the principal pipeline. In each clock cycle, the VMRF attempts to resolve a single valid entry in the CLT. In the case of CLT entries relating to vector stores, the store must be the oldest instruction in the pipeline before they are considered valid. Once the values in a cache line are successfully transferred to or from the physical register file, the VMRF uses the CLT entry's identifier to set the corresponding bit in the clt-resolved bitmask. When the clt-waiting and clt-resolved bitmasks match, it means that the entire vector memory operation has been completed. At this point, the VMRF wakes up the reorder buffer entry of the vector load or store and frees the entries in the LT, ST and CLT.

We add two buses that connect the physical register file to the L2 cache—one for load requests and another for store requests. As an optimisation, the structure can handle partial cache line transfers. The cache line is broken into discrete sectors—the size of an L2 cache line divided by the width of the bus. To save bus cycles, only necessary sectors need to be sent to or from the cache line. Each CLT entry specifies which bytes within the cache line are actually needed. Indexed operations benefit from this, especially if the number of required bytes per cache line is small. Reorder buffer entries of vector loads and stores must contain an identifier into the VMRF to be able to recover from misspeculation. In these cases, the allocated entries in the VMRF tables are annulled and recycled.

It is controversial to add indexed vector memory instructions to an out-of-order microprocessor. There are often reservations about doing this, especially about compromising the latency of scalar memory instruction which could affect the performance of existing non-vectorised applications. We have made two important design choices to circumvent this from happening—(1) the LSQ is untouched by vector instructions and, more importantly, avoids the complexities that would arise when detecting aliasing between indexed memory operations. (2) We leave the interface between the functional units and the L1D cache untouched and instead bypass this structure and access the L2 cache directly instead.

2.4 Experimental Setup

2.4.1 Simulators

We have evaluated our experiments using PTLsim [You07]—a cycle-accurate x86-64 simulator. The experiments were conducted using the *classic* mode of PTLsim, i.e. where system calls are emulated. We have extended the simulator extensively to incorporate the new vector instruction set, hardware structures and additional microarchitectural changes.

PTLsim uses a fixed latency memory model by default which does not model bandwidth and contention issues at all. It was felt that for a memory-intensive algorithm like hash join, it is paramount to model the memory accurately. Consequently, we have integrated DRAMSim2 [RCBJ11]—a cycle-accurate memory system simulator—into PTLsim and replaced the default memory model. This also allows us to experiment with multiple memory controllers. Having an accurate memory model allows the vectorised algorithms to work within a realistic bandwidth envelope thus enforcing a fairer comparison to non-vectorised algorithms. Our results in Section 2.5.3 show large discrepancies between the default simplified model of PTLsim and the more accurate model using DRAMSim2.

2.4.2 Default Parameters

This section lists the parameters of the baseline setup. In all the experiments that follow, these parameters are used unless explicitly stated otherwise. The parameters of the scalar baseline are based on Intel Westmere [Int14a], the same microarchitecture used to profile Vectorwise in Section 2.2.

Table 2.2: Simulator superscalar and out-of-order parameters.

parameter	value	parameter	value
fetch width	4	fetch queue	28
front end width	4	front end stages	17
dispatch width	4	writeback width	4
commit width	4	reorder buffer	128
x86-64 general purpose registers	16	physical registers	256
issue width per cluster	1	total issue width	5
issue queue entries per cluster	8	total issue queue entries	40
load queue	48	store queue	32
L1D outstanding misses	10	L2 outstanding misses	16

Table 2.2 lists the superscalar parameters as well as the sizes of various structures in the microarchitecture. Here *front end* refers to fetching, decoding, renaming/structure allocation and dispatching to clusters. Based on the measurement of branch misprediction penalties in the work of [Fog12], we estimate the latency of the front end to be 17 cycles.

In Westmere, the equivalent to an issue queue is the reservation station—a single structure with 36 entries shared by all the clusters. In PTLsim, it is not possible to model clusters as well as a single shared reservation station. To solve this, we divide the reservation station into five issue queues, each one with eight entries and assigned to its own cluster. We have measured the impact of larger issue queues with up to 32 entries and have found a difference in performance of only 3%. As shown in Figure 2.3, there is one cluster for loads, one cluster for stores and three general purpose clusters—exe 0, exe 1 and exe 2. As in Westmere, all functional units within the clusters have a single-cycle throughput, i.e. they are fully pipelined. For details on the functional unit latencies, we refer the reader to [Int14a].

Table 2.3 shows the parameters used in the cache hierarchy. The given latencies include address generation. The hierarchy is inclusive and write through with respect to L1D $\rightarrow$ L2, but writeback with respect to L2 $\rightarrow$ memory. Although Westmere has a larger shared L3 cache, we do not include it as the effects of multiple cores are not modelled in this work.

Table 2.3: Simulator cache hierarchy parameters.

cache level	size	latency	line size	ways	sets
l1 instruction	32 KB	1 cycle	64 bytes	4	128
l1 data	32 KB	4 cycles	64 bytes	8	64
l2 unified	256 KB	10 cycles	64 bytes	8	512

Table 2.4: Simulator memory system parameters.

parameter	value	parameter	value
type	DDR3-1333	clock	1.5 ns
memory controllers	1	DRAM bandwidth	10 GB/s
queue	per rank per bank	scheduling	rank then bank
transaction queue	64	command queue	256
policy	open page	burst length	64 bytes
banks	8	ranks	4
rows	32,768	columns	2,048
row accesses	8	device width	4
address layout	row:rank:bank:column:burst		

Table 2.4 contains the parameters of the memory system. We model a memory system with both one and two memory controllers. The memory modules are DDR3-1333 with a cycle time of 1.5 ns; since our CPU frequency is taken to be 2.67 GHz, the memory controllers are clocked once every four CPU cycles. The burst length is taken as 64 bytes as this coincides with the line sizes of the cache hierarchy. We use an open page policy, however, we find that using a closed page policy results in only marginally less overall performance. This may be important when considering energy consumption where the closed page policy can be more beneficial [JNW07].

Table 2.5: Simulator vector extension parameters.

parameter	value	parameter	value
maximum vector length	64	VMRF load table entries	12
lockstepped parallel lanes	1	VMRF store table entries	8
maximum datatype width	64 bits	VMRF cache line table entries	128
architectural vector registers	8	physical vector registers	16
architectural mask registers	4	physical mask registers	8
bus width: L2 $\rightarrow$ vector	32 bytes	total vector issue width	3

Table 2.5 shows the default configuration of the vector parameters. The number of physical vector registers has been made twice the amount of architectural vector registers. This is based on the work of Espasa et al. [EVS97] that states for register renaming to be effective, there should be—at minimum—twice as many physical re-

gisters to architectural registers. We have noted that eight architectural vector registers are more than enough to vectorise these kernels. Six architectural registers would be sufficient meaning that the physical register file could be reduced to 12 entries.

All experiments presented in the next section use a single vector lane (i.e. parallel lockstepped pipelines used to operate on elements within a single vector instruction). Our experiments have shown that adding more lanes improves performance only marginally since hash join is dominated by memory requests. In subsequent chapters, we introduce lanes into our experiments as they have a larger impact on other algorithms. Additionally, chaining (i.e. allowing some vector instructions to issue as soon as the first elements of an input operand are ready rather than waiting for all of the elements to be calculated first) does not exhibit significant performance gains and has thus been disabled.

We make the bus width between the L2 cache and the vector register file 32 bytes—the same as the bus width that connects the L1D cache to the L2 cache. We provide the cache line table of the VMRF with 128 entries. This number was chosen to allow at least two indexed memory operations in flight when the MVL is 64, however, this structure could be reduced when the MVL is shorter.

As mentioned in Section 2.3.3, three additional clusters have been added for vector support—vexe 0, vexe 1 and vmem. Therefore, one vector memory instruction and two non-memory vector instructions can be issued in the same cycle. Each cluster requires one write and two read ports to both the vector register file and the mask register file. A vector instruction must complete fully before another one can occupy the same functional unit.

2.4.3 Workload

We evaluate the proposed changes using a partial run of a hash join probe found in query 9 of TPC-H—the most time-consuming operation of the entire benchmark. This join uses two keys to query a hash table of 32 MB, a conflict table of 18 MB and RHS indices of 36 MB (totalling 86 MB). The LHS input was originally 600 million rows but we reduced this to 12 million in order to shorten the simulation times. This will be just as representative since the LHS input data is distributed in such a way that the selectivity of the query will remain fixed whether the LHS is evaluated entirely or partially. We observe that useful performance metrics such as the instructions per cycle and the cache miss ratio are invariant to the size of the LHS input. In the proceeding experiments and results, this query run with the default input is labelled **tpch**.

In order to evaluate the algorithm in different scenarios, we have added four extra synthetic datasets—**l1r**, **l2r**, **2mb** and **huge**. **l1r** and **l2r** are built such that the hash table, conflict table and RHS indices can be resident in the L1D and L2 caches respectively. Since the LHS input does not have temporal locality, its number of rows need not be reduced. **2mb** is eight times the size of **l2r**, thus allowing for a mixture of cache hits and misses in the experiments. **huge** has structures of an equal size to **tpch** but with a different selectivity that leads to more computational work, i.e. the LHS finds more matches in the RHS. In order to compare against **tpch**, the LHS input is fixed at 12 million rows for all the datasets.

We evaluate these datasets using a vectorised binary as well as a purely scalar one, i.e. the original code compiled without autovectorisation. Our vectorised binary is coded and optimised by hand using the proposed ISA extensions. The vectorised functions retain their semantics and minimal transformations are needed. This way, a comparison against the scalar implementation is fair and representative. We configure Vectorwise to use blocks of 1,024 elements—a good compromise between cache locality and function call overhead.

2.5 Results

In this section, we present the results of several key experiments used to evaluate the impact that our vector extensions have on the hash join probe algorithm. We choose to present our results in terms of either processor cycles or speedup; this allows us to fairly compare the purely scalar simulations with the vector ones. CPI/IPC metrics are not used because they don't translate well to something analogous with the baseline architecture. A single vector instruction is not comparable to a single scalar instruction. For example, l1r run with the scalar baseline commits 1,163 million instructions whereas the vectorised version run with a MVL of 64 commits only 51 million instructions of which 16 million are vector instructions. Even treating a single vector instruction as the number of scalar instructions equivalent to the VL is not fair due to (1)—the decrease in pipeline structural pressure, and (2)—the reduction of related bookkeeping scalar instructions such as loop constructs and conditionals.

2.5.1 Design Exploration

Figure 2.4 displays the results of various experiments related to the design and implementation space. These experiments are run with the vectorised binary using the setup described in Section 2.4. ooo-customfence-l2cache refers to the default configuration with out-of-order logic, customised fences and L1D cache bypassing. decoupled restricts the out-of-order capabilities of the vector issue queues and permits only the oldest instruction, i.e. at the head, in each cluster to issue. The scalar issue queues are still fully out of order to isolate the impact of dynamic scheduling applied to vector instructions. Vector memory instructions can still issue speculatively so the fence instructions are still necessary. fenceless restricts the out-of-order capabilities in the same way as decoupled and additionally limits the in-flight vector memory instructions to remove the necessity of fences entirely. flushed-fence replaces the custom fence mechanism with a simpler but slower alternative. l1cache forces vector memory instructions to communicate directly with the L1D cache instead of the default bypass mechanism. Both flushed-fence and l1cache use the default out-of-order issue queues. For clarity, these results are presented with absolute numbers using simulated processor cycles of execution; accordingly, the lower the value–the better the result. We now discuss the results of each of these design choices individually.

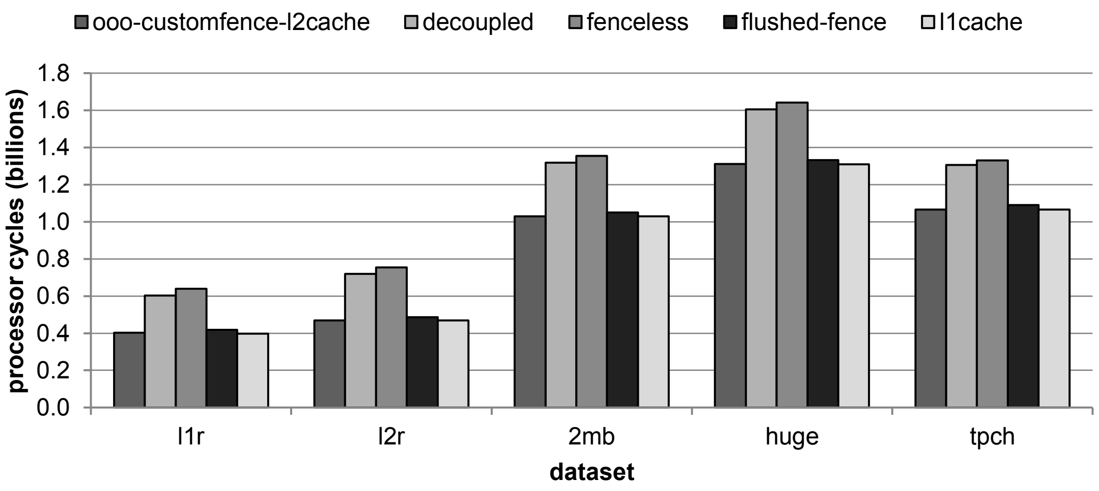


Figure 2.4: Hardware design space exploration running the vectorised hash join probe binary when $MVL = 64$.

2.5.1-a Out-of-Order Logic

Here we quantify the benefits of the out-of-order vector issue queues. It is immediately apparent that the out-of-order capabilities of ooo-customfence-l2cache outperform both of the more restricted configurations—decoupled and fenceless. Taking into account all the datasets, ooo-customfence-l2cache takes 75% of the number of cycles of decoupled and 72% of fenceless or, alternatively, gives 1.34× and 1.39× speedups respectively. Although restricting the scheduling policy simplifies the processor, the out-of-order logic allows the instruction stream to execute more aggressively and start independent vector instructions earlier thus utilising the available execution units more efficiently. It must be restated that out-of-order support does add complexity to the microarchitecture, however, our proposed design attempts to piggyback on the existing out-of-order support and reuse as much as possible from the scalar core.

2.5.1-b Custom Fences

Next, we evaluate the benefits of the custom fence logic described in Section 2.3.3-a. We compare ooo-customfence-l2cache against a simpler but slower alternative—flushed-fence. flushed-fence uses PTLsim's internal mechanism for creating a true instruction stream barrier. This is typically used to service hardware assists, i.e. special x86-64 instructions that cannot be decoded into µops. When the decoder encounters one of these instructions, the processor stops fetching new instructions, drains the pipeline, establishes a correct hardware state and then services the instruction. The semantics of our fence instructions are changed to evoke this behaviour.

The difference in performance is not as significant as we originally anticipated. On average, the custom fence mechanism outperforms flushed-fence by 1.03×. This may be explained by the fact that the fence instructions are not very frequent and occur off the critical path. Each phase of the algorithm finishes by storing a block of temporary results to memory. A different phase begins by loading these temporary results iteratively into the vector registers. The fence instructions, therefore, need only be placed between the different phases of the algorithm. The true difference between flushed-fence and ooo-customfence-l2cache in this scenario is that the latter allows instructions to decode and dispatch but the former does not, however, in both cases nothing will be able to issue until the fence has committed. The penalty of using flushed-fence over ooo-customfence-l2cache would become more apparent with a longer front end pipeline or if fences were to be needed more frequently on the critical path.

2.5.1-c Level 1 Data Cache Bypass

Here we measure the benefit of L1D cache bypassing. For the l1cache design, the VMRF uses the L1D cache in comparison to ooo-customfence-l2cache which instead goes directly to the L2 cache. The benefit is that any data resident in the L1D cache can be transferred to a vector register in fewer cycles. The disadvantage is that on a cache miss, an extra cycle is needed to request the missing data from the L2 cache. It must also be stated that in this evaluation, the scalar access time to the L1D cache remains unchanged. A vector access takes the same latency as a scalar one, however, transfers to the vector register file are still restricted to 32 bytes per cycle. The reality is that a direct vector access to the L1D cache could compromise the access cycle time as discussed in [QCEV99], thus the l1cache design may be more optimistic than it should be.

The results show that—on average—l1cache has a negligible speedup over ooo-customfence-l2cache. This can be explained by the fact that, internally, the VMRF still needs to generate the same number of requests to the caches. When accessing the L2 cache, these are pipelined and the penalty is amortised. Additionally, the majority of the workloads don't comfortably fit in the L1D cache which, in turn, has fewer outstanding misses available than its L2 parent. We conclude that going to the L2 cache in lieu of the L1D cache adds very little penalty and ensures that the existing scalar performance is not compromised.

2.5.2 Vector Scalability

It is desirable to have a large average vector length (AVL), as this is directly related to the scalability of the vectorised code. Figure 2.5 shows the trend of the AVL of the **tpch** dataset when increasing the MVL. The horizontal axis varies the MVL, i.e. the number of elements that can be contained in a vector register. The vertical axis shows the AVL normalised to the MVL, this way the scalability of the algorithm is made clear.

The AVL is calculated by dividing the total number of elements processed by vector operations divided by the total number of vector instructions. It has two variants—the AVL including masked out elements (inc.masked) and another excluding these (ex.masked). The former only considers the programmable vector length register. The results show that the AVL degrades gradually with larger MVLs, however, not too rapidly. We conclude that it is worth experimenting with large MVLs such as 64.

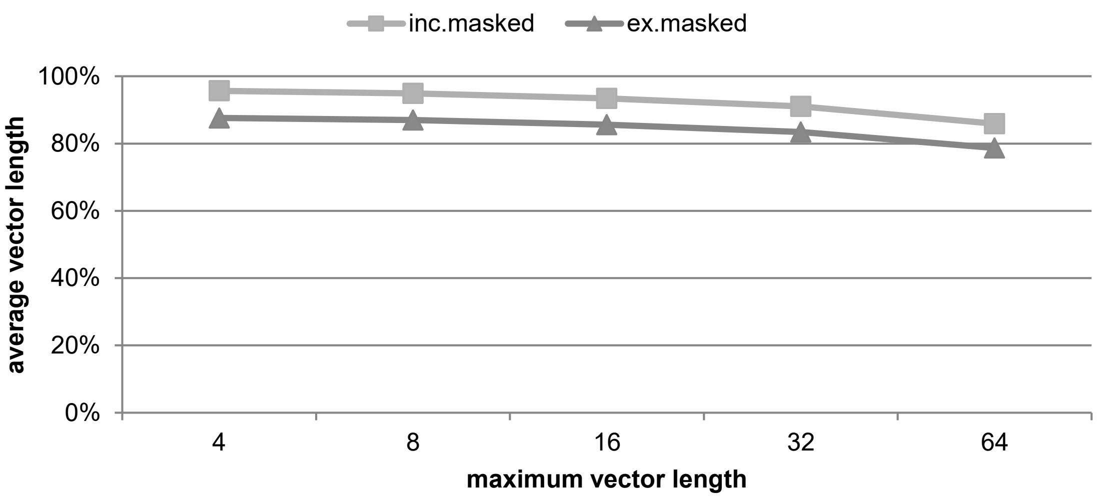


Figure 2.5: Average vector length compared to maximum vector length using **tpch**.

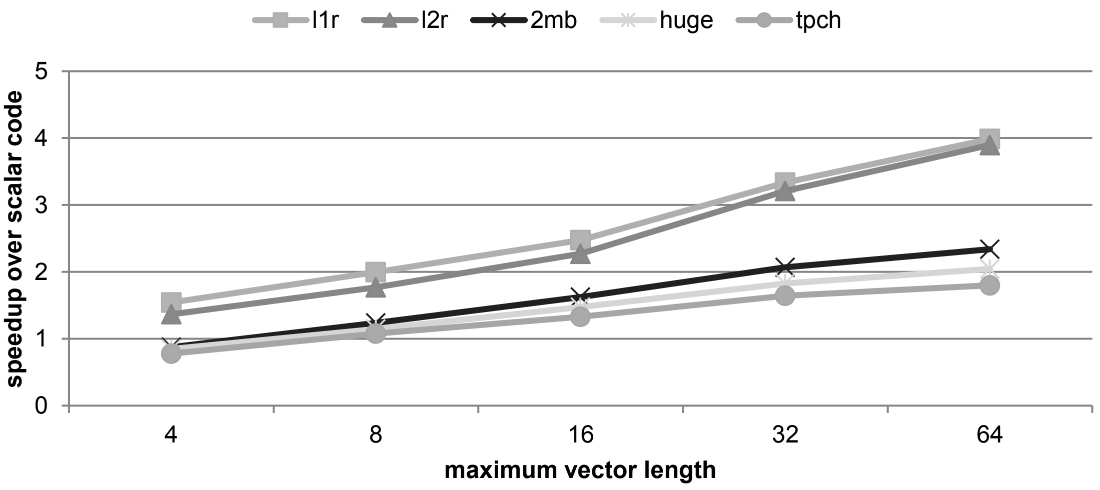


Figure 2.6: Scalability of vector code when increasing the maximum vector length.

Figure 2.6 shows the performance benefits of increasing the MVL for all of the datasets. The horizontal axis doubles the MVL at each increment and is shown on a logarithmic scale. The vertical axis shows the speedup of the vectorised code over the scalar equivalent. The speedup shown for each line is relative to the scalar baseline run with that particular dataset, i.e. each dataset has its own baseline.

l1r and **l2r**—the two cache-resident datasets—see the greatest benefit of a larger MVL with speedups of 4.0× in the best case. **2mb**, **huge** and **tpch**—the noncache-resident datasets—also scale with the MVL, albeit more slowly. When increasing the MVL from 32 elements to 64, the average performance increase of the cache-resident datasets is 1.2× whereas for the noncache-resident datasets it is 1.1×. Depending on the expected input size, it may be more economical to have a smaller MVL.

Increasing the size of the MVL is very significant to the performance speedups, even for single-lane configurations. For the l1r experiments, a MVL of four yields a speedup of 1.5× over the scalar baseline, however, changing the MVL to 64 increases the speedup to 4.0× without using additional lanes. We observe that the number of cycles reduced in l1r run with a MVL of 64 is very close to the number of cycles that the front end cannot dispatch due to full clusters when the MVL is four. With a larger MVL there are fewer instructions, and although an instruction with a MVL of 64 has a higher latency than an instruction with a MVL of four, the aggregate time is lower due to the vector startup penalty being paid less frequently. In general, there are less structural hazards leading to higher throughput.

It is interesting to note that the non cache-resident datasets—2mb, huge and tpch—run with a MVL of four perform worse than their scalar equivalents. It is known in vector research that there is a break-even vector length below which the vectorised operation needs more time than the equivalent scalar operation [Sch87b]. This can be explained by the penalty of going directly to the L2 cache which isn't yet fully amortised with such a small MVL. Since it exhibits little benefit over the scalar baseline, we discard four as a MVL from our experiments in subsequent chapters.

The vector solution is particularly good at describing the independence of individual operations, expressing them in a compact manner and scheduling them back to back. The scalar implementation still suffers from inter-instruction dependencies and stifles the potential of faster scheduling, especially with respect to memory instructions. The vector implementation reduces the number of instructions fetched, decoded, renamed, issued and committed as well as their occupancy in structures such as the fetch queue, issue queues and the reorder buffer. Appendix A contains extra runtime data of these experiments.

2.5.3 Memory Controller Saturation

Figure 2.7 shows the results of the vectorised code run with different memory configurations. Here, the tpch dataset is measured and the vectorised algorithm's speedup is shown relative to its scalar baseline. The diagram plots three trends—inf. bw, mc1 and mc2. inf. bw shows the relative performance when using PTLsim's default fixed latency memory model. This is configured at 150 cycles per memory request which is the average load memory latency of the scalar version reported by DRAMSim2. This model is considered to be infinite in bandwidth as it does not model contention, variable latencies, bandwidth nor any of the quirks found in a realistic memory system.

Comparatively, mc1 shows the same experiment run using an accurate DRAM model. For a MVL of 64, mc1 reports 1.84× performance over the scalar baseline whereas inf. bw discloses 3.4×—a massive discrepancy. The vector unit had been saturating the memory system with requests which in turn did not have enough bandwidth to sustain the requirements. mc2 shows the same experiment run with an additional memory controller used to increase the available bandwidth. Although it still falls short of the controversial inf. bw trend, it allows the speedup to increase to 2.61×.

The effectiveness of vector support applied to hash join probing comes from its ability to saturate the memory controllers with requests. Figure 2.8 shows the effects

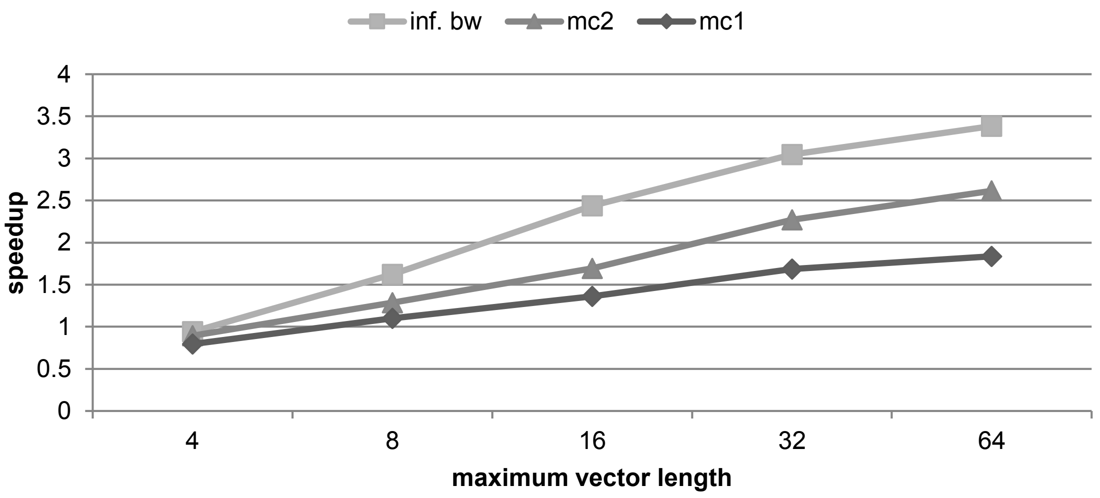


Figure 2.7: Impact of memory bandwidth on performance using tpch.

of increasing the maximum number of outstanding last level cache misses by increasing the number of miss status holding registers (MSHRs). Here the MVL has been fixed at 64 elements. The results are shown as the speedup over the scalar version with one memory controller and the default number of MSHRs using the tpch dataset. Here, s- and v- refer to the scalar and vector experiments respectively.

It can be seen that the scalar version does not show any speedup with the addition of a second memory controller nor with the infinite bandwidth memory model. s-mc1, s-mc2 and s-inf.bw do not exceed 1.0× even when extra MSHRs are offered. The scalar version of the algorithm may be able to take advantage of the available bandwidth in the system if it were able to generate its requests quicker. We have found that the scalar code uses about 3.5 GB/s of effective bandwidth out of DDR3-1333's maximum theoretical of 10 GB/s.

In comparison, it can be seen that there is little performance gain when the number of MSHRs is increased for the vector versions using a realistic memory model. The reasons for this are different to those of the scalar version. Clearly the vector version can generate a sufficient number of requests to main memory, otherwise the infinite bandwidth memory model v-inf.bw would not exhibit a speedup when more MSHRs are provided. The simulations that model detailed memory controllers don't exhibit additional speedup with more MSHRs because the memory resources are already strained.

The vector code with a single memory controller achieves 6.2 GB/s of effective bandwidth, however, it is normal for an application to peak at around 70% of the maximum theoretical bandwidth. When operating close to the application's maximum sustainable bandwidth, latencies tend to increase exponentially—this behaviour is described in detail in [JNW07, SZG+09]. Seeing this plateau of available bandwidth motivated us to experiment with an additional memory controller.

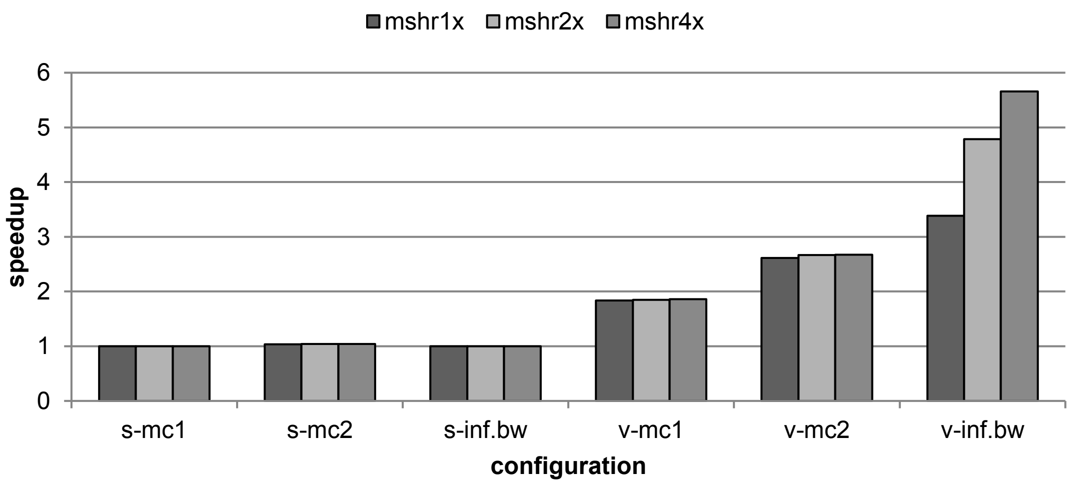


Figure 2.8: Speedups while varying memory bandwidth and MSHRs using tpch and $MVL = 64$.

2.5.4 Scalar Scalability

To illustrate that vectors are an appropriate solution to this problem, Figure 2.9 shows the effects of increasing the superscalar and out-of-order structures listed in Table 2.6. All the datasets are measured using the three new hardware configurations—ss2, ss4 and ss8—and the results are presented as the speedup over the baseline ss1 configuration. It must stated that it is extremely unrealistic to presume these parameters can be scaled in such a way, however, it makes for an interesting experiment and exposes the limitations of a purely scalar approach.

Table 2.6: Scaled superscalar and out-of-order simulator parameters.

parameter	ss1	ss2	ss4	ss8
fetch queue	28	56	112	224
load queue	48	96	192	384
store queue	32	64	128	256
reorder buffer	128	256	512	1,024
issue queues (total)	40	80	160	320
outstanding l1d misses	10	20	40	80
outstanding l2 misses	16	32	64	128
front end width	4	8	16	32
dispatch width	4	8	16	32
writeback width	4	8	16	32
commit width	4	8	16	32

It can be seen that doubling the parameters once increases the performance 1.16× on average, which is a minor gain considering the resources required to achieve this speedup. Increasing the hardware structures 8× will increase the performance between 1.22× (for l1r) and 1.40× (for **2mb** and **huge**). This means in the best case, a huge out-of-order superscalar design can yield an extra 40% of benefit at best whereas the simpler vector model can increase performance past 400%—an order of magnitude in difference.

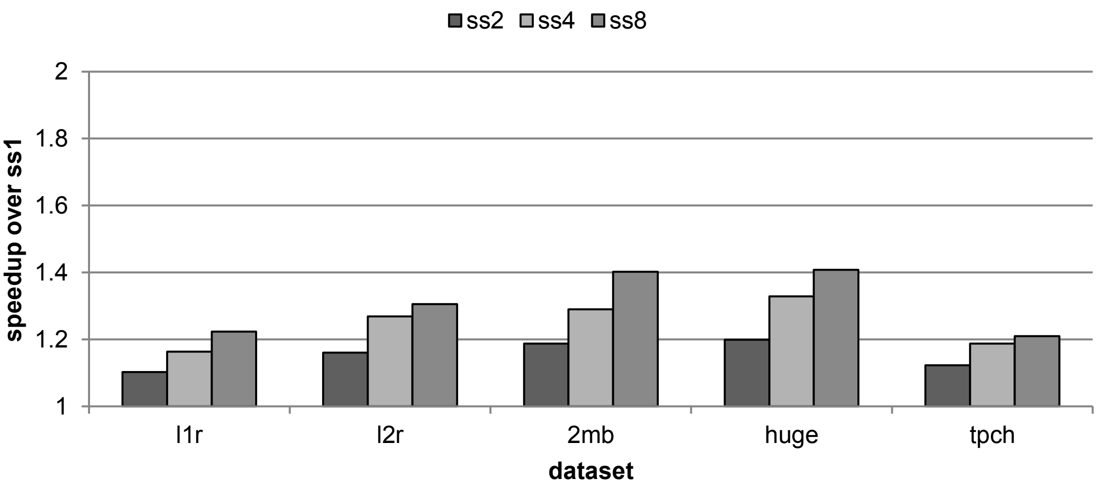


Figure 2.9: Speedup of scalar code when increasing superscalar capabilities.

2.5.5 Software Prefetching

The work of Chen et al. [CAGM04] showed the potential of increasing hash join performance using software prefetching. This is a particularly appealing solution as it takes advantage of existing hardware found in commodity processors. x86-64—the baseline ISA in these experiments—includes a set of software prefetching instructions defined by the SSE standard. We modify the scalar and vector versions of hash join probing to use the group prefetching technique described in [CAGM04].

Figure 2.10 shows the results of the experiments made with each dataset. s-pre is the scalar code with software prefetching enabled. v-no-pre is the default vectorised code without software prefetching modifications. v-pre is the vectorised code with software prefetching additions. The vector configurations use a MVL of 64. All experiments are presented as the relative speedup over the scalar baseline without software prefetching for that particular dataset.

It can be seen that s-pre improves the performance of the algorithm with an average speedup of 1.34×; this is quite a good performance boost considering it requires no additional hardware. That said, v-no-pre achieves much better speedups—between 1.8× for **tpch** and 4.0× for l1r and l2r—hence showing that the vector approach still has higher returns than a scalar version with prefetching. v-pre shows that, for all of the datasets, prefetching combined with the vectorised code pushes the performance even more. In the case of l2r, performance exceeds 4.5×. This helps demonstrate that the performance gains of software prefetching can be complementary to the proposed vector additions. It is important to note that for l1r and l2r, it is only the RHS that is cache-resident; the LHS is larger than the cache and prefetching helps reduce the effect of cold misses.

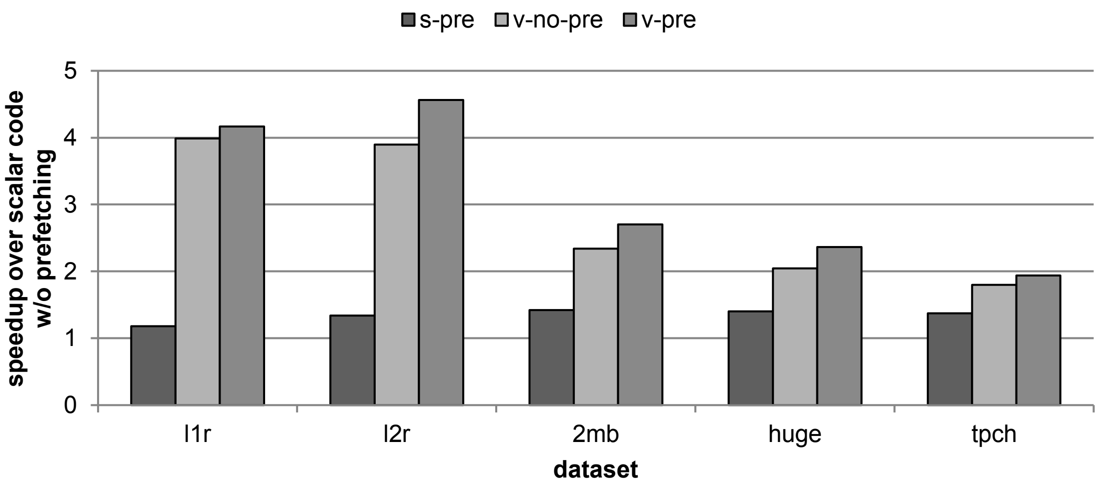


Figure 2.10: Speedups achieved with use of software prefetching when $MVL = 64$.

2.5.6 Comparison to SSE4.2

So far, different aspects of the proposed hardware have been evaluated and compared against a purely scalar baseline. What is missing is a comparison against a hash join implementation that utilises the multimedia extensions already present in the baseline architecture. The problem is that no version can exist given the limitations of SSE4.2—the multimedia extensions found in Westmere. The work of Kim et al. [KKL$^+$09] made extensive optimisations to—and an evaluation of—the hash join algorithm and concluded that for DLP to be exploited effectively, there must be efficient support for indexed memory operations.

Nevertheless, we measure the difference in execution time between an optimised scalar version of the code and a version with the compiler's autovectorisation feature enabled. In order to help the autovectorisation algorithm, we alter the functions to expose data alignment and the absence of aliasing. The compiler is able to transform a small portion of the code to use SIMD instructions, namely the part that computes the hashes of the LHS input. When both versions are run on the same system, the difference in execution time is less than 1%. This is mainly due to the fact that this particular part of the code is not the most dominant in the overall algorithm.

2.6 Related Work

This section details several works that have attempted to accelerate DBMS software by exploiting DLP. We compare and contrast our own research with those mentioned.

The work of Martin [Mar96] describes a hash join implementations for the Cray C90. The methodology of this work takes a different approach to ours. We profile an existing full-featured DBMS that has been optimised for modern out-of-order microarchitectures to find bottlenecks due to scalar inefficiencies. In contrast, this work proposes its own algorithm for hash join with no reference to a real DBMS. We are proposing vector extensions to an ISA that already dominates the server marker whereas Martin's work is done exclusively on a supercomputer. Meki and Kambayashi [MK00] also look at the vectorisation of database operators. This time the list is expanded to selection, projection and join operators, however, their methodology is still the same—naïve scalar implementations are run against vectorised versions on a supercomputer and so the same arguments still apply. Since we are conducting our experiments within a simulation framework, we have been able to characterise the hash join algorithm in terms of hardware scalability—something not achieved in either of these works.

Zhou and Ross [ZR02] make a broad study accelerating various database operators using the SSE instruction extensions for x86. The work investigates the benefits of DLP and reduction of conditional branches in implementations of scans, aggregations, indexed operations and joins. Since this chapter is primarily focused on joins, a comparison of this feature is given. The principal difference between our vectorised join and their join implemented with SSE is that their work looks at a simple nested loop implementation whereas our work looks at an optimised hash join implementation. A nested loop join compares every row from the LHS table with every row from the RHS table. This is not a problem when the tables are small, but if they are large then this is a very inefficient join algorithm. In contrast, we look at a hash join implementation that is suitable for large tables typically found in OLAP DSS databases.

Héman et al. [HNZB07] partially port a DBMS to the Cell Broadband Engine [GHF$^+$06]—an architecture abundant with DLP capabilities. What is interesting is that the query engine used in the study is MonetDB/X100 [BZN05] which is an earlier version of the query engine used in Vectorwise. The work mostly discusses the challenges that arise from using this esoteric architecture. Furthermore, the work is evaluated using TPC-H query 1 which lacks a join operation. This chapter is primarily focused on joins so it is difficult to make a comparison.

He et al. [HLY$^+$09] investigate the performance benefits of running DBMS operations on graphics processing units (GPUs). The study includes a hash join implementation that runs on a GPGPU coprocessor. There are some performance benefits, however, the study concludes that the necessity to transfer data between the global memory and the GPU's local memory can be a large bottleneck. Our approach adds vector processing capabilities into the CPU's execution core so this penalty is never encountered. This is important when treating the DBMS software as a whole since it can have complex control flow mixed with segments more suitable for DLP-oriented hardware.

2.7 Conclusions

In this chapter, we have examined a leading decision support DBMS—Vectorwise—and found that hash join can form a significant proportion of its execution. Using the TPC-H decision support benchmark, we measure that 61% of Vectorwise's total execution time is spent in the hash join operator. It was found that the probe phase of hash join contains an abundance of DLP that isn't expressible using the multimedia SIMD extensions found in the baseline architecture. We have proposed instruction set extensions to the x86-64 ISA that are suitable for vectorising the algorithm compactly and efficiently. We have introduced these instructions into a modern out-of-order x86-64 microarchitecture taking advantage of existing structures where possible and without compromising their performance.

We have explored various trade-offs in the design space. Our decision to issue vector instructions out of order gives a 1.34× performance speedup over a decoupled design. We have also evaluated the benefits of using fences to allow vector memory instructions to issue out of order without hardware alias checks. We have shown that this gives 1.39× extra performance over a model that restricts its vector memory instructions to issue non-speculatively and serially. Finally, we have measured the penalty incurred when bypassing the L1D cache in favour of using the L2 cache which we have found to be negligible.

Our results show that the new vectorised implementation of hash probe—accounting for 41% of total execution time—can achieve speedups between 1.94× and 4.56× over the scalar baseline. We have shown the benefits of using two memory controllers in conjunction with the vector hardware and have also demonstrated that the scalar code cannot take advantage of the extra available bandwidth. Furthermore, we have shown that increasing the out-of-order structures and superscalar widths gives disproportional returns whereas the vector approach achieves an order-of-magnitude greater speedup. Finally, we have confirmed that software prefetching techniques described in [CAGM04] can accelerate the scalar algorithm, albeit not as much as our solution using vectorisation. We have also demonstrated that this strategy is complementary to our work and can be used in conjunction with vectorisation for an even greater speedup.

This chapter serves a basis for the rest of the thesis document. Using our newly created vector ISA and simulation infrastructure, we explore sorting algorithms in Chapter 3 and data aggregation algorithms in Chapter 4. In Chapter 5 we focus on measuring the area overhead and power consumption of these new hardware extensions.

CHAPTER 3

A Study on Sorting

3.1 Introduction

Sorting is a widely studied problem in computer science and an elementary building block in database management systems. There are many unique ways to achieve sorted output and each technique has its own particular strengths and weaknesses. Numerous works have successfully leveraged DLP to accelerate sorting algorithms [Sto78, Lev90, ZB91, GBY07, IMKN07, CNL$^+$08, SKC$^+$10]. As current SIMD support found in microprocessors is still quite restrictive and the transformation from simple multimedia extensions to true vector support is still incomplete, the true potential of exploiting the DLP found in sorting algorithms is hitherto unknown.

In this chapter, we make three principal contributions. (1) We first study prior DLP-accelerated sorting proposals with a modern and uniform platform and assess them using consistent metrics thus allowing us to identify and compare the strengths and weaknesses of each algorithm. In particular, we use the vector extensions developed in Chapter 2 to explore vectorised implementations of quicksort, bitonic mergesort and radix sort. (2) Based on these evaluations, we propose a novel non-comparative sorting algorithm—VSR sort—a highly efficient vectorised implementation of radix sort that overcomes many of the drawbacks found in the evaluated sorting algorithms. (3) To facilitate this algorithm, we propose two new instructions as well as a hardware implementation that includes both a serial and parallel variant. Additionally, we suggest several other uses for the new instructions and hardware.

Based on experiments, we report that our VSR sort outperforms the aforementioned prior work and shows good scalability for large maximum vector lengths. Furthermore, it exhibits good performance using both a simple single-lane pipelined mechanism as well as with a more sophisticated implementation using lockstepped parallel lanes. We show that VSR sort has maximum speedups over a scalar baseline between 14.9× and 20.6×. On average it performs 3.4× better than the next-best vectorised sorting algorithm when run on the same hardware configuration.

The outline of this chapter is as follows. Section 3.2 discusses the changes we make to our architecture and simulation framework to accommodate the sorting algorithms. Section 3.3 outlines the experimental methodology and evaluates three sorting algorithms that leverage DLP. Section 3.4 describes VSR sort; we propose implementing this algorithm by extending the vector ISA with two new instructions and recommend two different ways of realising these instructions in hardware. The algorithm is then evaluated and compared with those of Section 3.3. Related work is discussed in Section 3.5. Finally, Section 3.6 concludes the chapter.

3.2 Changes to the Architecture

In this section, we outline the changes that we make to our vector architecture and simulation framework. In Chapter 2, we defined a baseline architecure as well as a vector ISA suitable for DBMS acceleration and various microarchitectural techniques and optimisations. In this chapter, we build upon this infrastructure and extend it further in order make detailed evaluations of the vectorised sorting algorithms.

In Chapter 2, we chose not to show the results with lanes because they did not make a significant impact on the execution time of the vectorised hash join. For sorting—depending on the particular algorithm—lanes can have a very significant influence on performance. In this chapter, we introduce lockstepped parallel lanes into our microarchitecture. We experiment with a small number of lanes to determine which algorithms are affected by their presence as well as by how much.

We also found that eight vector registers were more than sufficient to vectorise the hash join probe algorithm. For sorting, we find that several of the algorithms can take advantage of an increased number of registers. To ensure that each algorithm operates optimally, we increase the number of architectural vector registers from eight to sixteen. Consequently, we also increase the number of physical vector registers from sixteen to 32.

In order to vectorise hash join probing, we required unit-stride and indexed memory access patterns. In this chapter, we extend our vector memory instructions to include a strided access pattern. These memory instructions use a base address in addition to a parameter that refers to the increment in memory between elements. This allows for the loading to—or storing from—adjacent elements in a vector register from—or to—non-adjacent locations in main memory. Although indexed memory instructions already serve this purpose, a strided memory instruction is more compactly encoded and conveys more information regarding the regularity of its access pattern. As such, it is amenable to more optimisations in hardware. Our implementation of radix sort uses a strided load as one of its dominant instructions; we therefore opt to provide a good implementation of this instruction rather than emulating it with gathers.

We also expand our vector ISA to include more instructions that are necessary to vectorise the sorting algorithms. Instructions are classified and listed in Table 3.1 and those highlighted in bold indicate that it is newly introduced in this chapter. Complete definitions of each one can be found in Appendix B. The `shuffle` instruction is an all-to-all permutation instruction with three input operands—two vector registers that contain source values and a third vector register that encodes the shuffle pattern. The `merge` instruction—also known as blend or select—creates a new vector of values by selecting individual elements from two possible input vectors based on the state of a mask. Unlike the `shuffle` instruction, the selected elements do not change their relative order so the instruction can be implemented using simple hardware. Each instruction now requires $\frac{VL}{lanes}$ cycles to pass through a functional unit, with the exception of the mask, vector length and get/set element instructions.

Table 3.1: Overview of vector instructions with new additions for sorting.

class	instructions
integer arithmetic	`add`, `subtract`, `multiply`
bitwise logical	`and`, `xor`, `shift right`, **`shift left`**
comparison	`not equal`, **`less than`**, **`greater than`**
initialisation	`set all`, `clear all`, `iota`
mask	`set mask`, `clear mask`, `and`, `or`, `not`, `popcount`
permutative	`compress`, **`shuffle`**, **`reverse`**
vector length	`set`, `set MVL`, `get`
memory fence	`scalar-vector`, `vector-scalar`, `vector-vector`
other	**`merge`**, **`get element`**, **`set element`**

3.3 Evaluation of Existing Sorting Algorithms

This section presents an evaluation of three existing vectorised sorting algorithms—quicksort, bitonic mergesort and radix sort. The former two algorithms are classified as comparative sorts whereas the latter is a non-comparative sort. While comparison sorts generally have fewer limitations and facilitate simpler implementations, non-comparison sorts are still suitable in many scenarios at the expense of tailoring the algorithm to a specific datatype. Each algorithm chosen is suitable for a DLP-accelerated processor but they have very different characteristics from one another. Their strengths and weaknesses are found experimentally and used to guide our own algorithm in the next section.

Each algorithm is evaluated with three datasets, a MVL varying between eight and 64 elements and between one and four lockstepped parallel lanes. Note that we no longer use a MVL of four since in Chapter 2 we found that in many cases it has a negative effect on performance. Based on these initial experiments, we further evaluate some of the algorithms with additional configurations. All results use the metric cycles per tuple (CPT)—the total execution time of the algorithm in cycles divided by the length of the input n.

We evaluate each algorithm with three datasets—**small**, **medium** and **large** which contain 51,200, 512,000 and 5,120,000 tuples respectively. There are several reasons why these datasets have been chosen. Firstly, **small** is able to reside in the L2 cache while the latter two increase in length by one and two orders of magnitude respectively.

This helps identify performance issues related to the input size and also to pinpoint any noteworthy effects due to the cache. Secondly, these input lengths are multiples of the MVL, however, they are not perfect powers of two. Using perfect power of two lengths can often lead to outliers in trends. We have observed some strided memory patterns impacting performance negatively when n is a power of two and for this reason we prefer to show the general case rather than the exception.

Each dataset contains a random uniform distribution of 32-bit integer values. Although it is true that some sorting algorithms exhibit different behaviours when the input is not uniformly distributed, out of the algorithms presented in this work only quicksort would be affected. A random uniform distribution will at least present quicksort's average case behaviour.

Merely sorting an array of integers has very limited applications, we therefore create the datasets with both key and payload values. This opens up the applicability of these algorithms to DBMSs. The two-value tuples are 64 bits each and have been organised as a structure of arrays which is common in column-store OLAP DBMSs [ABH+13, CK85].

A purely scalar algorithm called **reference** is also included with the results of each vectorised sorting algorithm. This allows for a common baseline when comparing the vectorised algorithms to one another. We use an in-place quicksort using a median of three pivot selection and an insertion sort cleanup; this variant of quicksort is known to perform well [Sed78]. We have optimised all these algorithms by hand and have used software prefetching when beneficial.

3.3.1 Quicksort

Quicksort [Hoa62] is a comparison sort that uses a divide and conquer strategy to iteratively partition its input until it is sorted. It is well known for having fast implementations due its $O(n \cdot log_2 n)$ average complexity, cache-friendly memory access patterns and the simplicity of the algorithm's main body.

A vectorised version was first proposed by Stone for the CDC STAR [Sto78]. Its implementation is recursive, stable and has relatively few operations per pass. It is not in-place and thus requires an auxiliary array to store partial results. The algorithm has a unit-stride memory access pattern thus making it very bandwidth friendly. We optimise this algorithm further by using the median of three technique for choosing the pivot as suggested by Sedgewick [Sed78]. The pseudocode is shown in Figure 3.1; we have simplified it by omitting any vector stripmining code.

Figure 3.2 displays the results of this algorithm. The maximum speedup over the reference benchmark is 1.4×, 1.6× and 1.8× for **small**, **medium** and **large** respectively. Going from a MVL of eight to sixteen increases the performance in all cases and is more pronounced than subsequent increases of the MVL. This can be explained by the fact that sixteen 32-bit elements accessed in a unit-stride fashion is exactly one cache line. Increasing the number of lanes can help the algorithm's performance a little bit as the `compare` and `compress` instructions can benefit. Adding more than two lanes yields very little extra performance; at this point the memory operations dominate

```
function QUICKSORT($\vec{a}, alen$)
    $pivot \leftarrow$ median_of_3($a[0]$, $a[alen-1]$, $a[\frac{alen}{2}]$)
    $mask \leftarrow$ compare($\vec{a} > pivot$)
    $rlen \leftarrow$ popcount($mask$)
    $llen \leftarrow$ popcount($\neg mask$)
    $\vec{t} \leftarrow$ compress($\vec{a}$, $mask$)
    $\vec{u} \leftarrow$ compress($\vec{a}$, $\neg mask$)
    $\vec{a} \leftarrow$ concatenate($\vec{u}$, $\vec{t}$)
    if $llen > 1$ then QUICKSORT($\vec{a}_{\langle 0:llen-1 \rangle}$, $llen$)
    if $rlen > 1$ then QUICKSORT($\vec{a}_{\langle llen:alen-1 \rangle}$, $rlen$)
end function
```

Figure 3.1: Pseudocode for the vectorised quicksort algorithm.

the execution time and none of these can be accelerated with lanes, i.e. there are no indexed memory operations.

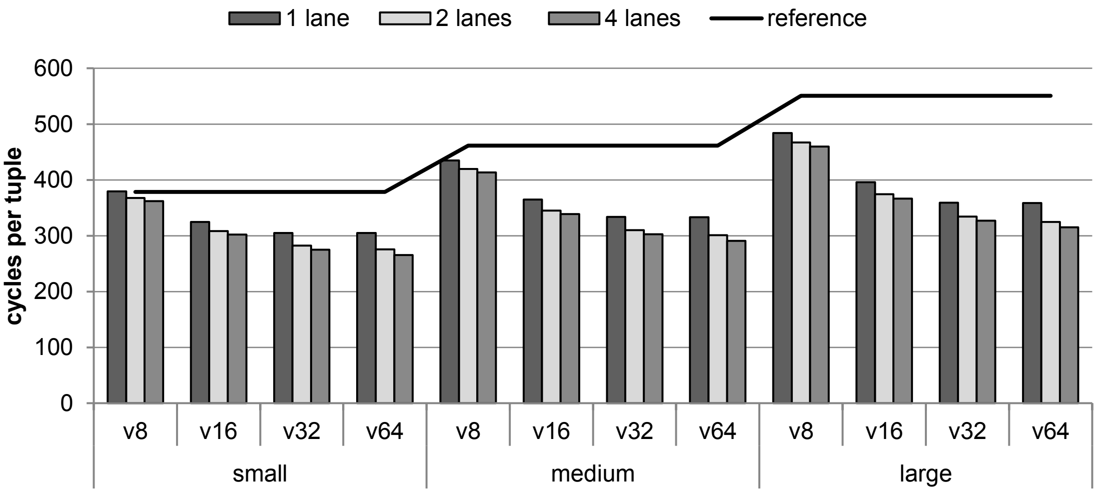


Figure 3.2: Performance results for quicksort.

Increasing the MVL from 32 to 64 only improves the performance marginally due to quicksort's divide and conquer strategy. As partitions get smaller, the effective vector length per function call is reduced and eventually causes serialisation which leads to an increased CPT. Table 3.2 illustrates this for a MVL of 64 and the **large** dataset. The leftmost column shows the percentage of time spent processing partitions with a length in the range of from and to. call frequency displays the number of calls to the function operating on a partition with a length within the specified range. The highlighted column—average cpt—indicates the average CPT measured in the main body. Going from top to bottom, it can be seen that the CPT remains similar until reaching partitions of 64 elements or fewer. At this point the vector registers become underutilised and serialisation begins. It can be seen that 62% of the total execution time is spent operating on partitions with fewer elements than the MVL.

Table 3.2: Quicksort's performance per partition.

percent of time	from	to	call frequency	average cpt
15.6%	5,120,000	119,417	72	6.31
10.6%	119,416	2,786	3,091	4.43
6.2%	2,785	228	35,350	4.99
5.3%	227	65	94,389	8.56
10.4%	64	19	306,189	18.43
19.8%	18	6	849,856	41.11
32.1%	5	1	2,100,685	92.15

To help alleviate this problem, we use a variant of the algorithm. The work of Levin [Lev90] modifies the original algorithm and uses a vectorised odd-even transposition (OET) sort [Hab72] as a cleanup mechanism. When the quicksort phase creates partitions equal or less to some predefined threshold, it gracefully returns instead of continuing recursively. When this modified quicksort finishes, OET sort is applied to the entire dataset. Typically this algorithm has a complexity of $O(n^2)$, but in this case it is only $O(n \cdot \textit{threshold})$. We have found empirically a value of 16 to be optimal for the threshold.

Figure 3.3 displays the results of the modified algorithm. The maximum speedup over the reference benchmark is 2.4×, 2.5× and 2.6× for **small**, **medium** and **large** respectively. On average, these results are 1.4× better than Stone's quicksort implementation. A huge advantage of OET sort is that it operates on the entire dataset without concerning itself with partition boundaries and therefore leverages the entire MVL of the configuration. A disadvantage is that the cleanup algorithm suffers from a complexity of $O(n \cdot \textit{threshold})$ and we have observed that increasing the threshold higher than 16 will cause the overall performance to decrease.

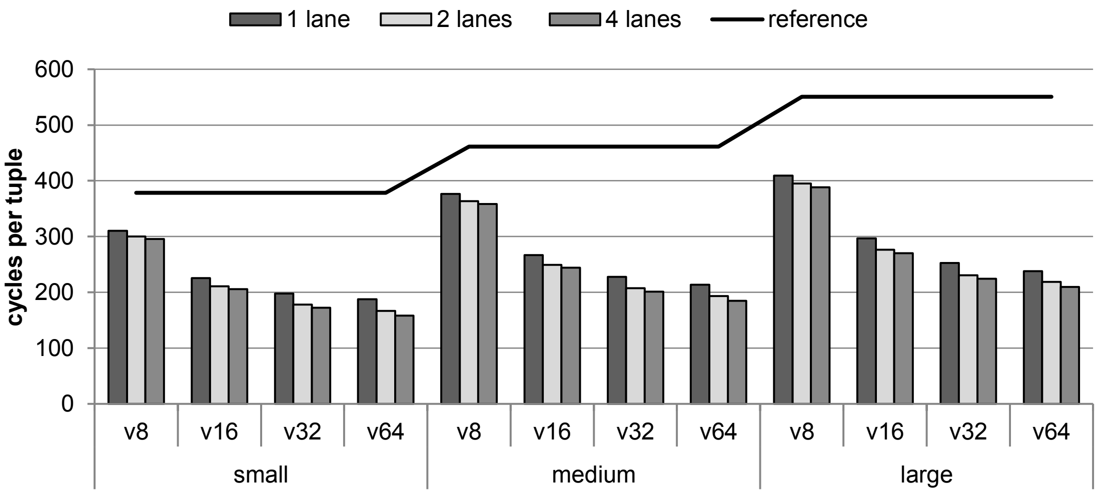


Figure 3.3: Performance results for quicksort with an OET sort cleanup.

3.3.2 Bitonic Mergesort

This subsection explores bitonic mergesort, a type of algorithm that makes use of sorting networks [Bat68]. Sorting networks differ from many other categories of sorting algorithms in that their complexity and number of operations are fixed ahead of time and do not depend on the values of the input. The algorithm is broken into two phases. Phase 1 creates relatively short sorted blocks from the input. Phase 2 iteratively merges these blocks into a single sorted block.

```
1: $mask \leftarrow$ compare($\vec{v0} > \vec{v1}$)          ▷ v0 and v1 hold the input
2: $\vec{v2} \leftarrow$ merge($\vec{v0}$, $\vec{v1}$, $mask$)          ▷ minimum values
3: $\vec{v3} \leftarrow$ merge($\vec{v1}$, $\vec{v0}$, $mask$)          ▷ maximum values
4: $\vec{v0} \leftarrow$ shuffle($\vec{v2}$, $\vec{v3}$, pattern$\alpha$)
5: $\vec{v1} \leftarrow$ shuffle($\vec{v2}$, $\vec{v3}$, pattern$\beta$)
```

Figure 3.4: Pseudocode for a single step of the vectorised bitonic network.

Phase 1 loads a block of $2 \cdot MVL$ contiguous elements into two vector registers, sorts them using a bitonic sorting network and stores the sorted output to memory. Sorting $x = 2 \cdot MVL$ elements requires $\sum_{i=1}^{log_2 x} i$ steps. The pseudocode of a single step is shown in Figure 3.4 and has five non-memory vector instructions (nine when using a payload). Each step uses two unique shuffle patterns which are pregenerated before compilation. Increasing the MVL also increases the number of steps needed, however, all of the instructions within a step can be accelerated using parallel lanes. This phase creates $\frac{n}{2 \cdot MVL}$ sorted blocks which are used in the next phase. For more information regarding vectorised bitonic sorting networks, we refer the reader to [GBY07, Sto78].

Phase 2 iteratively merges adjacent sorted blocks into larger sorted blocks until finally producing a single sorted block. At this point the blocks produced from Phase 1 are already larger than the MVL meaning that they must be merged piece by piece rather than all at once. To accomplish this we use a technique proposed in [IMKN07]. Merging two blocks of combined length m requires $\lfloor \frac{m-1}{MVL} \rfloor$ calls to a bitonic merging network. Bitonic merging networks are similar to the sorting networks described in Phase 1, however, as both inputs are already sorted they need only $log_2 x$ steps instead of $\sum_{i=1}^{log_2 x} i$ steps. This merging strategy has a linearithmic complexity of $O(n \cdot log_2 blocks)$ where $blocks$ is the number of sorted blocks created in Phase 1. For more information about iterative block merging using bitonic merging networks, we refer the reader to [CNL$^+$08, IMKN07, SKC$^+$10].

Figure 3.5 displays the results of bitonic mergesort. It can be seen that for all the single lane experiments, increasing the MVL degrades the CPT instead of improving it. This is because, as previously mentioned, increasing the MVL increases the number of steps in the sorting network and therefore adds extra work. Using multiple lanes allows this work to be done in parallel. It can be seen that four lanes is enough to overcome the penalty of increasing the MVL and using two lanes has mixed results due to the two opposing effects. The maximum speedup of bitonic mergesort over the reference benchmark is 2.9×, 3.0× and 2.9× for **small**, **medium** and **large** respectively.

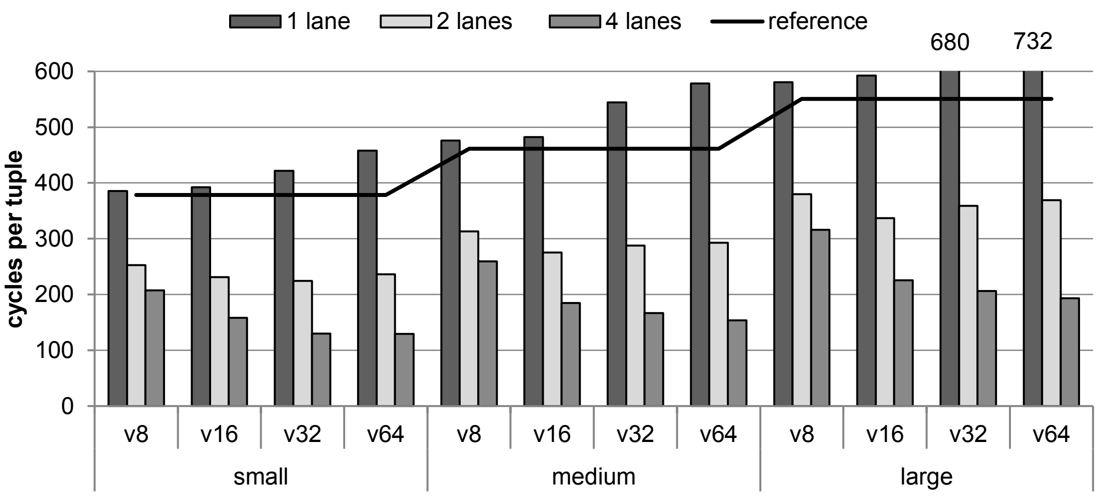


Figure 3.5: Performance results for bitonic mergesort.

Figure 3.6 shows the trend for each dataset run with a MVL of 64 and varying the number of lanes from one to 64. When 64 lanes are used, the maximum speedup over the reference benchmark is 6.4×, 6.9× and 6.4× for **small**, **medium** and **large** respectively. Increasing the number of lanes beyond 16 yields very little extra benefit. This shows that while a large number of lanes is useful in this case, increasing this number all the way to the MVL has disproportionate returns.

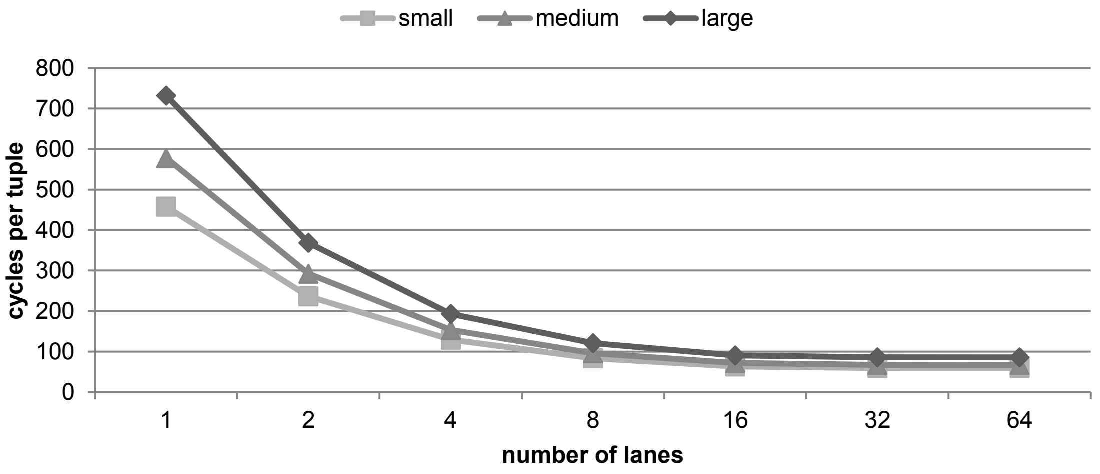


Figure 3.6: Varying the number of lanes of bitonic mergesort when $MVL = 64$.

One drawback of this approach is that it requires a general `shuffle` vector instruction. As the MVL of the vector registers increases, this instruction becomes more complex to implement with a low number of cycles. Intel's AVX2 offers an eight-element 32-bit general shuffle instruction with a 3-cycle latency. The Intel Xeon Phi has a sixteen-element 32-bit general shuffle and an average latency of 6 cycles has been measured by Fang et al. [FVS+13]. This would mean their `shuffle` instructions require

0.375 cycles per element. All non-memory vector instructions run in our simulation environment assume a latency of $\frac{VL}{lanes}$, therefore the four-lane configurations require a lower 0.25 cycles per element. This implies that our configurations with more than four lanes assume an aggressive implementation of shuffles and the results should be considered optimistic. Furthermore, the implementation of `shuffle` instructions becomes increasingly more complex as the number of lanes grows. We discuss this with more detail in Chapter 5.

3.3.3 Radix Sort

Radix sort [Knu98] is a non-comparative numerical sort with a complexity of $O(k \cdot n)$ where the value k depends on the number of passes of the algorithm. For m-bit integers, the number of passes is $\lceil \frac{m}{log_2 bins} \rceil$ where *bins*—an input parameter of the algorithm—refers to the size of the histogram used internally and $log_2 bins$ is the number of bits sorted per pass.

The work of Zagha and Blelloch [ZB91] proposed a technique to vectorise radix sort which was originally implemented on the CRAY Y-MP. Figure 3.7 provides a high-level overview of the algorithm using a step-by-step example of the three principal steps. The three steps together make up the body of the main loop that forms a single pass of the algorithm. For a more detailed explanation of the vectorised radix sort algorithm, we refer the reader to [ZB91].

Step 1—The entire input is loaded iteratively and a histogram is created for a subset of the input's bits. For example, the first pass of the algorithm uses the first $log_2 bins$ bits of each value, the second pass uses the next $log_2 bins$ bits, etc. Subfigures 3.7i and 3.7ii illustrate a single iteration of the vectorised loop implementing this step; in total there are $\frac{n}{MVL}$ iterations. **(a)** The input array is loaded into a vector register and $log_2 bins$ bits of these values are extracted. Due to the nature of the algorithm, each vector element must see a contiguous portion of the dataset, therefore, the input is loaded using strides of $\frac{n}{MVL}$ and the base address is incremented after every iteration. **(b)** The extracted bits are indices into a histogram which is incremented. Multiple elements within the same vector can index to the same bin. If a single histogram were used, updates using these indices would be incorrect. To avoid such conflicts, a local histogram is used for each vector element, i.e. there are MVL local histograms. **Step 2**—A prefix sum is performed over the collection of local histograms. **Step 3**—The entire input is loaded again and distributed to an output array with offsets determined by the prefix sum. Similar to Step 1, Subfigures 3.7iv, 3.7v and 3.7vi illustrate a single iteration of the vectorised loop. **(a)** The input is reloaded into a vector register in an identical way to Step 1a and the same bits are extracted. **(b)** The extracted bits are indices into the prefix sum which is read and incremented. Unlike Step 1b, the values before the increment are retained. **(c)** The retained values are used as a vector of offsets to scatter the input values to an output array. This output array becomes the input array in the next pass of the algorithm. The algorithm guarantees that all elements are scattered to unique locations.

We first look for the optimal number of bins for a given dataset. Figure 3.8 shows the average CPT for each of the datasets as the number of bins is increased from four

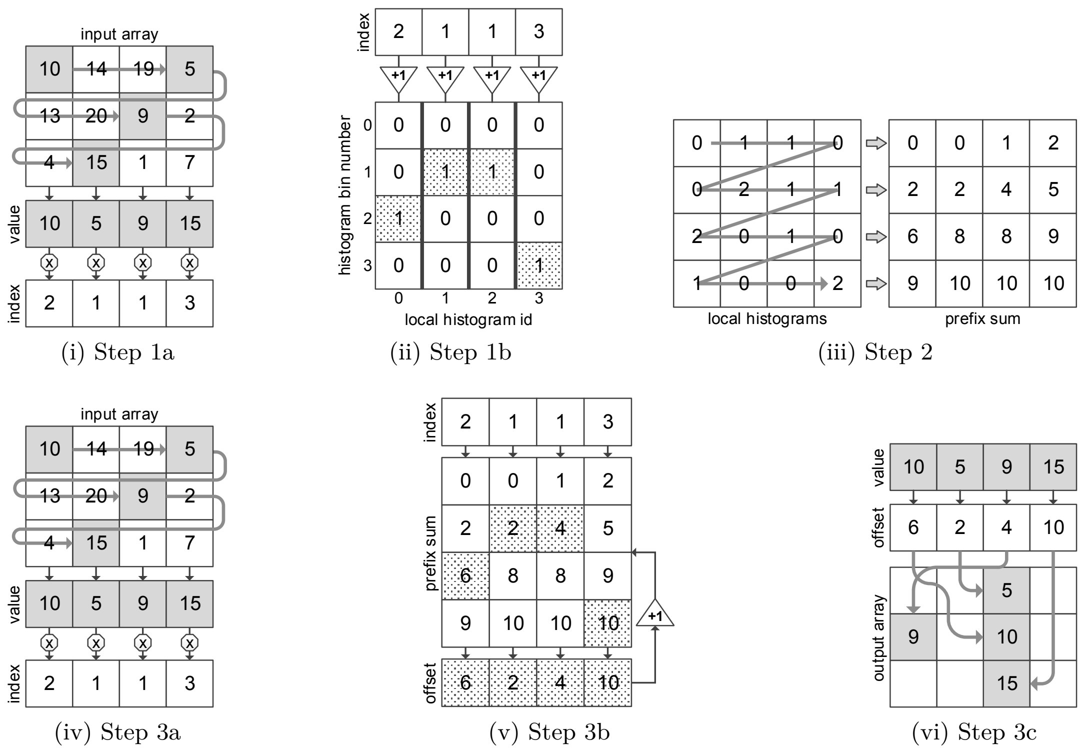


Figure 3.7: The three steps of the first pass of the vectorised radix sort where $n = 12$, $MVL = 4$ and $bins = 4$.

to 1,024. Each experiment is run with a MVL of 64 and a single lane. As the number of bins is increased the number of passes is decreased, e.g. four bins require sixteen passes and 256 bins require four passes. Increasing the number of bins also increases the memory needed and reduces the locality of the working set, this has the effect of increasing the number of cycles needed per pass. For all the datasets we find the sweet spot to be sixteen bins.

Figure 3.9 shows the results of radix sort executed with sixteen bins. The values follow a nice trend in that increasing the MVL decreases the average CPT. The lanes also help, however—similar to quicksort—adding more than four would give disproportionate returns. Although the indexed memory instructions can benefit from using lanes, the algorithm is still dominated by its strided memory access pattern which does not benefit from using them. Another interesting property is that each dataset exhibits very similar behaviour in terms of CPT; this is because radix sort has a complexity of $O(k \cdot n)$. The maximum speedup over the reference benchmark is 3.6×, 4.3× and 5.2× for **small**, **medium** and **large** respectively.

There are two significant drawbacks to this algorithm. (1) Each pass of the algorithm must be stable, i.e. equal values do not change their relative order. To ensure stability, a technique called loop raking is used which partitions the input into MVL chunks of sequential values; each element of the vector register, termed 'virtual pro-

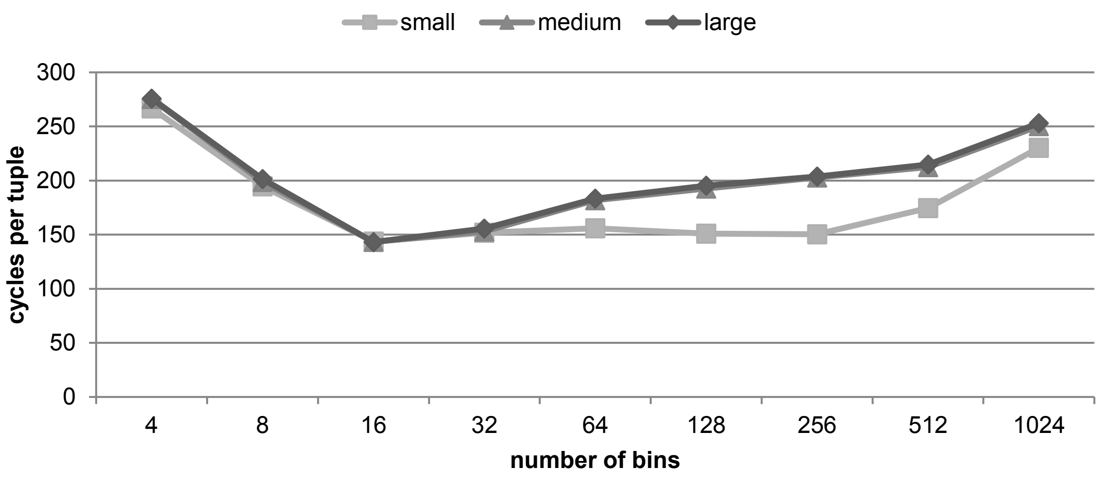


Figure 3.8: The optimal number of bins for radix sort when $MVL = 64$.

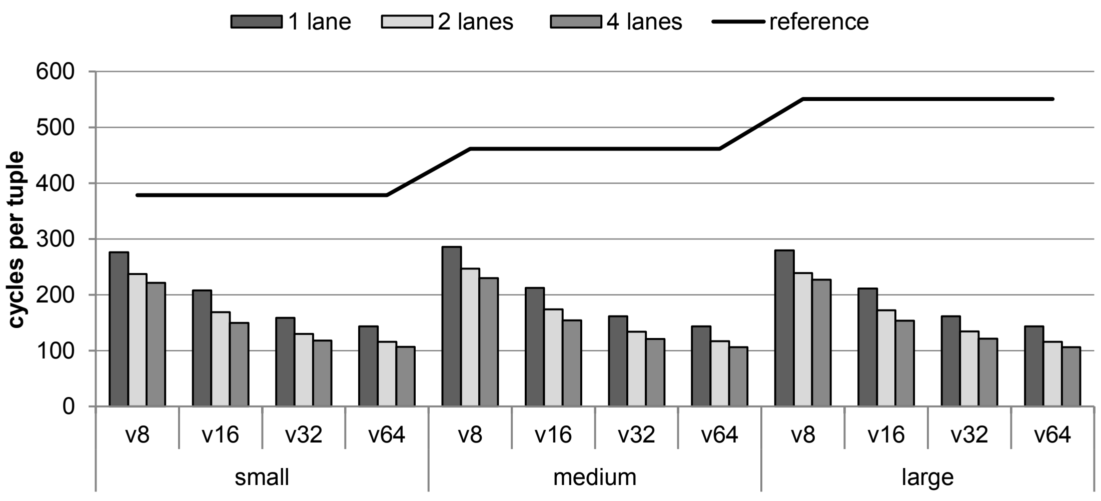


Figure 3.9: Performance results for radix sort run with sixteen bins.

cessor', operates on its own chunk. This requires using a strided memory access pattern in Step 1a and Step 3a which guarantees that any element of the vector register sees a contiguous portion of the input. As the input length n increases, so does the stride; this consequently reduces spatial locality and underutilises the available memory bandwidth. (2) In order to avoid conflicts, the algorithm's histogram must be replicated MVL times. This limits the feasible number of bins which the algorithm can use without encountering cache locality problems and—in turn—directly influences the number of passes the algorithm must make. This replication also increases the work necessary to calculate the prefix sum in Step 2.

3.3.4 Summary

We have evaluated three distinct sorting algorithms on a uniform platform using consistent metrics. Although all these algorithms outperformed the scalar reference, each one suffered from unique weaknesses and bottlenecks.

Quicksort was shown to perform very well when working with partitions much larger than the MVL, however, as the size of the partition decreased the performance of quicksort degraded. We were able to rectify this somewhat by adopting a variant of the algorithm that uses OET sort as a cleanup mechanism. Increasing the parametrisable threshold reduced the amount of time that quicksort spent processing short partitions. However, due to the fact that the OET sort has an $O(n^2)$ complexity, we observed that we could not increase the threshold indefinitely without degrading the overall performance.

Bitonic mergesort was shown to perform better, but only when the number of parallel lanes is high. Four parallel lanes were needed to outperform quicksort and even more were needed to outperform radix sort in most cases. When a single-lane configuration was used, the algorithm performance degraded as the MVL increased. This behaviour is in stark contrast to that of quicksort whereby the algorithm performed better as the MVL grows but exhibited only marginal speedups when adding parallel lanes. Additionally, bitonic mergesort requires a general shuffle instruction which is complicated to implement with a low latency when the MVL is large.

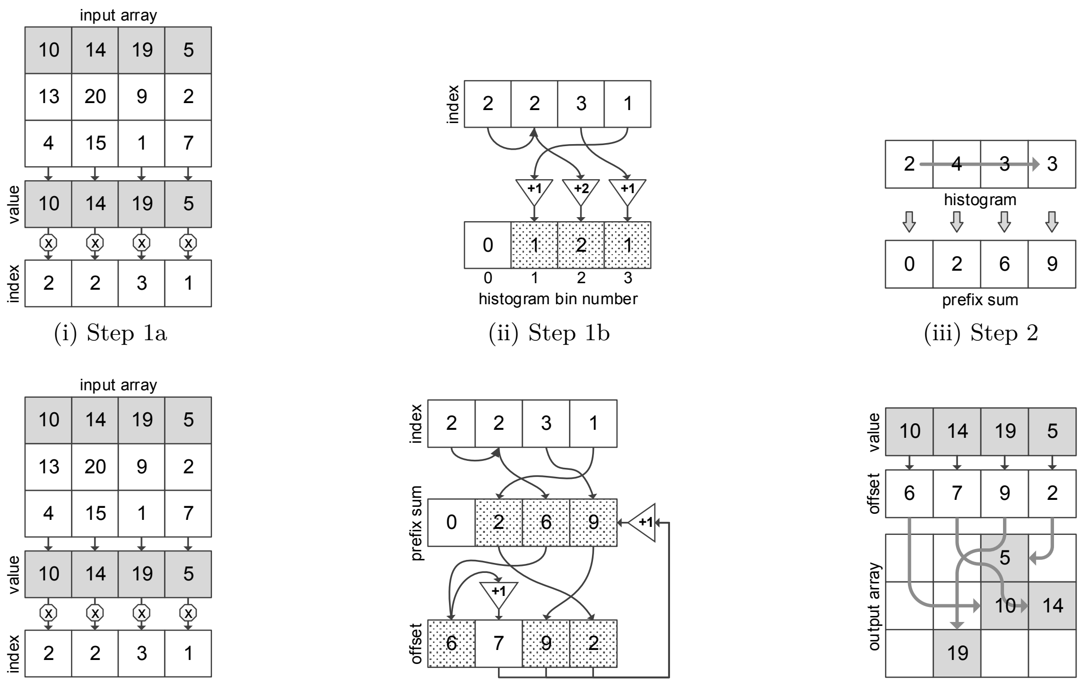
Both quicksort and bitonic mergesort suffer from a complexity of $O(n \cdot log_2 n)$ whereas radix sort has a highly desirable complexity of $O(k \cdot n)$. In contrast to bitonic mergesort, radix sort works well on SIMD hardware with or without parallel lanes. It also does not get bottlenecked by a divide and conquer strategy causing low effective vector lengths as is the case with quicksort. Unfortunately, radix sort suffers from using a stride in its dominant memory access pattern as well as having to replicate its internal bookkeeping structures by the number of elements in a vector register. This is in contrast to other algorithms which primarily use unit-stride access patterns and have very modest memory requirements. We use these weaknesses to motivate the next section of this chapter.

3.4 VSR Sort

This section presents Vectorised Serial Radix (VSR) sort, our novel vectorised non-comparative sorting algorithm based on radix sort. We first explain and discuss the algorithm. After, we introduce two new vector instructions in order to facilitate the algorithm's vectorisation; we also propose a corresponding hardware structure to implement these instructions. Finally, we evaluate the algorithm experimentally and compare the results and behaviour with those of the previous algorithms.

3.4.1 The Algorithm

We developed VSR sort with the aim to be efficiently vectorisable without the drawbacks identified in Section 3.3. VSR sort is inspired by radix sort and follows more



(i) Step 1a (ii) Step 1b (iii) Step 2

(iv) Step 3a (v) Step 3b (vi) Step 3c

Figure 3.10: Step by step example of the first pass of the new VSR sort algorithm where $n = 12$, $MVL = 4$ and $bins = 4$.

closely a serial implementation of the algorithm. Instead of accessing the input with a large strided access pattern, a much more efficient unit-stride access is used. Additionally, the bookkeeping structure is not replicated MVL times making the algorithm a lot more cache friendly. Figure 3.10 provides a high-level overview of one pass of the algorithm. Each step is analogous to its counterpart in Section 3.3.3.

Step 1—The input is loaded iteratively and a histogram is created for a subset of the input's bits. **(a)** The input is loaded into a vector register and log_2bins bits of these values are extracted. Unlike the vectorised radix sort, a unit-stride access pattern is now used. **(b)** The extracted bits are indices into a histogram which is incremented. Instead of MVL local histograms, only a single instance is used. Conflicting indices are detected dynamically and the increment value is modified accordingly. This conflict detection/correction is discussed in the next subsection. **Step 2**—A prefix sum is performed over the histogram. Fewer operations are needed than the previous implementation due to the single histogram. **Step 3**—The entire input is loaded again and distributed to an output array with offsets determined by the prefix sum. **(a)** The input is reloaded into a vector register in an identical way to Step 1a and the same bits are extracted. **(b)** The extracted bits are indices into the prefix sum which is read and incremented. The prefix sum is updated, however, unlike the previous implementation the increment may be more than 1 due to conflicting indices. The values loaded from the prefix sum are corrected according to the conflicts; these corrected values become the offsets used in the next substep. Subfigure 3.10v illustrates such a conflict—the index 2 is used twice within the same iteration. The correction ensures that the first and second offsets generated are distinct and that the prefix sum is incremented twice, i.e. the number of times this conflicting index is seen. **(c)** The offsets are used to scatter the input values to an output array.

3.4.2 New Instructions

Steps 1a, 2, 3a and 3c can be implemented in a vectorised way using conventional vector SIMD instructions. Steps 1b and 3b are different because there is a lack of suitable instructions in existing vector SIMD ISAs. Although each element is processed in the same way, there are loop-carried dependencies which make the operations difficult to vectorise. To solve this, we introduce two new instructions—`vector prior instances` (`VPI`) and `vector last unique` (`VLU`).

3.4.2-a Semantics and Usage

`VPI` uses a single vector register as input, processes it serially and outputs another vector register as a result. Each element of the output asserts exactly how many instances of a value in the corresponding element of the input register have been seen before. An example is shown in Figure 3.11 (elements are processed from left to right).

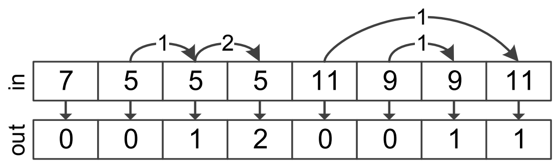


Figure 3.11: Example of `Vector Prior Instances` (`VPI`).

`VLU` also uses a single vector register as input but produces a vector mask as a result. The idea is to mark the last instance of any particular value found. An example is shown in Figure 3.12 (elements are processed from left to right). A bit in the output mask register is set only if the corresponding value in the input vector is not seen afterwards; these cases are shaded in the input vector. A bitmask is useful to generate because it can be combined with the results of `VPI` and a `scatter` instruction to increment VSR sort's bookkeeping structures without conflicts.

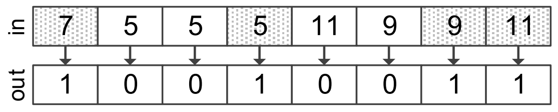


Figure 3.12: Example of `Vector Last Unique` (`VLU`).

To show the usage of `VPI` and `VLU`, pseudocode for Step 3b of VSR sort is listed in Figure 3.13. In this version we make use of vector `compress` instructions followed by a `scatter`, however, it is possible to achieve the same behaviour without the `compress` instructions by using a masked `scatter` instead.

```
1: $\vec{v1} \leftarrow$ vpi($\vec{v0}$)                                  ▷ $\vec{v0}$ is $index$ in the diagram
2: $mask \leftarrow$ vlu($\vec{v0}$)
3: $\vec{v2} \leftarrow$ gather(**base**=$prefix\_sum$, **idx**=$\vec{v0}$)
4: $\vec{v3} \leftarrow$ vvadd($\vec{v1}$, $\vec{v2}$)                    ▷ $\vec{v3}$ is $offset$ used in Step 3c
5: $\vec{v4} \leftarrow$ compress($\vec{v0}$, $mask$)
6: $\vec{v5} \leftarrow$ compress($\vec{v3}$, $mask$)
7: $vlen \leftarrow$ popcount($mask$)
8: $\vec{v6} \leftarrow$ vsadd($\vec{v5}$, 1)
9: scatter(**base**=$prefix\_sum$, **idx**=$\vec{v4}$ **vals**=$\vec{v6}$)
```

Figure 3.13: Pseudocode for Step 3b of VSR sort.

3.4.2-b Applicability Beyond Sorting

Although these instructions are proposed here to enable VSR sort, there are many opportunities to accelerate other classes of algorithms, especially where irregular DLP is an obstacle. We present a sample of uses to give an idea of the broader applicability of `VPI` and `VLU`.

Histogram generation is used in a wide variety of applications from image processing to range partitioning; `VPI`/`VLU` can be used to circumvent collisions and increment conflicting bins in a single update. Parallel queue insertion [GGK$^+$83] is used in scenarios such as game engines; `VPI`/`VLU` can be useful to avoid collisions when multiple elements are inserted in the same queue and instead place the elements in consecutive locations—similar to Step 3 of VSR sort. Maximal matching [SBF$^+$12] is an algorithm for undirected graphs, often used as scheduling scheme for multi-hop wireless networks. By laying out edges as an array-of-structures, `VLU` can be used to detect repeated vertices within a vector of edges and prune them accordingly.

The work of Lee et al. [LKC$^+$10] discusses the limitations of current SIMD instructions with irregular DLP and concludes that new instructions are necessary to open up further possibilities. There are several proposals that try to tackle this however none are suitable for VSR sort. `VPI`/`VLU` provide determinism in their behaviour which opens up more possibilities than detecting and correcting conflicts in a non-deterministic way [BFGS12]. VSR sort requires this determinism due to the strict stability requirement of the algorithm. This will be discussed further in Section 3.5.

In Chapter 4, we will look at `VPI`/`VLU` in the context of data aggregation. Interestingly, while these instructions cannot be directly used for aggregation, we propose repurposing the instructions' hardware implementation—discussed in the next subsection—with minimal modifications to better support this type of workload.

3.4.2-c Hardware Implementation

There are many possible ways to implement both `VPI` and `VLU`; the methods proposed here take into account the circumstance in which these instructions arise. A naïve implementation could calculate the results of each instruction in $O(MVL^2)$ time, however, this would not be very useful in this context for several reasons. (1) These instructions occur on the critical path of tight loops executed many times and such suboptimal implementations would negatively impact performance due to very long pipeline stalls. (2) The goal is to propose a vectorised sorting algorithm that scales well with the MVL; an implementation of $O(MVL^2)$ complexity becomes less practical each time the MVL is enlarged. (3) In VSR sort, both instructions do similar work, use the same input and are always scheduled back to back; calculating both results separately would be doing redundant work. The implementation we propose can calculate VPI and VLU together in $2 \cdot MVL$ cycles. After, we propose a variant that can reduce the execution time to approximately $\frac{2 \cdot MVL}{lanes}$ cycles.

To achieve this, a content-addressable memory (CAM) with MVL entries is used. Each CAM entry contains a key entry and a valid bit used for the lookup, and two possible return values—count and last idx. count is the running total of the number of times a particular value of the input vector register has been seen. last idx is an index of the input vector register and refers to the last position where a particular value has been observed. Calculating each element of the output vector register requires two cycles—one for reading the CAM and another for updating it.

Figure 3.14 shows an example of the process to calculate VPI. The diagram shows the state of the hardware before the instruction has completed. Six of the eight elements have already been calculated (shown with a dotted pattern) and the seventh element is just about to be calculated. Arrows with a solid line represent activity done on the first cycle and arrows with a dashed line represent activity done on the second. The value 9 located at index 6 of the input is used to access the CAM. The value located in the count field is copied into the seventh element of the output vector register. This value is 1 because there has been exactly one element of input seen up until this point in time which contains the value 9. On the second cycle, count is incremented and the corresponding last idx value will be updated with 6 as this refers to the most recent index of the input vector register where the value 9 was observed.

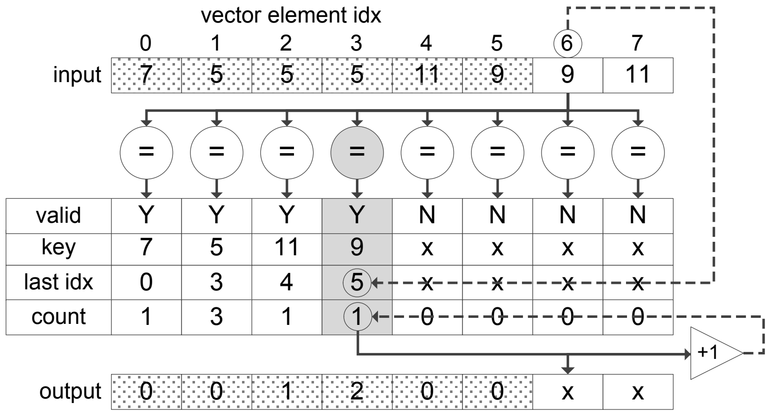


Figure 3.14: Proposed hardware calculating VPI.

last idx is not used to calculate VPI, however, it is relatively simple to update this field when updating count, this way when VLU is executed after VPI using the same input, all that remains to be done is to convert the array of last idx values to a bitmask. This can be done in relatively few cycles. Although the CAM requires MVL

entries to cover a worse-case scenario, the size of each field can be relatively small. valid is a single bit and both count and last idx require $log_2 MVL$ bits. To keep the implementation general purpose, we make key 32 bits.

One obvious obstacle extending this implementation to multiple lanes is that the semantics of the instructions `VPI` and `VLU` are defined serially. In a similar vein to ILP in out-of-order superscalar microprocessors, we propose finding the DLP dynamically during runtime. Adjacent elements of the input vector register are arranged into groups; the elements within a group can be processed in parallel provided they do not conflict with one another, otherwise they are processed serially. Detecting conflicts requires $\frac{l!}{2\cdot(l-2)!}$ comparators where l is the number of parallel lanes targeted, i.e. the group size. To keep the number of comparators low, a small l is expected to be chosen. We distinguish the conventional parallel lanes used in a vector unit from the group size of `VPI`/`VLU` which we will refer to as CAM lanes.

Figure 3.15 illustrates the parallel optimisation using two CAM lanes. There is an input vector register which is processed from left to right, and underneath there are two timelines that represent the relative execution time of both the parallel and serial implementations. Each block of the timelines represents two execution cycles, however, the parallel timeline shows stacked blocks meaning these are processed in tandem. The first, third and fourth groups of elements can be processed in parallel as there are no conflicts. The second group of elements has a conflict and is serialised. The hatched box represents the time saved over the serial implementation. In this case, it is $\frac{3}{8}$ blocks or a 1.6× speedup.

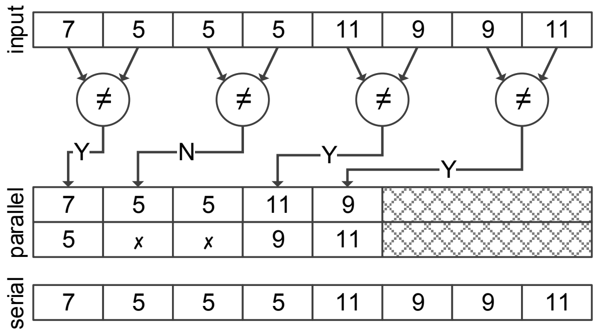


Figure 3.15: Parallel optimisation for `VPI`/`VLU` hardware with two CAM lanes.

The benefit of a parallel implementation depends highly on the input. For VSR sort, it is the number of histogram bins that will determine the probability of a conflict within a group. Increasing the number of bins decreases the number of collisions and thus allows for more parallelism. This is a tradeoff because increasing the number of bins also increases the cache footprint of the algorithm. The probability of not having a conflict for b bins and l CAM lanes is $\frac{b!}{b^l\cdot(b-l)!}$ when $l \leq b$. Table 3.3 shows the probability of group collisions calculated from the formula with various combinations of bins and CAM lanes. As l increases, the probability of a collision approaches 1.0 and all blocks are serialised with a net result performance equal to the implementation with a single CAM lane.

Table 3.3: The probabilities of structural collisions in the VPI/VLU hardware for a uniform distribution when varying the number of histogram bins and CAM lanes.

bins	2 CAM lanes	4 CAM lanes	8 CAM lanes	16 CAM lanes
4	0.25	0.91	1.00	1.00
16	0.06	0.33	0.88	1.00
64	0.02	0.09	0.37	0.87
256	0.00	0.02	0.10	0.38
1,024	0.00	0.01	0.03	0.11

We propose choosing two or four CAM lanes as increasing the parallelism further would give diminishing returns due to a high amount of serialisation. Increasing the CAM lanes further would also require (1)—increasing the complexity of the CAM with more ports and (2)—increasing the number of comparators used to detect collisions. A configuration with four CAM lanes requires a modest six comparators whereas a configuration with eight CAM lanes would require 28 comparators. Detecting conflicts in this way resembles the logic needed to detect hazards in superscalar register renaming. In subsequent experiments, we set the number of CAM lanes equal to the number of conventional vector lanes and refer to both together as lockstepped lanes, or simply as lanes.

In order to verify our timing assumptions, we model this structure using CACTI-P [LCA+11]. We choose a 32 nm technology and a V_{dd} of 0.75 V which matches our baseline Westmere microarchitecture. Assuming a CAM of 64 entries and between one and four ports, our experiments show that searching and accessing this structure together requires between 0.276 ns and 0.339 ns. These values are lower than the 0.375 ns cycle time of our simulated processor which indicates it is reasonable to assume updating the structure can be done in two cycles per element. In Section 3.4.3 we make further experiments assuming different latencies to measure the impact of alternative hardware implementations.

3.4.3 Results

To directly compare VSR sort with the previous vectorised radix sort, we first run it with sixteen bins as this was the optimal value found in Section 3.3.3. Figure 3.16 displays the results using single-lane configurations. On average, VSR sort performs 1.8× faster than the previous algorithm; in this case, the constant factor of $O(k \cdot n)$ has been improved.

Because we have eliminated the histogram replication of the previous vectorised radix sort, it should be possible to increase the number of bins without compromising performance and thus reduce the number of passes. Figure 3.17 shows VSR sort run with a MVL of 64 and a single lane while varying the number of bins. In comparison to the equivalent experiment in Section 3.3.3, the optimal number of bins for VSR sort is 256 (four passes) instead of sixteen (eight passes).

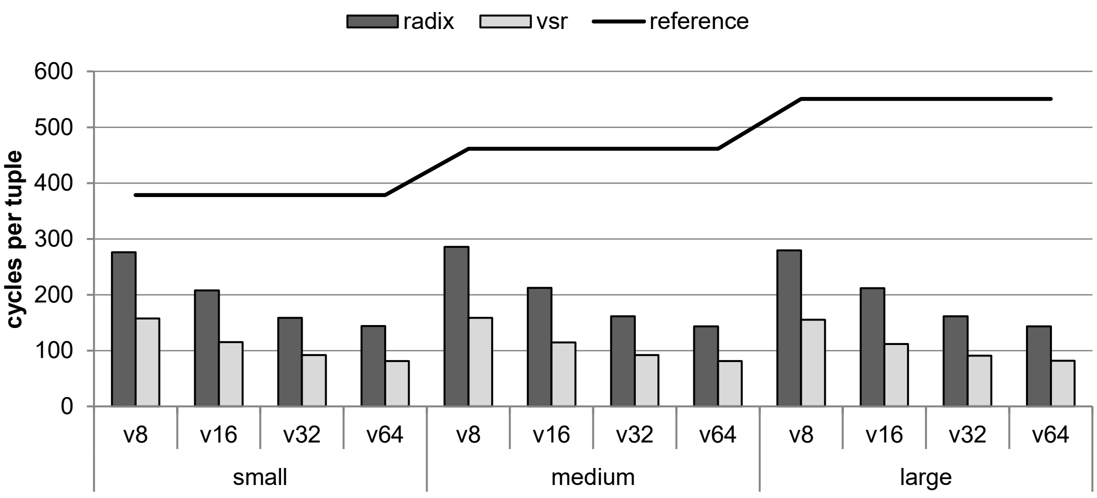


Figure 3.16: Comparison of radix sort and VSR sort using sixteen histogram bins.

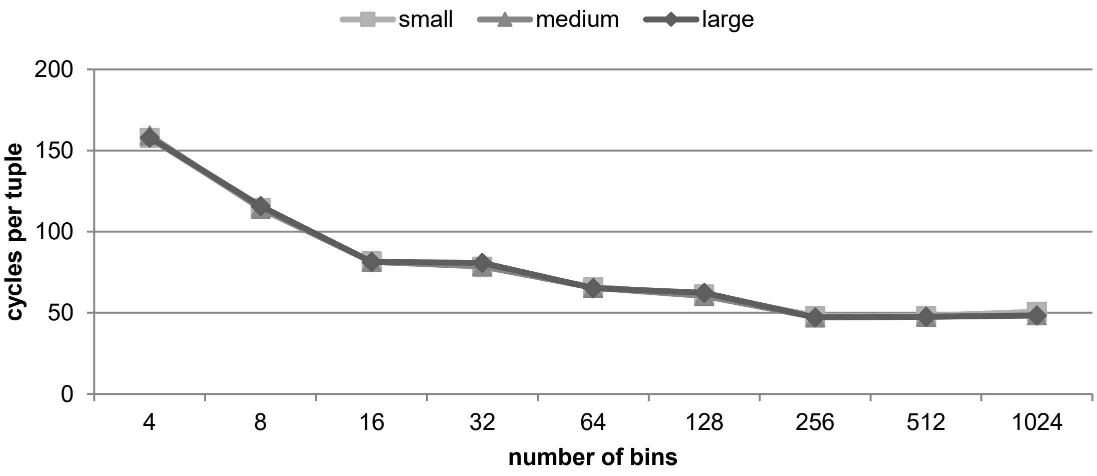


Figure 3.17: The optimal number of bins for VSR sort when $MVL = 64$.

Figure 3.18 shows the results of VSR sort run with 256 bins. On average it performs 1.8× faster than VSR sort run with sixteen bins; in this case the k from $O(k{\cdot}n)$ has been improved. With a single-lane configuration, the maximum speedup over the reference is 7.9×, 9.8× and 11.7× for **small**, **medium** and **large** respectively. In our experiments, this is the highest speedup seen so far *including* other algorithms run with parallel lanes. When this algorithm is run with parallel lanes the maximum speedups over the reference benchmark increase to 14.9×, 17.3× and 20.6×. On average, VSR sort performs 3.4× better than the next-best vectorised sorting algorithm when run on the same hardware configuration.

There are some noteworthy observations about VSR sort. Firstly, while the algorithm benefits from longer MVLs, very good results can still be achieved with a short MVL. Using a MVL of eight can achieve a CPT below 100 for all datasets whereas the algorithms evaluated in Section 3.3 could not achieve this kind of performance—even

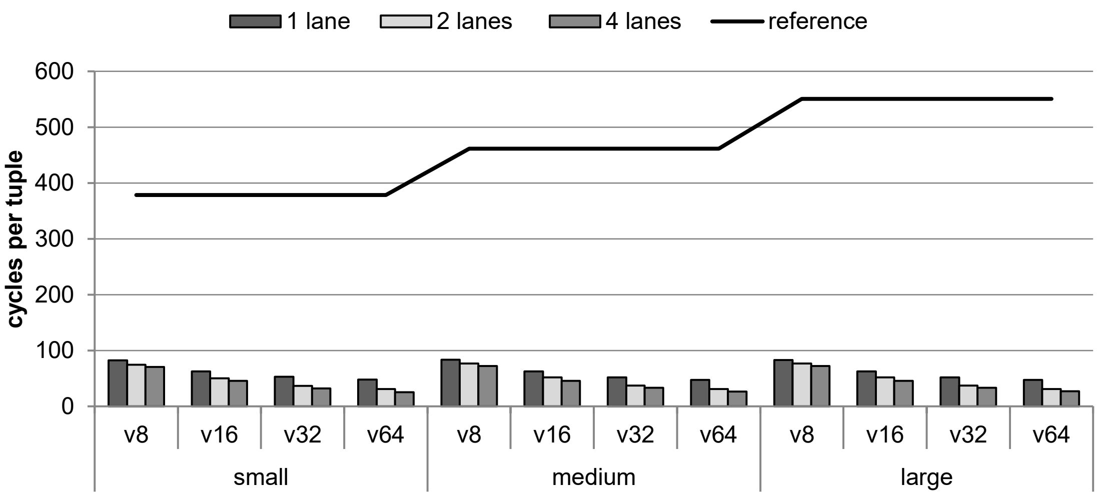


Figure 3.18: Performance results for VSR sort run with 256 histogram bins.

with a MVL as large as 64. Only bitonic mergesort run with at least sixteen parallel lanes could achieve an average CPT lower than 100 for all the datasets. Secondly, while lanes give a clear advantage, the performance of VSR sort when run with a single lane is commendable. VSR sort can achieve an average CPT below 50 on the single-lane configurations; even bitonic mergesort run with a large number of lanes could not achieve a CPT below 60. Thirdly, it can be seen that all the datasets achieve similar CPTs to each other. This implies that the algorithm's complexity of $O(k \cdot n)$ is being adhered to. The original vectorised radix sort also has a complexity of $O(k \cdot n)$, however, our VSR sort improves k as well as the runtime constant.

We conduct additional experiments with alternative implementations of `VPI` and `VLU`. Figure 3.19 shows the results of VSR sort executed with the **large** dataset run with six alternative hardware configurations. cam1, cam2, cam4 and cam8 fix the CAM's latency to one, two, four and eight cycles per element respectively—cam2 being the default configuration used in the previous experiments. naïve-a removes the CAM structure and calculates `VPI` and `VLU` separately using a simple hardware implementation with quadratic complexity. Essentially, both `VPI` and `VLU` are microcoded to perform a type of all-to-all comparison using the existing arithmetic/logical functional units. Each instruction requires $\sum_{i=1}^{VL-1} \lceil \frac{i}{lanes} \rceil$ cycles where VL is the vector length of the operation. naïve-b has a similar implementation to naïve-a, however, both `VPI` and `VLU` are calculated simultaneously and the latency is paid just once.

In all cases, increasing the latency of the CAM structure decreases the overall performance of VSR sort. What is notable is that the impact on performance is far more pronounced when only one lane is used; as the number of lanes is increased, the relative penalty of a higher latency CAM decreases. For example, using a MVL of 64 and one lane, the increase in CPT when changing from cam2 to cam4 is 27%. In contrast, when four lanes are used the increase in CPT is only 3%. Likewise, when moving from cam2 to cam8, the single-lane configuration has a 92% increase in CPT whereas the four-lane configuration's increase in CPT is a milder 21%. Using the more aggres-

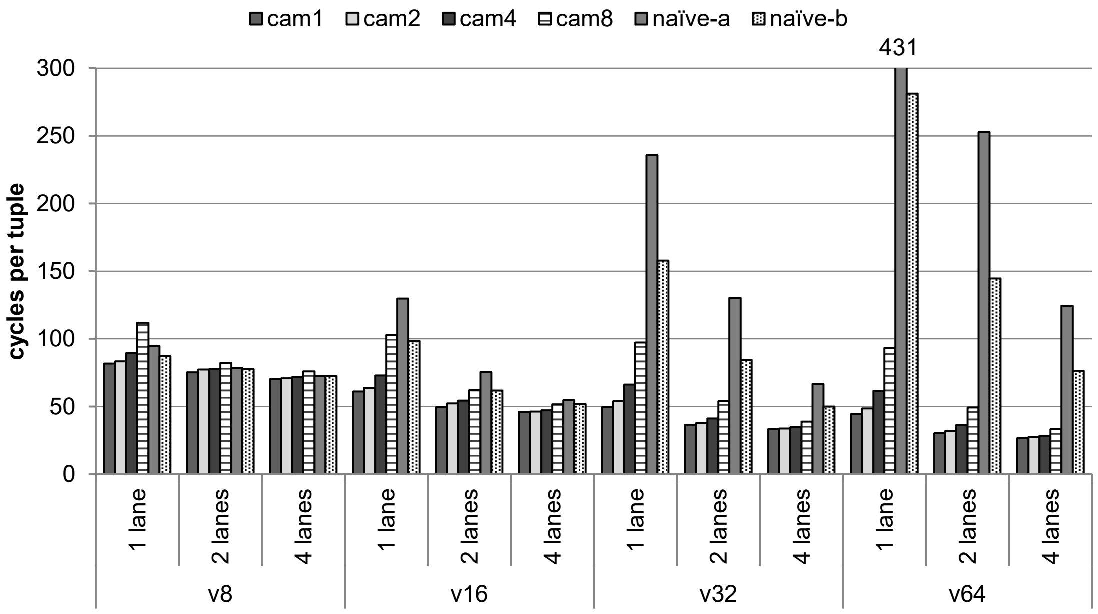


Figure 3.19: Results for VSR sort when processing the large dataset run with various hardware implementations of VPI and VLU.

ive cam1 configurations yields speedups over our default cam2 configuration between 1.01× and 1.09×. In contrast, using the less aggressive cam4 configurations over cam2 exhibits CPT increases between 1% and 27%. Our default latency of two cycles per element appears to be a good compromise between cam1 and cam4, however, the results also suggest that it is possible to use less aggressive CAM-based implementations of VPI/VLU and still have good performance.

Both naïve-a and naïve-b have different trends to the CAM-based implementations. Although using multiple lanes tempers the impact of higher latencies, we observe that increasing the MVL decreases the overall performance of the algorithm which is in contrast to the CAM-based experiments. It can be seen that for a MVL of eight, naïve-a performs comparably with the CAM-based implementations and even outperforms cam8. Once the MVL is increased, however, the burden of the quadratic complexity takes effect. Even when multiple lanes are leveraged, they are not numerous enough to always counteract the effects of the $O(MVL^2)$ implementation. naïve-b can be seen as an optimisation of naïve-a. When multiple lanes are used, naïve-b can sometimes reach comparable performance to cam8, however, in the case of $MVL = 64$—even when using four lanes—naïve-b has a 130% increase in CPT over cam8. We conclude that a CAM-based approach has clear performance advantages over either of the naïve approaches.

3.5 Related Work

There have been numerous works related to sorting on SIMD architectures and the most relevant articles have already been cited and discussed in previous sections. There have also been several works that focus on the problem of using SIMD-like instructions to update data structures where there may be memory conflicts. We summarise these works and explain why they are different to our own instructions and also why none of them are appropriate for VSR sort.

The work of Ahn et al. [AED05] proposes `scatterAdd`, an instruction for stream architectures that provides support for parallel histogram generation. This instruction takes the following form: `scatterAdd(`$base,\ \overrightarrow{bins},\ c$`)` where *base* is a location in memory of a histogram, $\overrightarrow{bins}$ is an array of values that correspond to bins of the histogram and c is the increment (typically 1). `scatterAdd` is advantageous in the sense that updating a histogram can be done with a single stream instruction whereas our approach requires a combination of six vector instructions to do the same. Our proposed instructions `VPI` and `VLU` can be used to emulate the behaviour of `scatterAdd`, however, the semantics are much stricter meaning they can also be used for VSR sort whereas `scatterAdd` cannot. `scatterAdd` is a FetchAndOp-style instruction but lacks a return path for the original values; furthermore, the instruction lacks deterministic ordering semantics. VSR sort requires a return path as well as strict serial ordering therefore `scatterAdd` is not suitable.

The work of Kumar et al. [KKS+08] proposes atomic SIMD memory instructions. First, gather/scatter functionality is added to a SSE-like ISA, then this functionality is extended with load-linked/store-conditional semantics. The instructions use a best-effort implementation meaning that stores to conflicting addresses will fail and this is explicitly visible to the user in the form of a bitmask result. When working with a data structure like a histogram, a SIMD gather-modify-scatter operation must be contained in a data-dependent loop and repeated until every element has been successfully updated. In the worst case this will have the penalty of complete serialisation including the overhead of the loop. Unlike `scatterAdd`, this technique has a return path meaning that it is possible to retrieve a copy of the original value before it is updated. Unlike `VPI` and `VLU`, there is no intra-register ordering guarantees which implies that this will not work for VSR sort in which the ordering is of utmost importance. The atomic SIMD memory operations also work between threads meaning that multiple cores with SIMD support can attempt to update the same data structure. `VPI` and `VLU` do not need such complexity because all input and output is contained within registers rather than shared memory.

Intel has recently developed conflict detection instructions as extensions to AVX-512 known as AVX-512CD [Int14b]. The `VPCONFLICT` instruction uses a single multimedia register as input and returns a bitmask showing conflicting pairs. By using this bitmask it is possible to remedy gather-modify-scatter operations with conflicting indices and this could be used to aid histogram generation. Any indices that conflict can be masked out and—similar to [KKS+08]—the programmer must use a loop construct to iteratively process the same input until there are no pending elements. While we also generate histograms in VSR sort, this occurs in very tight loops on the critical path with a high number of iterations; a best-effort implementation could severely impact performance. We have proposed suitable instructions to aid this as well as implementations that are efficient and scalable. We avoid data-dependent loops to handle conflicts and we prevent unnecessary repeated accesses to the same position of the histogram structure.

3.6 Conclusions

In this chapter we have performed extensive analysis on three diverse sorting algorithms using a uniform and modern platform and assessed them using consistent metrics. We have learned that all of the algorithms suffer from bottlenecks and scalability problems due to the irregularity of the DLP and the limitations of a standard vector SIMD instruction set.

Based on these findings, we have proposed VSR sort—a novel way to efficiently vectorise radix sort. To enable this algorithm in a SIMD architecture we have defined two new instructions—`vector prior instances` and `vector last unique`. We have provided a suitable hardware proposal that includes both serial and parallel variants. We have demonstrated that the algorithm scales well when increasing the maximum vector length, and works well both with and without parallel lockstepped lanes. VSR sort has shown maximum speedups over a scalar baseline between 7.9× and 11.7× when a simple single-lane pipelined vector approach is used and maximum speedups between 14.9× and 20.6× when as few as four parallel lanes are used.

We have compared VSR sort with three very different vectorised sorting algorithms—quicksort, bitonic mergesort and a previously proposed implementation of radix sort. VSR sort outperforms all of the aforementioned algorithms and achieves a comparatively low CPT without strictly requiring parallel lanes. It has a complexity of $O(k \cdot n)$ meaning that this CPT will remain constant as the input size increases—a highly-desirable property of a sorting algorithm. The k factor is significantly improved over the original vectorised radix sort as well as the constant performance factor.

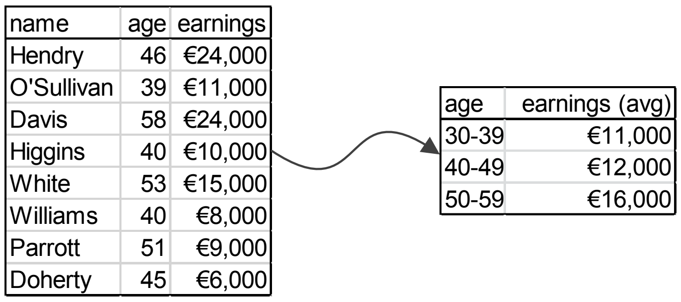
VSR Sort's dominant memory access pattern is unit-stride which helps maximise the utilisation of the available memory bandwidth. Unlike the previous vectorised radix sort, VSR sort does not replicate its internal bookkeeping structures which consequently allows them to be larger and reduces the number of necessary passes of the algorithm. On average VSR sort performs 3.4× better than the next-best vectorised sorting algorithm when run on the same hardware configuration.

CHAPTER 4

A Study on Aggregation

4.1 Introduction

Aggregation is a very useful operation when summarising considerable amounts of data and is a cornerstone of important technologies such as SQL DBMSs, OLAP cubes, MapReduce, pivot tables and statistical languages. In the TPC-H decision support benchmark, aggregations can dominate eight of the twenty-two queries [BNE14]. An example of a simple aggregation is shown in Figure 4.1; earnings per persons are grouped together and averaged by age. A summary like this may help the user uncover trends not immediately apparent from the raw data, e.g. if there is a correlation between earnings and age. Since the rate of data generation is growing exponentially each year [MCB+11, CML14], this has led to enormous volumes of data to aggregate and summarise. As such, there is pressure on both software and hardware developers to create solutions that can cope with these increasing requirements. In this chapter, we explore different vectorisation techniques applied to data aggregation.



name	age	earnings
Hendry	46	€24,000
O'Sullivan	39	€11,000
Davis	58	€24,000
Higgins	40	€10,000
White	53	€15,000
Williams	40	€8,000
Parrott	51	€9,000
Doherty	45	€6,000

age	earnings (avg)
30-39	€11,000
40-49	€12,000
50-59	€16,000

Figure 4.1: Example of an aggregation operation. The input table on the left is summarised on the right. Earnings are grouped by age range and averaged.

In this chapter, we make three principal contributions. (1) We propose and implement several vectorised algorithms for data aggregation using common vector SIMD instructions and evaluate them using the infrastructure created in Chapters 2 and 3. (2) In order to determine the sensitivity of the input to performance, we assess the behaviour of these algorithms for a range of data distributions and cardinalities. (3) Leveraging our proposal for irregular DLP from Chapter 3, we extend this hardware with minimal additions to create new instructions useful for data aggregations.

There are several notable outcomes of this work. Firstly, we find that the performance of vectorised data aggregation is immensely dependent on the distribution and cardinality of the input. As a consequence, there is not a single vectorised algorithm that provides the best performance in every case. Secondly, we discover that vectorising the algorithms is not trivial due the irregularity of the DLP. We propose two distinctly different types of solutions—the first, *evasion*, attempts to avoid this irregularity through transformation whereas the second, *confrontation*, tackles it head on. The evasion techniques—relying on typical vector SIMD instructions—yield speedups only in a subset of the cardinalities/distributions with significant slowdowns over the scalar baseline in other cases. On the other hand, the confrontation techniques—embracing the irregularity with new SIMD instructions—achieve speedup for all cardinalities/distributions, even when in some cases results are surpassed by an evasion technique. Finally, since cardinality can be determined at runtime, we introduce an adaptive near-optimal implementation that selects the most appropriate algorithm. Our proposed vector implementations exhibit speedups between 2.7× and 7.6× over a scalar baseline for a maximum vector length of 64 and four lockstepped lanes.

The outline of this chapter is as follows. Section 4.2 discusses the changes we make to our architecture and simulation framework. Section 4.3 outlines our experimental setup, the scalar baseline and discusses the obstacles related to vectorising data aggregations. We propose and evaluate vectorised evasion techniques in Section 4.4 and vectorised confrontation techniques in Section 4.5. Related work is discussed in Section 4.6 and Section 4.7 concludes the chapter.

4.2 Changes to the Architecture

In Chapter 2, we defined a vector ISA suitable for DBMS acceleration. In Chapter 3, we extended this vector ISA with new instructions useful for sorting, including our own novel vector instructions—`VPI` and `VLU`. In this chapter, we build upon this infrastructure and extend it further in order make detailed evaluations of different vectorised aggregation algorithms. We now describe the various changes that we make to our vector architecture and simulation framework.

In Chapter 2, we outlined a mechanism whereby the vector memory unit bypasses the L1D cache and instead directly accesses the L2 cache. We found that in practise this works well with the evaluated algorithms and datasets. In Chapter 3, we mentioned briefly that some of the sorting algorithms exhibit pathological behaviour when a particular stride is used, however, we opted not to fix this issue and instead avoid it by using non-problematic datasets. In this chapter, we experiment with a large variety of data distributions which can inadvertently trigger troublesome memory access patterns. In essence, the problem occurs when individual strided or indexed memory instructions load from—or store to—the same cache set causing premature evictions and consequently poor performance. To fix this, we interleave the L2 cache sets using a simple mapping scheme based on irreducible polynomials suggested in the works of Rau [Rau91] and González et al. [GVTP97]. We find that this scheme completely eliminates all of the pathological behaviour caused by problematic memory access patterns. Interestingly, the cited work did not consider vector memory instructions as an application for this idea.

We also expand our vector ISA to include more instructions which are necessary to vectorise the aggregation algorithms. Instructions are classified and listed in Table 4.1 and those highlighted in bold indicate that they are newly introduced in this chapter. Complete definitions of each one can be found in Appendix B. The `expand` instruction is the inverse of the `compress` instruction [Kog81]. It requires an input vector register with VL entries and a mask register with MVL bits of which VL bits are set. The instruction creates an output vector register whereby the i-th element of the input vector register goes to the position indicated by the i-th bit set of the mask.

Table 4.1: Overview of vector instructions with new additions for aggregation.

class	instructions
integer arithmetic	`add`, `subtract`, `multiply`
bitwise logical	`and`, `xor`, `shift right`, `shift left`
comparison	`not equal`, `less than`, `greater than`
initialisation	`set all`, `clear all`, `iota`
mask	`set mask`, `clear mask`, `and`, `or`, `not`, `popcount`
permutative	`compress`, `shuffle`, `reverse`, **`expand`**
reduction	**`sum`**, **`minimum`**, **`maximum`**
vector length	`set`, `set MVL`, `get`
memory fence	`scalar-vector`, `vector-scalar`, `vector-vector`
CAM-based	`VPI`, `VLU`
other	`merge`, `copy`, `get element`, `set element`

Historically, vector architectures have offered some support for aggregating vectors to scalars in the form of reduction instructions [NH83]. A reduction instruction takes a single vector register as input, applies an associative/commutative operation to all its elements, and outputs a single reduced scalar value. Table 4.1 includes a set of vector reduction instructions which we will use in some of our aggregation algorithms. In our implementation of reductions, there is a partial reduction local to each lane requiring $\frac{VL}{lanes} - 1$ cycles and then $log_2 lanes$ additional cycles needed for interlane reduction. Figure 4.2 shows an example of a sum reduction operation performed on a vector register of eight elements. There are two parallel lockstepped lanes that each processes four elements in three cycles followed by one extra ($log_2 lanes$) cycle of interlane reduction.

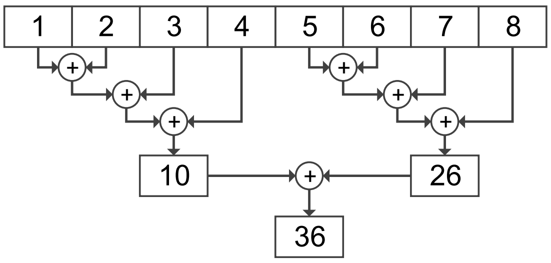


Figure 4.2: Sum reduction when $VL = 8$ and $lanes = 2$.

4.3 Experimental Setup

In this section, we describe the experimental setup. Our goal is to define a representative data-intensive aggregation query, implement it in a variety of ways and evaluate the implementations with a diverse range of parameters. This will help expose the strengths and weaknesses of different algorithm designs. Additionally, we present a scalar aggregation algorithm which we use as a common baseline in subsequent experiments. Finally, we discuss the obstacles to vectorising data aggregation and propose two possible solution paths.

4.3.1 Query and Input Data

In our experiments, we evaluate the SQL query in Figure 4.3. This type of query has been successfully used in prior work to evaluate data aggregations [CR07, YRV11, PR13]; its performance depends highly both on the underlying implementation as well as the characteristics of the input data. r is a two-column table with n rows consisting of a 32-bit integer group key g and a 32-bit integer value v. The result is a three-column output table where each row contains: a group; the frequency of that group—count; and the sum of all values corresponding to that group—sum. Examples of these tables are shown in Figure 4.4. Similar to previous chapters, we emulate the behaviour of a column-oriented OLAP DBMS in which columns are stored contiguously as arrays in memory.

```
SELECT g, COUNT(*), SUM(v)
FROM r GROUP BY g
```

Figure 4.3: SQL code used to evaluate the various aggregation algorithms.

In all the experiments, we fix the number of input rows n at 10,000,000. This value is sufficient to represent behaviour indicative of non-cache resident datasets while also being small enough to simulate experiments to completion in a reasonable time frame. The value column v is a uniform distribution in the interval $[0, 9]$; since this column does not directly affect the performance of the different algorithms, it remains constant in all experiments. We generate 110 variations of the group column g by varying the distribution and cardinality c of the data.

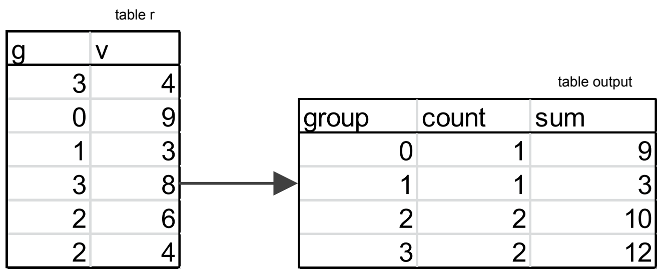


table r
table output

g	v
3	4
0	9
1	3
3	8
2	6
2	4

group	count	sum
0	1	9
1	1	3
2	2	10
3	2	12

Figure 4.4: Example of input table r and output table of SQL code.

We use five unique data distributions similar to the ones used by Cieslewicz et al. [CR07]. (1) **uniform**: a pseudo-random selection in the interval $[0, c)$ with equal probability. (2) **sorted**: a presorted **uniform** distribution. (3) **sequential**: a repeating sequence $\{0, 1, 2, ..., c - 1\}$. (4) **hhitter**: similar to **uniform** however 50% of the data is a single heavy hitting value. (5) **zipf**: a pseudo-random selection in the interval $[0, c)$ with a Zipfian probability.

There are 22 possible cardinalities $c \in \{10,000,000,\ 5,000,000,\ 2,500,000,\ ...,\ 38,\ 19,\ 9,\ 4\}$. Due to the nature of each distribution, c represents a maximum possible cardinality rather than a guaranteed cardinality. For example, it is not always possible to generate a Zipfian distribution where $|g| = c$, therefore—for **zipf**—c represents the upper bound of the domain in which we sample from rather than a strict cardinality. **sequential** is the only distribution where c guarantees both a maximum and an actual cardinality in every case. Unless otherwise stated, cardinality refers to this upper bound.

For the sake of discussion, we group the cardinalities into four divisions. (1) *low* cardinalities [4, ..., 152], e.g. gender of a person. (2) *low-normal* cardinalities [305, ..., 9,765], e.g. date of birth of a client. (3) *high-normal* cardinalities [19,531, ..., 312,500], e.g. a zip or postal code. (4) *high* cardinalities [625,000, ..., 10,000,000], e.g. a passport number.

We assume that the application has a priori knowledge that the **sorted** datasets are already ordered and thus avoids the overhead of resorting. This is normal in DBMSs in which similar metadata is used to choose between alternative algorithms and make optimisations. This assumption also helps identify performance trends independent of a sorting phase.

In some aggregation techniques, it is useful to detect the maximum group key and use it to improve the algorithm's runtime behaviour. In algorithms with a sorting phase—or if the input is presorted—the maximum group key is simply the last value in the array.[1] In algorithms without a sorting phase—excluding presorted input—we locate an exact maximum group key by scanning the entire array g. We find that this

[1]We accidentally only applied this trick to the vectorised algorithms. When finding the maximum group key, scalar experiments using the **sorted** dataset are thoroughly scanned like any other dataset. Scalar results using this dataset are thus a few CPT higher than the optimum. This scanning is responsible for the extra branch mispredictions shown in Appendix A.

adds little overhead compared to the aggregation itself, however, it could be replaced with sampling and some additional checks.

Since we are already looking at many variables, we fix the vector parameters at $MVL = 64$ and $lanes = 4$. These parameters were shown to be reasonable in Chapter 3. They also represent a configuration that we anticipate could eventually appear on the market given current trends. Similar to Chapter 3, we report all our results using CPT—the total number of cycles needed to execute the algorithm divided by the total number of input tuples n.

4.3.2 Scalar Baseline

Here we introduce the baseline algorithm—***scalar***—designed without any vector SIMD instructions. The algorithm uses a separate table in main memory for each aggregation in the query, i.e. there is a count table and a sum table. We divide its implementation into four steps. (1) Find the highest value, maxg, stored in the array g. (2) Clear $maxg + 1$ cells of the output tables count and sum. (3) Aggregate the input arrays g and v to output tables count and sum. Pseudocode for this step is shown in Figure 4.5. (4) Compress the tuples to remove absent groups with NULL results.

```
for each i in n do
    count[g[i]]++;
    sum[g[i]] += v[i];
end for
```

Figure 4.5: Pseudocode for step 3 of the baseline algorithm ***scalar***.

The results for this algorithm are shown in Figure 4.6. For all datasets, the performance is similar in *low* and *low-normal* but then changes drastically entering *high-normal*. When $c = 9,765$, the L1D cache capacity of 32 KB is exceeded. At this point **hhitter**, **uniform** and **zipf** increase their CPT intensely; **uniform** alone exhibits a dramatic $8\times$ increase in CPT. This behaviour is not surprising as a uniform distribution exhibits poor locality when the bookkeeping structures exceed the cache size. In contrast, **sorted** does not take any significant hit in performance in *high-normal* as having the tuples presorted introduces a lot more locality. This effect wears off in *high* and **sorted** experiences a steep slope in its CPT as well.

sequential follows a similar pattern to **sorted** although slightly increases its CPT in *high-normal*. After processing the first 9,765 tuples out of n, the L1D cache will be filled and processing subsequent tuples causes dirty line evictions thus reducing the memory system's performance. These evictions can occur with **sorted** as well, but unlike **sequential**, there will be repeated values stored adjacently causing more locality. This behaviour would suggest that sorting all the datasets will lead to better performance, however, the cost of doing this with a scalar ISA would be very high—especially for a large n.

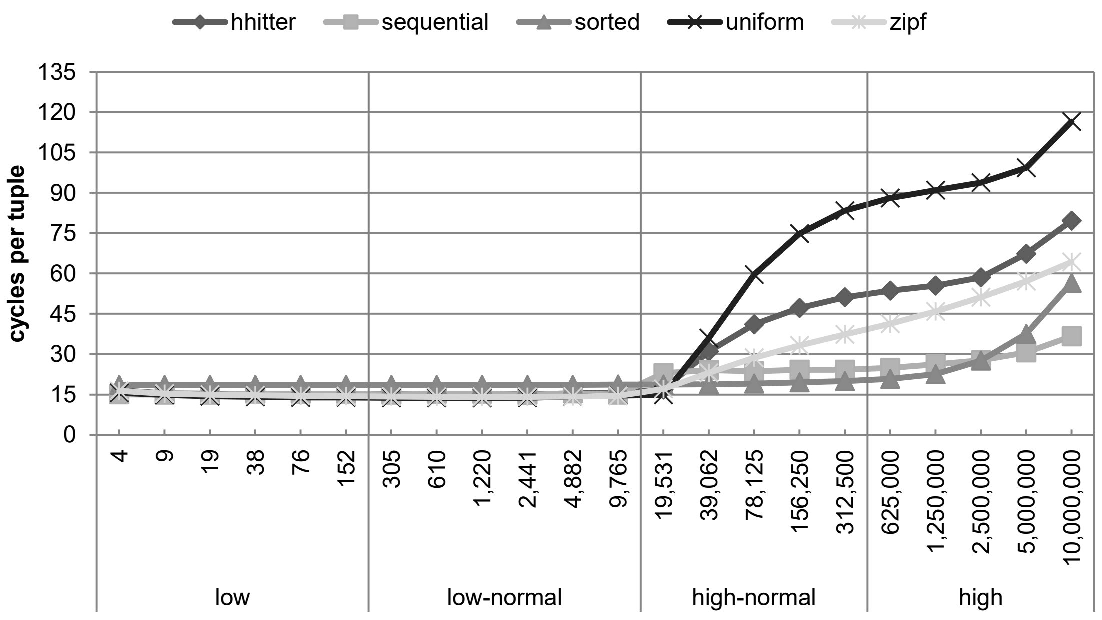


Figure 4.6: Performance results for the baseline algorithm ***scalar***.

4.3.3 DLP and Vectorisation

As discussed in Chapter 1, DLP is accomplished when the same operations are applied to multiple elements of homogeneous data, i.e. a vector of data. DLP can be achieved by leveraging a vector SIMD instruction set such as the one described in Section 4.2. We further categorise DLP as either regular or irregular.

Regular DLP is a form of DLP in which result i of a vector procedure depends only on element i of its input vectors' operands, i.e. every element is independent. A typical vector SIMD instruction set is generally geared towards regular DLP.

Irregular DLP can be defined as DLP where result i of a vector procedure depends on element i of its input vectors' operands and may additionally depend on other results of the vector procedure. It is still DLP as the same operations are applied uniformly on all data, however, the result of one action may depend on the outcome of another action within the same unit of work, e.g. SIMD instruction.

Our reference ***scalar*** baseline is a relatively straightforward algorithm that makes use of tables. Nevertheless—due to the irregularity of the DLP—there are numerous obstacles when vectorising the code. Updating a table is accomplished by—(a) an indexed load from the table (b) modifying the value (c) an indexed store to the table. In a SIMD model of computation, this translates to—(a) gathering multiple table entries to a vector register (b) modifying the vector of loaded values (c) scattering the modified values back to the table. If the indices used in the gather/scatter operations are not unique, i.e. conflicting, the behaviour is undefined and updates can be lost causing erroneous output. We refer to this as a gather-modify-scatter (GMS) conflict.

There are two possible ways to tackle this. One is to *evade* the irregularity by transforming the problem into something more regular and then vectorising it. The other is to *confront* the irregularity directly through the use of novel instructions. In Section 4.4 we evaluate our evasion solutions and in Section 4.5 we evaluate our confrontation solutions.

4.4 Evasion Techniques

In this section, we propose and evaluate two alternative vectorisable solutions using typical vector SIMD instructions.

4.4.1 Standard Sorted Reduce

In this subsection, we explore the benefits of using vector reduction instructions coupled with a standard sorting algorithm taken from Chapter 3, i.e. not VSR sort.

We classify reductions as semi-regular DLP instructions. They are not completely regular because the output element depends on more than input element i, yet, they are not irregular either as there is a single output value and, therefore, output element i does not depend on any other output element.

We evaluate the benefit of using these types of instructions in data aggregation. If the input is sorted, vector reduction instructions can be used directly. If not, the input must be sorted first. Our algorithm is as follows. (1) If not already sorted, g is sorted using v as the associated payload. (2) The sorted g is scanned for runs of repeated keys. Runs can be found by first comparing g[i] with g[i+1] to generate vector masks. The distance between set bits in these vector masks corresponds to the length of a run. These lengths also correspond to the elements of the output column count. (3) The run lengths are used to load and reduce segments of v. Run lengths that exceed the MVL are stripmined.

To sort the input arrays in step 1, we choose the vectorised radix sort algorithm evaluated in Section 3.3.3. It is a good match for this scenario for several reasons. Firstly, it is vectorisable using typical vector SIMD instructions, i.e. nothing esoteric. Secondly, we demonstrated in Chapter 3 that it outperforms quicksort and bitonic mergesort when $MVL = 64$ and $lanes = 4$—the same vector configuration used in this chapter. Thirdly, it has an equal CPT for any input size n, hence making it scalable for larger datasets. Finally, it can be optimised for a particular maximum group key thereby reducing the cost of sorting any particular cardinality. We elaborate on this last point.

In Chapter 3, we evaluated the sorting algorithms using arrays of key-value tuples where the key was taken from a uniform distribution of 32-bit integers. It was likely that keys would span most of the—if not the entire—integer range. As such, we configured radix sort to process all 32 bits of the keys over multiple passes. In this chapter, we experiment with alternative data distributions and cardinalities. In many datasets, the most significant bit of the maximum group key will be lower than the most significant bit of the largest 32-bit integer value. Knowing this allows us to optimise the algorithm and potentially reduce the number of passes. For example, if the group keys of a data distribution can be represented with sixteen bits, we could reduce the number of algorithm passes by a factor of two since radix sort needs only process half the number of bits as usual. In order to enable this optimisation, we must first find the largest group key. As mentioned in Section 4.3.1, we achieve this by scanning the input array g. We find that, in practice, this overhead is offset by the performance gains achieved through the optimisation.

The results of ***standard sorted reduce*** evaluated with all data distributions and cardinalities are shown in Figure 4.7. To make comparisons easier, we keep the scale of the y-axis the same as the scalar baseline for all vector experiments. In Table 4.2, a summary is given of the overall performance by taking the average speedup (and standard deviation) over ***scalar*** for each cardinality division. Highlighted cells indicate that this is the best average performance so far for that particular combination of dataset and cardinality division.

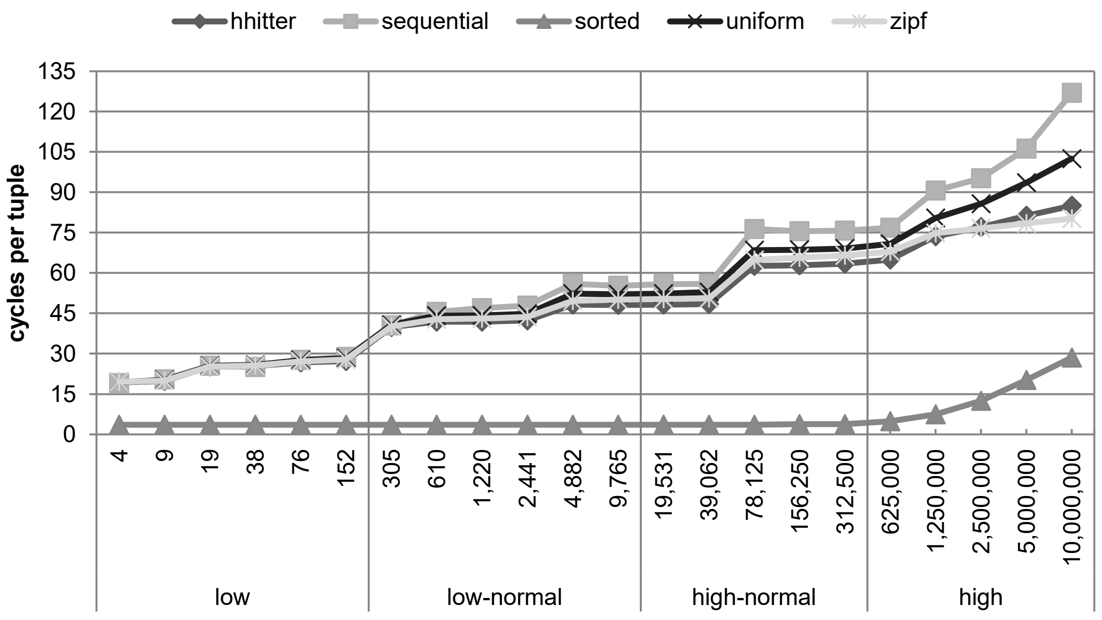


Figure 4.7: Performance results for ***standard sorted reduce***.

Table 4.2: Average speedups (and standard deviation) of ***standard sorted reduce*** over ***scalar***. Highlighted cells mark best result so far.

	low	*low-normal*	*high-normal*	*high*
hhitter	0.7× (0.1)	0.3× (0.0)	0.6× (0.2)	0.8× (0.1)
sequential	0.6× (0.1)	0.3× (0.0)	0.4× (0.1)	0.3× (0.0)
sorted	5.1× (0.0)	5.1× (0.0)	5.2× (0.1)	2.7× (1.0)
uniform	0.6× (0.1)	0.3× (0.0)	0.8× (0.4)	1.1× (0.1)
zipf	0.6× (0.1)	0.3× (0.0)	0.5× (0.1)	0.7× (0.1)

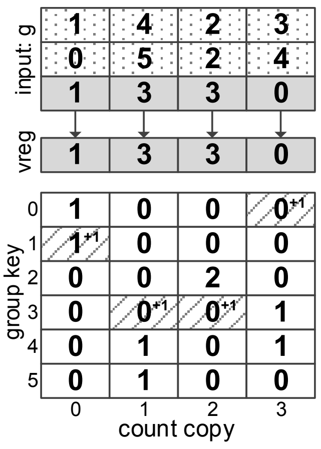
sorted is the only dataset that does not cause additional sorting overhead, as such, we see the cost of the aggregation step itself. Its performance is consistent for *low*, *low-normal* and *high-normal* but then diminishes in *high*. The increasing cardinality causes the average run length to decrease and serialises the algorithm thereby underutilising the vector units.

In most cases, it can be seen that **hhitter**, **sequential**, **uniform** and **zipf** show slowdowns over ***scalar***; only **uniform** exhibits a 1.1× average speedup for *high*. These slowdowns are due to the overhead of sorting the input which often exceeds the total cost of ***scalar***. The effects of our cardinality optimisation can also be seen these results. The overhead of radix sort is moderate for *low* but gradually rises as the cardinality of each dataset increases.

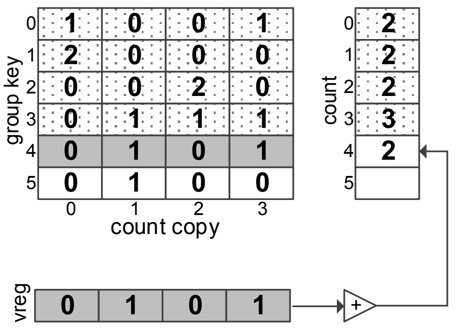
Although being the most efficient DLP-accelerated sorting algorithm using typical vector SIMD instructions, radix sort must undergo significant transformations to be vectorised. As discussed in Chapter 3, the vectorised algorithm suffers from two major bottlenecks. (1) In order to avoid GMS conflicts, its internal bookkeeping structures need to be replicated by the number of elements in a vector register. (2) To ensure sorting stability, each element of a vector register must process a contiguous portion of the input. To achieve this effect, the input must be loaded into a vector register using a strided memory access pattern in lieu of a unit-stride one. In Section 4.5.1, we experiment with VSR sort as a viable alternative to radix sort.

4.4.2 Polytable

It is also possible to make a vectorised translation of ***scalar*** using vector instructions. Steps 1, 2 and 4 can be vectorised directly using typical vector SIMD instructions, however—in a similar vein to radix sort—the third step requires transformation due to GMS conflicts.



(a) Table replication avoids GMS conflicts.



(b) Local tables reduce to a single global table.

Figure 4.8: An illustrative example of ***polytable*** calculating the count table where $n = 12$ and $MVL = 4$.

To circumvent GMS conflicts, we must replicate the output tables count and sum for every element of a vector register, i.e. there are MVL independent versions of each table. Figure 4.8a shows the process of incrementing the count table when $MVL = 4$. In the figure, input array g is arranged in blocks of consecutive MVL elements. The elements with dotted patterns have already been processed. The highlighted values are currently being used to update the table. In this case it can be seen that there are multiple instances of the value 3 in the vector register (vreg). This duplication would cause a GMS conflict if a single table were used, however, since each vector element accesses a local copy, we avoid conflicts entirely.

After the input has been processed, the local copies of count and sum must be reduced to singular global tables. MVL consecutive elements—which form a single group—are loaded into the vector register (vreg) that is then summed together using a reduction instruction. This local to global reduction is illustrated in Figure 4.8b.

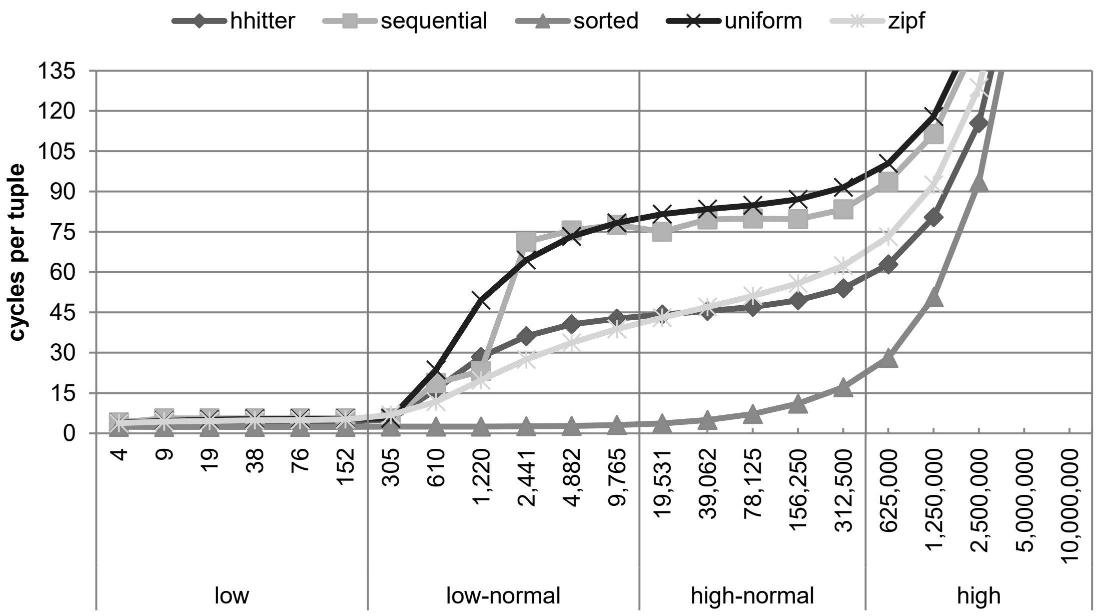


Figure 4.9: Performance results for ***polytable***.

Table 4.3: Average speedups (and standard deviation) of ***polytable*** over ***scalar***. Highlighted cells mark best result so far.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.7× (0.4)	0.9× (1.0)	0.8× (0.2)	0.5× (0.2)
sequential	2.9× (0.4)	0.8× (1.0)	0.3× (0.0)	0.2× (0.1)
sorted	7.6× (0.0)	7.0× (0.6)	2.9× (1.6)	0.4× (0.2)
uniform	3.0× (0.6)	0.7× (0.9)	0.6× (0.3)	0.6× (0.2)
zipf	3.3× (0.6)	0.9× (0.7)	0.5× (0.1)	0.4× (0.2)

The results of ***polytable*** are shown in Figure 4.9 and Table 4.3. For *low*, all datasets exhibit a speedup. Due to the arrangement of the table structures, **sorted** shows the biggest improvement and **sequential** exhibits the least improvement. This is due to the layout of the MVL table copies. Replications are stored contiguously in memory, i.e. the cell for group k's local copy i is adjacent in memory to copy i+1.

Since **sorted** contains long runs of the same group, the number of cache lines accessed is minimal. In contrast, **sequential** has the opposite behaviour. The datasets have runs of ascending groups which causes a strided memory access pattern where the stride is $MVL + 1$ elements, i.e. a diagonal access through the structure. Since the MVL is larger than the number of elements in a cache line, MVL cache lines will be accessed with every memory instruction. All other datasets exhibit performance between these two extremes.

After *low*, the performance begins to decrease. Similar to the ***scalar*** baseline, the tables grow larger than what the cache can accommodate and performance drops. In this case, replicating the tables causes the deterioration to happen sooner. In the scalar baseline, this transition occurs when $c = 9,765$ whereas here it happens when $c = 152$ which is sixty-four—the MVL—times smaller than the former. For **hhitter**, **sequential**, **uniform** and **zipf** the results are always worse than ***scalar***. **sorted** continues to outperform ***scalar*** in *low-normal* and *high-normal* due to the spatial locality of its accesses, however, in *high*, it deteriorates and becomes worse than ***scalar***. A slightly surprising result here is that for **sorted**, *low* and *low-normal* outperform their counterparts in ***standard sorted reduce*** from Section 4.4.1. This due to an unignorable overhead incurred when scanning the input to build the array of run lengths.

4.4.3 Summary

We have evaluated two distinct techniques that vectorise data aggregations through algorithm transformation. If the input is already sorted, there are positive speedups to be gained using ***polytable*** for lower cardinalities and ***standard sorted reduce*** for higher cardinalities. For non-sorted data distributions, it is beneficial to use ***polytable*** if the cardinality is very low. For other combinations of distribution and cardinality, neither of these techniques suffice. These limitations arise due to the transformations necessary to vectorise data aggregation using a typical vector SIMD ISA. These findings motivate us to explore other techniques using novel vector SIMD instructions which will allow us to vectorise the algorithms without these detrimental transformations.

4.5 Confrontation Techniques

In this section we look at alternative solutions that attempt to confront the irregular DLP head on rather than evade it.

4.5.1 Advanced Sorted Reduce

The vectorised radix sort used in Section 4.4.1 suffers from performance bottlenecks caused by algorithm transformation. In Chapter 3, we proposed VSR sort—a novel vectorised implementation of radix sort that avoids replicating its internal table structures and processes the input arrays sequentially. Contiguous portions of the input are read into vector registers using an efficient unit-stride memory access pattern; the algorithm then searches for elements that may cause GMS conflicts and corrects them accordingly before accessing the bookkeeping structures. To enable this new algorithm in a vector architecture, we defined and implemented two new instructions—`VPI` and `VLU`. We first revisit the implementation of these instructions and optimise it for the aggregation datasets. After, we evaluate the sorted reduction algorithm using VSR sort in place of radix sort.

4.5.1-a Parallel Hardware Optimisation

In Chapter 3, we proposed an implementation of `VPI` and `VLU` using a CAM structure coupled with an increment unit. We also introduced an optimisation whereby adjacent elements of the input register can be processed in parallel provided they don't cause a structural hazard in the CAM. The logic to handle these hazards was designed with simplicity in mind. A group of consecutive input keys are checked for conflicts; if there are no conflicts, the CAM is updated in parallel; if a conflict is detected, the group is processed serially.

VSR sort was evaluated with a uniform distribution of integers and only a subset of each value's bits were used when executing `VPI` and `VLU`. As such, we could derive the mathematical probability of a conflict and determined that the most beneficial hardware configurations have two or four CAM lanes. In this chapter, we experiment with five new data distributions, each with twenty-two different cardinalities. Since the variability of these distributions could alter the effectiveness of the CAM lanes, we construct a microbenchmark to evaluate the implementation of `VPI` and `VLU` using these new datasets.

Figure 4.10 shows the average speedup of executing `VPI` with two and four CAM lanes over an implementation with a single CAM lane. We observe that for **sequential** and **uniform** there is a significant advantage when using four CAM lanes over two. For **hhitter**, it is clear that using four CAM lanes exhibits a performance slowdown over two CAM lanes. This is to be expected since 50% of the values are the same thus increasing the probability of collisions and serialisation. For **zipf**, there is an advantage of using four CAM lanes except in the case of *low*. For **sorted**, there is almost no advantage when using four CAM lanes, however, for *high* there is a small benefit when using two CAM lanes. It is clear from these results that the optimal hardware configuration depends a lot on the data distribution and cardinality.

These observations lead us to refine the implementation of `VPI` and `VLU`. In the case of four CAM lanes—instead of serialising the updates on encountering a collision, we can instead try two CAM lanes for each pair of adjacent elements. If both pairs conflict internally, only then will the procedure be completely serialised. Figure 4.11 illustrates an example of this optimisation. In the original implementation—labelled parallel—the hardware selects a group of four consecutive elements. Although neighbouring values are non-conflicting in most cases, there are more conflicts when four values are considered together. As such, the operation is completely serialised giving the net effect of a configuration with one CAM lane. In our improved implementation—labelled parallel optimised—the hardware takes into account that numerous pairs of values don't conflict internally. It can schedule values from these pairs in parallel and achieve the net effect of a configuration with two CAM lanes. The hatched box represents the time

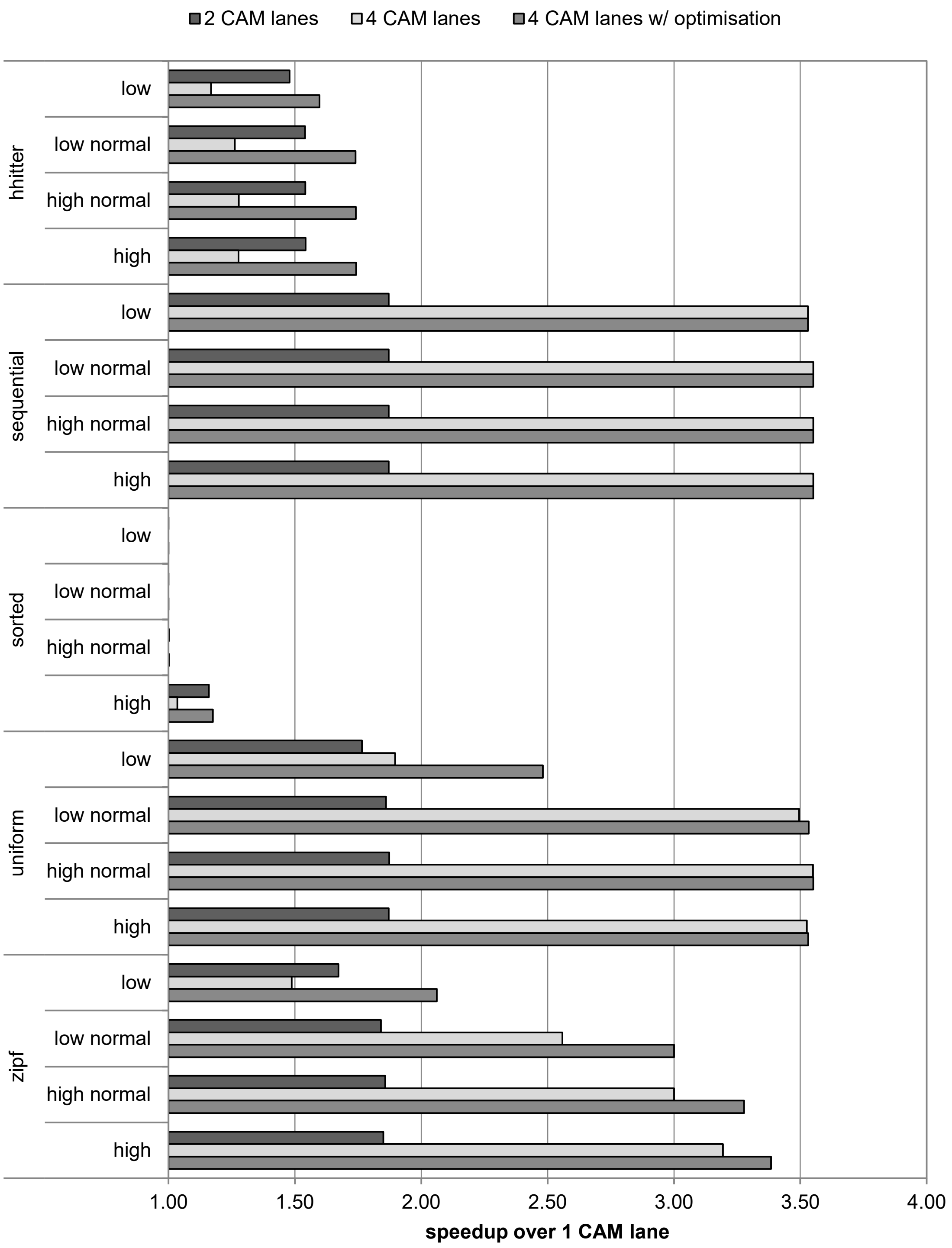


Figure 4.10: VPI speedups over one CAM lane when using multiple CAM lanes.

saved over our original implementation, i.e. parallel. In this case it is $\frac{3}{8}$ blocks or a $1.6\times$ speedup. This is quite a straightforward transition since the logic to detect a collision between four values already contains the logic to detect a collision between any pair of these values, i.e. all the comparators needed are already present, only the control logic changes. Further optimisations may be possible, e.g. not selecting adjacent elements, however, this may complicate the design as it is still necessary to keep the sequential semantics of `VPI` and `VLU`.

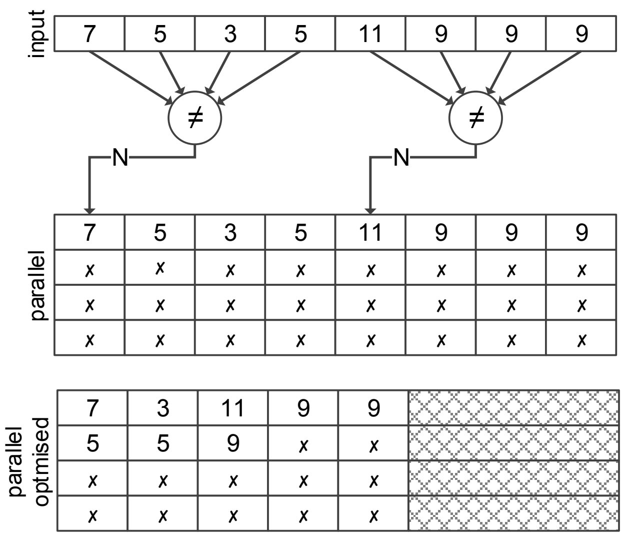


Figure 4.11: Optimised parallel hazard detection for hardware with four CAM lanes.

Figure 4.10 also plots the results of configuration with four CAM lanes using the new optimised conflict detection. Several of the datasets are able to take advantage of its fallback mechanism. At worse, the results are equal to a configuration with two CAM lanes, i.e. there are no slowdowns. However, there are also some cases where the performance achieved is better than both of the other configurations due to a composite effect of having the four CAM lanes available and not completely serialising the groups with conflicts, e.g. for **zipf**. We run the remainder of our experiments using this optimisation.

4.5.1-b Algorithm Evaluation

We now evaluate the same algorithm used in ***standard sorted reduce*** but replace radix sort with VSR sort while keeping all other steps equal. We also apply the optimisation described in Section 4.4.1 whereby the number of algorithm passes can be reduced if the maximum group key is less than the largest possible 32-bit integer. The results are shown in Figure 4.12 and Table 4.4. Since the **sorted** dataset can skip the sorting step, its behaviour and performance remain equal to ***standard sorted reduce***; these cases are marked with a Ξ symbol.

For **hhitter**, **sequential**, **uniform** and **zipf** the results are always better than ***standard sorted reduce***. There are still some slowdowns over ***scalar*** for *low* and *low-normal*.

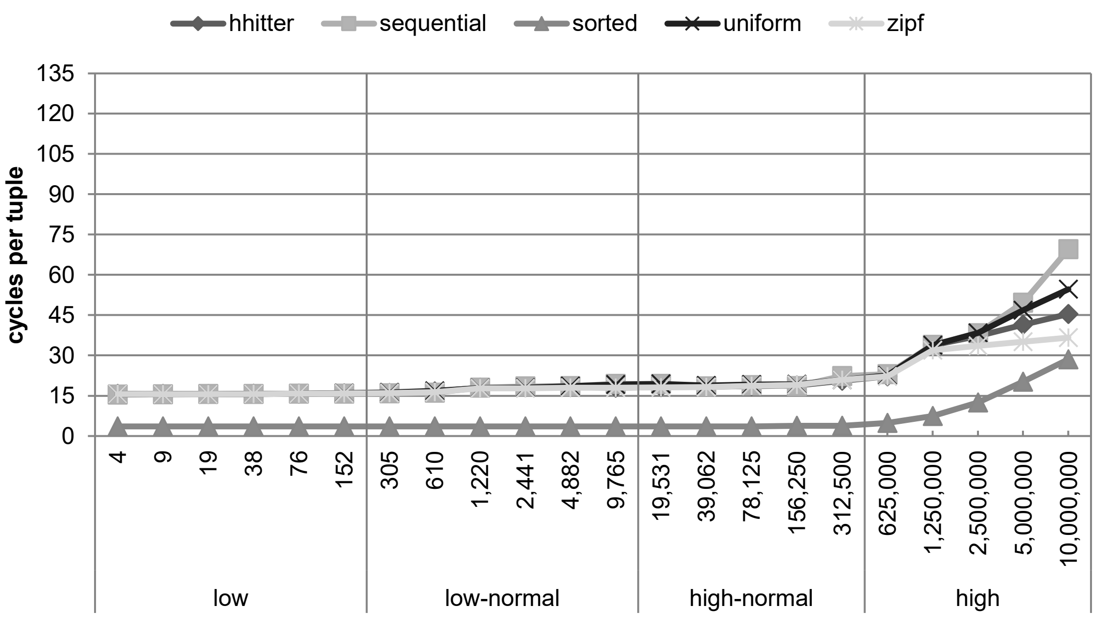


Figure 4.12: Performance results for ***advanced sorted reduce***.

Table 4.4: Average speedups (and standard deviation) of ***advanced sorted reduce*** over ***scalar***. Highlighted cells mark best result so far.

	low	*low-normal*	*high-normal*	*high*
hhitter	1.0× (0.0)	0.9× (0.0)	2.0× (0.7)	1.8× (0.4)
sequential	1.0× (0.0)	0.9× (0.1)	1.2× (0.1)	0.7× (0.2)
sorted	5.1× (0.0) Ξ	5.1× (0.0) Ξ	5.2× (0.1) Ξ	2.7× (1.0) Ξ
uniform	0.9× (0.1)	0.8× (0.0)	2.7× (1.4)	2.7× (0.7)
zipf	1.0× (0.1)	0.8× (0.0)	1.5× (0.4)	1.6× (0.2)

Despite the performance of VSR sort being better than radix sort, the overhead is still too high to surpass the CPT of ***scalar*** for lower cardinalities. For *high-normal*, this sorting overhead becomes less significant and we achieve speedups in all cases.

For *high*, **hhitter**, **uniform** and **zipf** continue to exhibit speedups whereas **sequential** shows a slowdown. The reason for this is twofold: (1) **sequential** exhibits good locality in *high* for ***scalar*** thereby having better performance relative to the other three datasets. (2) The average vector length is reduced to values below the MVL in *high*. For example, when $c = 10,000,000$ the vector length of every reduction is 1 and this reduces performance considerably. This second point also affects **hhitter**, **uniform** and **zipf** for *high*, but to a lesser extreme than **sequential**; as mentioned in Section 4.3.1, c represents an upper bound rather than an absolute value for the cardinalities of these four datasets.

4.5.2 Monotable

The principal problem with the ***polytable*** algorithm of Section 4.4.2 is that the replication of tables destroys any locality that may otherwise be present in the ***scalar*** baseline. Here we propose an alternative implementation called ***monotable*** which draws from the novel instructions—`VPI` and `VLU`—used in ***advanced sorted reduce***.

`VPI` and `VLU` cannot be used to vectorise ***scalar*** directly, i.e. without transformations. This is because the calculation of the sum column in Figure 4.4 requires a type of conflict correction with an increment based on the input array v. `VPI` and `VLU` correct conflicts, but only for increments of 1; these instructions could, however, be used to calculate the count column.

`VPI` and `VLU` use a hardware implementation based on a CAM and an increment unit. We propose reusing this hardware structure and building new functionality on top of it. We define a new set of instructions called `Vector Group Aggregate (VGAx)` that can aid us further when vectorising data aggregation. There are three operations supported which form the new instructions—sum (`VGAsum`), minimum (`VGAmin`) and maximum (`VGAmax`). Each `VGAx` instruction uses two registers as input—a vector of groups in-g and a vector of values in-v. The instructions produce a vector out of running partial aggregates among values of the same group.

We can implement these instructions with relatively minor additions to the hardware already in place for `VPI` and `VLU`. As an example, we describe `VGAsum`. The semantics are illustrated in Figure 4.13 and the implementation is shown in Figure 4.14. For each input element, instead of incrementing its CAM entry by one as would be done with `VPI`, the entry is summed with the corresponding value in in-v. The semantics resemble `VPI` when the values of in-v are all 1s, however, an important difference is that the output of `VPI` comes from the CAM entry's value before the increment whereas the output of `VGAsum` is taken after the increment. There are two other differences between this implementation and that of Section 3.4.2-c used to calculate `VPI` and `VLU`. (1) Two vector registers are used as inputs instead of one. (2) The increment unit must be generalised to a 32-bit adder with two integer inputs.

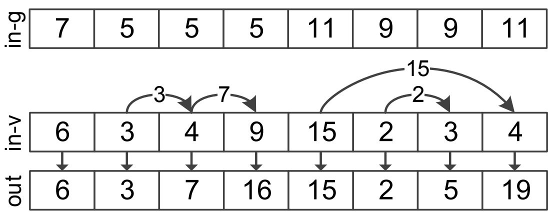


Figure 4.13: Semantics of the `VGAsum` instruction.

We use `VGAsum` to build a vectorised version of ***scalar*** using non-replicated tables with no GMS conflicts. Combining `VGAsum` with `VLU` allows us to update a single table in parallel while avoiding conflicts. Figure 4.15 shows the pseudocode of this step. The masked `scatter` instruction could optionally be replaced with a `compress` followed by a non-masked `scatter`.

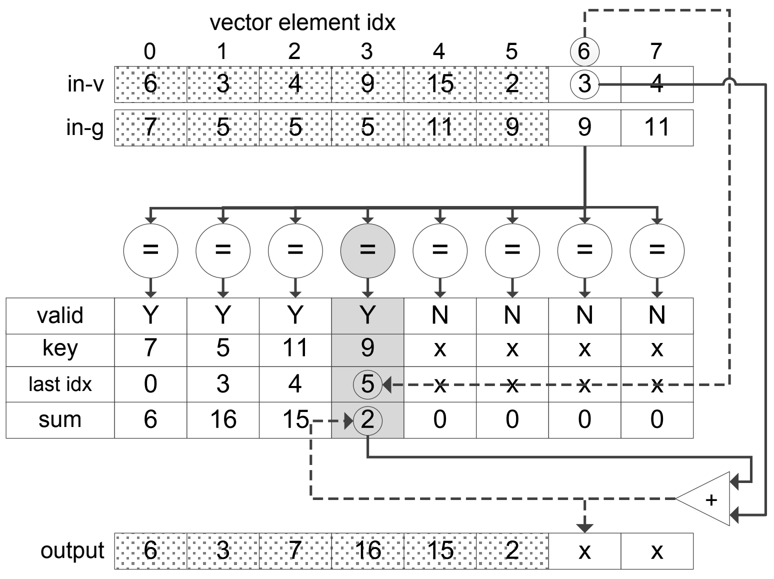


Figure 4.14: Hardware implementation of `VGAsum`.

```
1: $\vec{v2} \leftarrow$ vgasum($\vec{v0}$, $\vec{v1}$)                    ▷ groups in $\vec{v0}$ & values in $\vec{v1}$
2: $m0 \leftarrow$ vlu($\vec{v0}$)
3: $\vec{v3} \leftarrow$ gather(**base**=*table*, **idx**=$\vec{v0}$, **mask**=$m0$)
4: $\vec{v4} \leftarrow$ vadd($\vec{v2}$, $\vec{v3}$)
5: scatter(**base**=*table*, **idx**=$\vec{v0}$, **vals**=$\vec{v4}$, **mask**=$m0$)
```

Figure 4.15: Pseudocode for updating a table using `VGAsum`.

Figure 4.16 and Table 4.5 show the results of ***monotable***. The graph resembles the trends found in ***scalar*** (see Figure 4.6) but with lower CPTs. For *low*, ***monotable*** exhibits good performance for **hhitter**, **sequential**, **uniform** and **zipf** and outperforms ***polytable***—the only evasion method that was useful for this cardinality division. **sorted** is not as fast as ***polytable*** for *low* and *low normal*, which is understandable since the majority of the `VGAsum` instruction's input will cause CAM port conflicts and, therefore, pay the maximum latency. In contrast, ***monotable*** outperforms ***polytable*** in all cases for **sorted** in *high-normal* and *high*.

It can be seen that ***monotable*** has consistent performance for lower cardinalities, but for higher cardinalities **hhitter**, **sequential** and **uniform** become worse whereas **sequential** and **sorted** remain relatively stable. This behaviour is related to the locality of memory accesses. When $c \leq 9,765$, the data structures can reside fully in the L2 cache. When this cardinality is exceeded—depending on the distribution of the data—it may destroy the locality. Despite this behaviour, all the datasets in the higher cardinalities exhibit a speedup and beat the ***polytable*** method in every case. Compared with ***advanced sorted reduce***, sometimes the performance is better and sometimes worse.

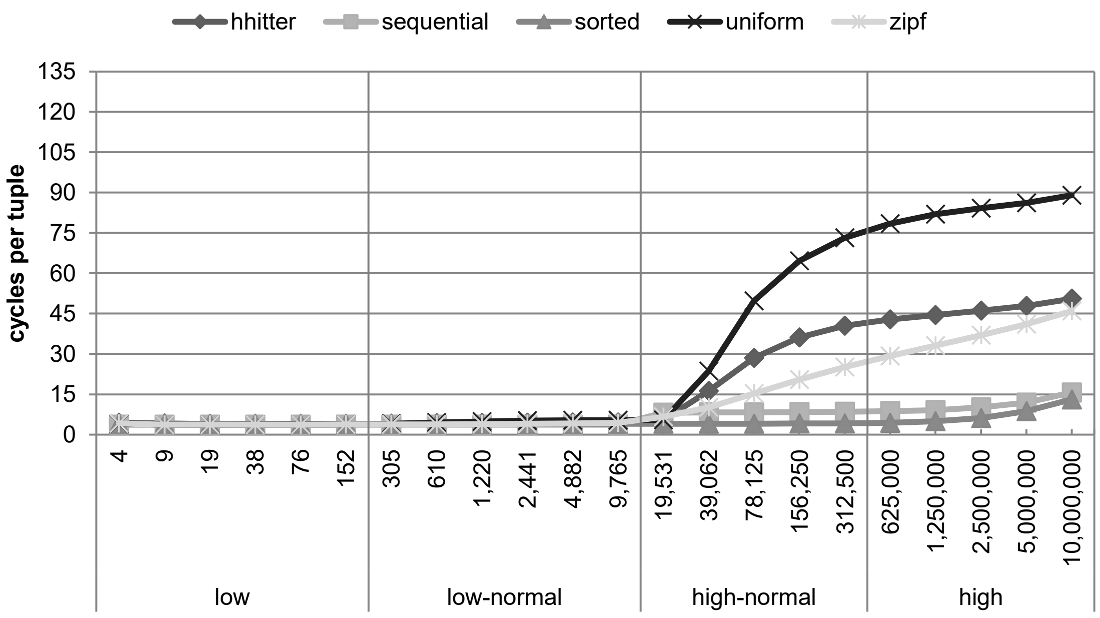


Figure 4.16: Performance results for ***monotable***.

Table 4.5: Average speedups (and standard deviation) of ***monotable*** over ***scalar***. Highlighted cells mark best result so far.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (0.1)	3.5x (0.1)	1.8× (0.8)	1.3× (0.1)
sequential	4.1× (0.0)	4.1× (0.1)	2.9× (0.0)	2.7× (0.2)
sorted	4.6× (0.0)	4.6× (0.0)	4.7× (0.0)	4.5× (0.2)
uniform	3.8× (0.1)	2.9× (0.3)	1.5× (0.7)	1.2× (0.1)
zipf	4.0× (0.1)	3.5× (0.2)	2.0× (0.4)	1.4× (0.0)

4.5.3 Partially Sorted Monotable

We observe that ***monotable*** works particularly well for the lower cardinalities. For higher cardinalities, some of the datasets lose their cache locality and exhibit rapid increases in CPT. **sorted** and **sequential**—the datasets that do not lose their locality—maintain more consistent behaviour. We estimate that to achieve the optimal behaviour of ***monotable***, the input does not necessarily have to be fully sorted but instead be partitioned in such a way that maximises temporal locality.

In ***advanced sorted reduce***, we use VSR sort to fully sort the input before reducing it. Each pass of VSR sort orders the input according to a subset of bits of each value, building from the order already found by previous passes. By default, VSR sort finishes after the last pass processes the most significant bits of the values resulting in a completely sorted input. Each pass contributes to the algorithm's overhead. If only sorting on a subset of each value's bits is necessary, the number of passes could be significantly reduced. In ***monotable***, it is not paramount that all group keys be stored together contiguously like in the ***sorted reduce*** methods. Instead, it should be sufficient to position repeated groups keys just close enough to one another that nothing in between will evict that group key's line from the cache. Accordingly, we propose partially sorting the inputs with higher cardinalities before executing ***monotable***.

We modify VSR sort to perform a single pass of the algorithm on a subset of bits between the most significant bit of the maximum group key and a user-specified offset. As the L2 cache in our experiments is 256 KB and each group key requires 4 bytes, up to 16 bits of each value could be ignored when sorting. Using a configuration that considers the remaining 16 bits would divide the input into partitions with a maximum of 65,536 unique groups keys. In practice, we find the best performs is achieved when sorting the most significant 8 bits of the datasets in *high-normal* and increasing this gradually to 11 bits for the largest cardinality in *high*. This way, the partial sort can be limited to a single pass in all cases. We do not need to partially sort any datasets in *low* and *low-normal* as these exhibit good temporal locality already.

Figure 4.17 and Table 4.6 show the results of ***monotable***. Since we need not partially sort the lower cardinalities or the **sorted** dataset, their behaviour and performance remain equal to ***monotable***; these cases are marked with a Ξ symbol.

For *high-normal* and *high*, there is a significant increase in performance for **hhitter**, **uniform** and **zipf**. These results are considerably better than ***polytable***, ***monotable*** and either ***sorted reduce*** methods. **sequential** is the only dataset that takes a hit in performance over ***monotable*** for the higher cardinalities. This degradation is because **sequential** already exhibits enough spatial locality to compensate for a lack of temporal locality, therefore, partially sorting the input only adds to its CPT.

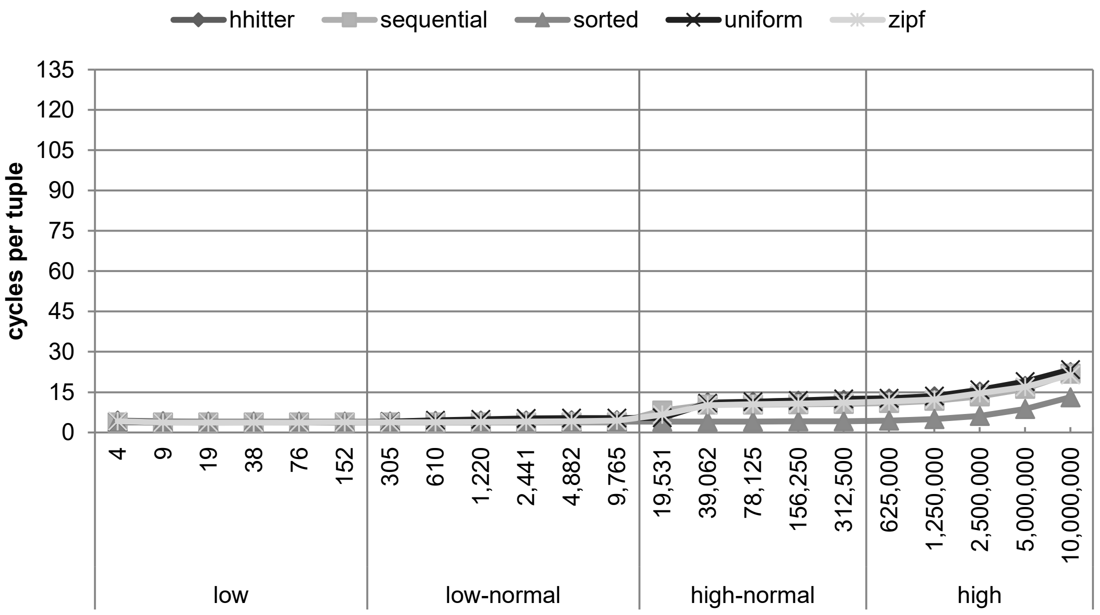


Figure 4.17: Performance results for ***partially sorted monotable***.

Table 4.6: Average speedups (and standard deviation) of ***partially sorted monotable*** over ***scalar***. Highlighted cells mark best result so far.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (0.1) Ξ	3.5× (0.1) Ξ	3.5× (0.6)	3.9× (0.3)
sequential	4.1× (0.0) Ξ	4.1× (0.1) Ξ	2.4× (0.2)	2.0× (0.2)
sorted	4.6× (0.0) Ξ	4.6× (0.0) Ξ	4.7× (0.0) Ξ	4.5× (0.2) Ξ
uniform	3.8× (0.1) Ξ	2.9× (0.3) Ξ	4.8× (1.8)	5.9× (0.8)
zipf	4.0× (0.1) Ξ	3.5× (0.2) Ξ	2.8× (0.4)	3.4× (0.3)

4.5.4 Summary

We have evaluated a broad range of datasets using five vectorised data aggregation techniques—each with varying success—that either evade or confront the irregular DLP inherent to this workload. In the majority of cases, using an evasion technique adds too much overhead to be useful whereas our proposed confrontation techniques show more promising results. Table 4.7 summarises the best results for all data distributions and cardinality divisions. Each cell provides an average speedup over ***scalar***. The ‡ symbol indicates it may not be practical to detect this configuration at runtime in order to apply the most suitable algorithm.

Table 4.7: Best average speedup (and algorithm) over ***scalar***.

	low	*low-normal*	*high-normal*	*high*
hhitter	3.9× (mono)	3.5× (mono)	3.5× (psm)	3.9× (psm)
sequential	4.1× (mono)	4.1× (mono)	2.9× (mono) ‡	2.7× (mono) ‡
sorted	7.6× (poly)	7.0× (poly)	5.2× (sr)	4.5× (mono)
uniform	3.9× (mono)	2.9× (mono)	4.8× (psm)	6.0× (psm)
zipf	4.0× (mono)	3.5× (mono)	2.8× (psm)	3.4× (psm)

In all cases, the results are positive although there is no single algorithm that matches all of the configurations. The best speedup is achieved using a variety of different techniques. For *low* and *low-normal*, the non-sorted datasets fare best using ***monotable*** (mono) whereas **sorted** achieves the highest speedup using ***polytable*** (poly). For **hhitter**, **uniform** and **zipf** the best method for *high-normal* and *high* is ***partially sorted monotable*** (psm) and for **sequential** the best choice is ***monotable***. **sorted** performs best using either of the ***sorted reduce*** (sr) methods for *high-normal* and ***monotable*** for *high*.

In most situations, we have enough information to dynamically choose the best method for a particular combination of dataset and cardinality. In general, the rule is to apply ***monotable*** to non-sorted datasets for lower cardinalities and ***partially sorted monotable*** for higher cardinalities; for sorted datasets, ***polytable*** can be used for lower cardinalities and ***sorted reduce*** and ***monotable*** for higher cardinalities. Only detecting the case of **sequential** with higher cardinalities would prove difficult, however, the difference between ***partially sorted monotable*** and ***monotable*** for these two cases is not overly significant. Using the ideal algorithm selection yields a 4.21× total average speedup whereas a realistic algorithm selection—where **sequential** with higher cardinalities is evaluated using ***partially sorted monotable***—yields 4.15×. This slowdown is a mere 1.3%.

4.6 Related Work

In this section we discuss the related work. We separate the section into two subsections. The first looks at work related to parallel data aggregation acceleration. The second looks at alternative hardware proposals that attempt to tackle irregular DLP vectorisation that could be amenable to aggregations.

4.6.1 Parallel Aggregation Acceleration

Zhou and Ross [ZR02] explore the implementation of DBMS algorithms with basic SIMD multimedia instruction extensions. They only mention GROUP BY aggregation in passing and do not find a way to implement it with SIMD instructions unless first sorting the input. We have seen that fully sorting the input has major overhead for radix sort—an evasion algorithm—and even a significant overhead for VSR sort—a fast confrontation algorithm.

Ye et al. [YRV11] evaluate how various aggregation methods scale with multiple threads. Although multithreading and SIMD are not completely comparable, they do observe some similar behaviour found in our experiments. We implement table replication to avoid GMS conflicts between vector register elements; Ye et al. do the same for read-modify-write conflicts that can occur between threads. Similar to our observations, they observe a massive loss in performance when cardinalities exceed the L1D cache size. In general, our motivation is such that vector acceleration is more efficient than multithreading. We achieve a 7.6× speedup in some cases using vector extensions within a single core; achieving this result using multithreading would require—at minimum—eight cores. That said, vector SIMD and multithreading are not mutually exclusive and can complement each other nicely with the right algorithm design.

Polychroniou and Ross [PR13] propose SIMD optimisations when aggregating datasets similar to **zipf** and **hhitter**. Their approach uses multimedia SIMD extensions, although in a very different way to our vector instructions. Where we use a struct-of-arrays model and completely vectorise our algorithms, they use an array-of-structs model and partially vectorise their algorithm in an orthogonal direction. In all of our vector implementations, we vectorise with n, i.e. along the output arrays count and sum. Polychroniou and Ross instead pack each count[i] and sum[i] adjacently in SIMD registers to process both together. Although this approach offers some benefits, it has limited applicability and little scalability since the amount of parallelism depends on the number of aggregation operations in the query. In many cases this will be one aggregation and, therefore, offers no advantage over a scalar algorithm. Additionally, the number of aggregations and their datatypes in a query may not be available until runtime time thereby adding an obstacle when defining the appropriate memory structures. This scrutiny is not a criticism of their work, but instead an observation on the limitations when using simple multimedia extensions to vectorise complex algorithms like data aggregation. Our work uses a true vector ISA and we vectorise our algorithms in the direction of the arrays. This type of vectorisation is very beneficial for column-store databases—typically used for OLAP—which favour a struct-of-arrays model over an array-of-struct model.

Power et al. [PLH+15] utilise GPGPUs to perform data aggregation. They argue that when using discrete off-chip GPUs, there is a high overhead associated with data movement as well as the coordination between the CPU/OS and the GPU. They motivate using integrated GPUs, i.e. GPUs on the same die as the CPU. They present two techniques. The first approach uses a replicated table for each GPU thread. The second approach uses a single lock-free table with the GPU threads repeatedly trying to perform atomic read-modify-write updates. The first technique shows benefits for very low cardinalities whereas the second approach proves to be good for very high cardinalities. For middle/normal cardinalities, neither approach works very well. While GPU and vector hardware organisations are not directly comparable, we do see some commonalities between the techniques used to parallelise aggregation. Their first technique is very similar to our ***polytable*** method, and—congruous to our observations—they also find table replication causes rapid performance deterioration as cardinality increases. Their second technique is similar to our ***monotable*** approach in that we both try to update a single table with potential GMS conflicts. Power et al. propose using atomic memory instructions, however, they find that contention is too frequent to achieve good performance if the cardinality is not very high. Our ***monotable*** method does not use such instructions; instead, we rectify any conflicting operations that would cause GMS conflicts in the registers before even making the memory access. We show experimentally that this is useful for a variety of data distributions and cardinalities.

4.6.2 Hardware Support for Irregular DLP

The following proposals have already been discussed in Chapter 3, however, we consider them again in the context of data aggregations.

Scatter-add [AED05] is a proposal for streaming architectures that allows a conflict-free gather-modify-scatter operation on an array using one instruction. There are several significant differences with our proposal. The first and foremost is that scatter-add cannot be used to implement VSR sort. There are two reasons for this—(1) It lacks a return path for original values in the array before the modifications, and (2) it lacks the deterministic ordering semantics found in `VPI` and `VLU`. Scatter-add, therefore, has limited applicability to our proposed algorithms in which partially sorting is a major component. Secondly, scatter-add is an extension to the processor's memory hierarchy. Adding a major feature to memory can be less modular, highly intrusive and more difficult to verify. In contrast, the `VGAx` instructions use only vector registers as input and output. Thirdly, although scatter-add is beneficial in the sense that a single instruction expresses a lot of work, the same behaviour can be emulated by

VGAsum. Since the VGAx instructions generate a running cumulative for each group in a vector register, this could have uses beyond aggregation, e.g. a customised prefix sum operation. Finally, we are building upon hardware that is already in place for the instructions VPI and VLU. The addition required to implement the VGAx instructions are minor.

Atomic vectors operations [KKS+08], and more recently, AVX-512CD [Int14b], are both solutions from Intel that attempt to solve the problem of GMS conflicts. Both of these proposals operate with a best-effort mechanism as follows. A mask register with all its bits set is coupled with the vector GMS procedure. The processor attempts to execute as many non-conflicting elements of the procedure as it can and clears the associated mask bits of successful outcomes. The programmer is responsible for placing the GMS procedure inside a loop that is dependent on the state of the mask register. This means in the worst case scenario the operation will be completely serialised inside a loop with a difficult to predict exit condition. Since each retry requires loading, modifying and storing the data again, it could even lead to more operations than its scalar counterpart. We anticipate that for datasets with low cardinalities and skewed distributions, the number of retries will be high and thus impede performance. This problem will be exacerbated further as vector SIMD register widths increase. VPI, VLU and the VGAx instructions are different because they exist as self-contained non-memory instructions. This difference means GMS conflicts are resolved completely and deterministically before committing to the memory hierarchy. We have shown experimentally that datasets with low cardinalities and skewed distributions perform well with a large MVL. Furthermore, it is not obvious how VSR sort could be constructed from either the atomic vector operations or AVX-512CD. We have demonstrated that partially sorting the input using VSR sort for high cardinalities provides major performance improvements and, consequently, remains an important part of this work.

4.7 Conclusions

As the amount of data increases exponentially each year it is important that data aggregation algorithms can scale accordingly. In this chapter, we have looked at vector SIMD instructions as a means to accelerate GROUP BY data aggregations. We have found that this is not a trivial target due to the irregularity of the DLP in this workload.

We have made experiments with the vector ISA proposed and developed in Chapter 2. We have found that this ISA has limitations since it only permits us to evade the irregular DLP through performance-degrading algorithm transformations. Based on this realisation, we have proposed the use of novel vector instructions which directly confront this irregularity and allow us to vectorise the algorithms directly without alteration. We have made detailed evaluations using multiple algorithms taken from both *evasion* and *confrontation* techniques.

We have observed that the evasion techniques have limited applicability unless the input is presorted, otherwise, the confrontation techniques prove to be more advantageous. The latter draws heavily from our work in Chapter 3. We have discovered that VPI and VLU—and their associated sorting algorithm—can aid data aggregation, especially with the realisation that the input need not be fully sorted. With minimal modifications, we have extended the base hardware used in this proposal to accommodate data aggregation further by defining a suite of new instructions called `VGAx`.

We have found that the best algorithm depends highly on both the distribution and cardinality of the input. In most cases, this can be detected at runtime to make a choice dynamically. Using a combination of these techniques, we have achieved speedups over a scalar baseline between 2.7× and 7.6× for a maximum vector length of 64 and four lockstepped lanes.

CHAPTER 5

A Study on Area, Energy and Power

5.1 Introduction

In this chapter, we perform an area, energy and power study of our vector SIMD extensions. As a case study, we use the sorting algorithms of Chapter 3 to evaluate energy and power. These algorithms will allow us to examine the scalability of the hardware, i.e. maximum vector length and lanes, in terms of energy and power. At the same time, we can also evaluate the energy and power from an algorithmic perspective in order to gain a deep understanding of how each sorting implementation influences the activity of the different hardware units. We discover that our own VSR sort uses up to 20.1× less dynamic energy than the scalar baseline and uses between 5% *less* and 22% *more* power, depending if a single-lane or multi-lane implementation is used.

To model the area, energy and power of our baseline architecture we use McPAT [LAS$^+$09] in conjunction with runtime statistics generated from our simulation framework. We use a high-performance 32 nm technology clocked at 2.67 Ghz and configure its out-of-order model to use the simulation parameters found in Table 2.2. We modify and extend McPAT extensively to model our vector additions. We add new register files for vector and mask registers. The modelled vector ALUs are copies of the ALU used in the scalar core. If the number of lanes is greater than one, the permutative instructions, e.g. `compress` or `shuffle`, are implemented using a crossbar. As described in Chapter 3 and 4, `VPI`, `VLU` and `VGAx` are implemented using a multiported CAM, adders and a conflict detection unit.

5.2 Area

Here we assess the area requirements of our vector extensions. McPat estimates an area of 13.26 mm^2 for the baseline scalar architecture, i.e. without vector support. Our vector extensions—including the CAM unit—introduce an overhead between 9% and

19% as shown in Table 5.1. This increase in area might be considered reasonable since a vector ISA is versatile enough to accelerate many types of algorithms and applications.

	1 lane	2 lanes	4 lanes
mvl8	14.46 mm^2 (9%)	14.75 mm^2 (11%)	15.43 mm^2 (16%)
mvl16	14.50 mm^2 (9%)	14.78 mm^2 (11%)	15.46 mm^2 (17%)
mvl32	14.56 mm^2 (10%)	14.86 mm^2 (12%)	15.55 mm^2 (17%)
mvl64	14.80 mm^2 (12%)	15.09 mm^2 (14%)	15.79 mm^2 (19%)

Table 5.1: Total area (and overhead) of processor with vector extensions.

Since the area overhead of our vector extensions is at most 19%, the differences in static energy consumption will relate more to execution time rather than the new hardware structures. We have observed that, in all but one case, the vectorised algorithms exhibit significant decreases in execution time and, as such, the static energy consumption will be decreased proportionally. In Section 5.3 we focus on dynamic energy as this pertains to the behaviour of the algorithms and their instruction mix. In Section 5.4 we look at power—both dynamic and static.

5.3 Runtime Dynamic Energy

In this section, we evaluate the runtime dynamic energy using the sorting algorithms from Chapter 3 as a case study. Figure 5.1 shows the dynamic energy consumption of each algorithm run on each hardware configuration using the **large** dataset. Two things are immediately clear from this diagram—(1) that all the vectorised solutions consume less dynamic energy than the scalar baseline, and (2) that the vectorised algorithms consume different amounts of energy to each other and exhibit different patterns from each other with respect to the MVL and number of lanes.

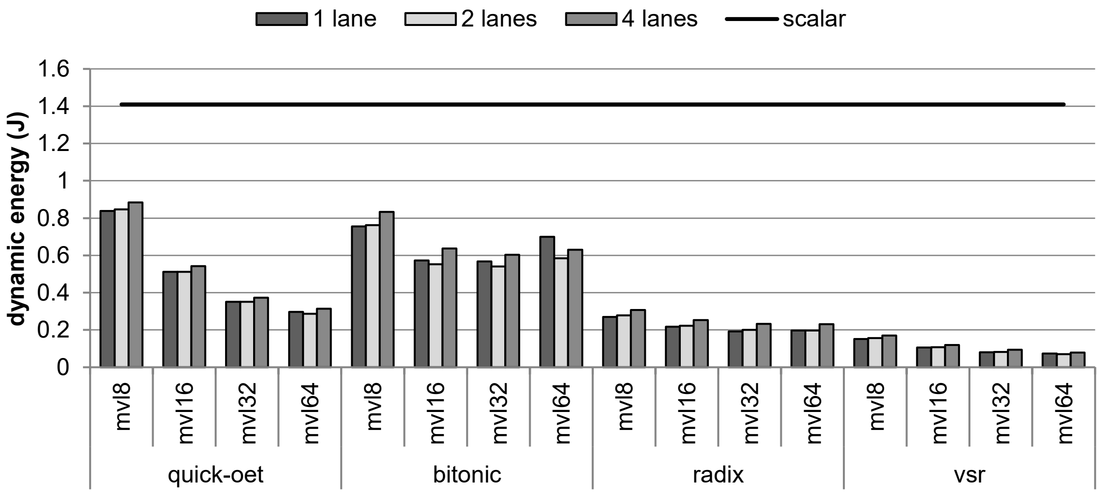


Figure 5.1: Runtime dynamic energy comparison of all the sorting algorithms.

5.3.1 Scalar and Vectorised Quicksort

We first compare the scalar baseline to the vectorised quicksort with OET cleanup. These algorithms are more easily comparable because both are variants of quicksort. Figure 5.2 shows a detailed breakdown per processor component of both algorithms using all the hardware configurations.

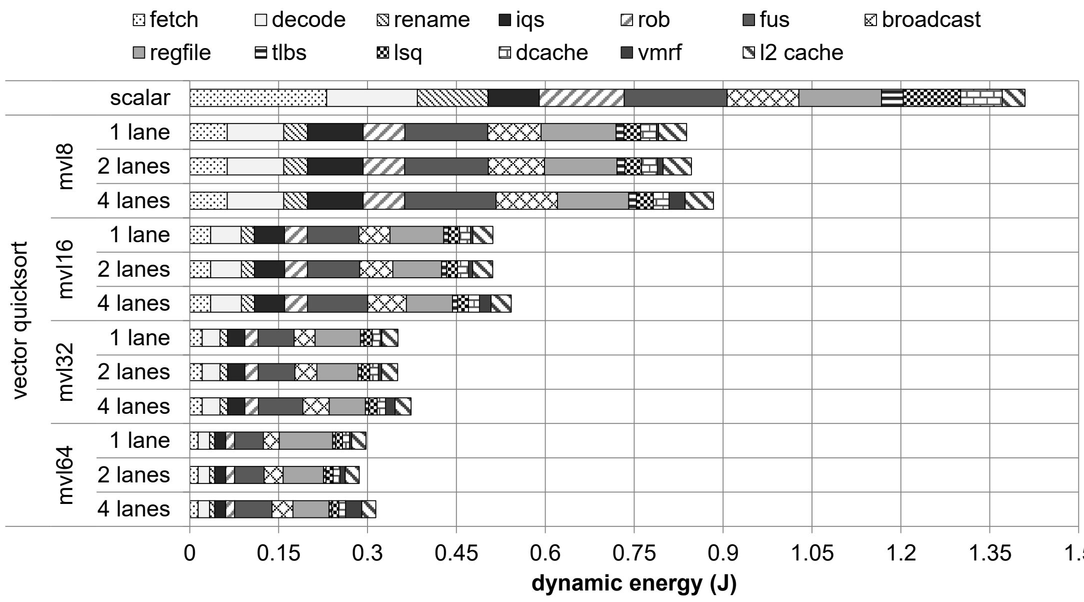


Figure 5.2: Dynamic energy breakdown of scalar and vectorised quicksort algorithms.

The scalar baseline uses approximately 1.41 J of dynamic energy. The front end (fetch, decode and rename) uses 36% of this energy. The out-of-order mechanism (issue queues and reorder buffer) uses 16%. Value generation (the functional units and result broadcast) uses 21%. The register file 10% and the memory-related units (TLBs, load/store queue, L1D and L2 caches) use the remaining 17%.

The most simple vector configuration using a MVL of eight with a single lane consumes 0.84 J of energy—59% of the scalar baseline's dynamic energy consumption—while achieving a 1.4× execution time speedup. The front end components exhibit a 2.5× decrease in energy. We correlate this with a similar reduction in the number of lines fetched from the instruction cache (see Appendix A). The out-of-order mechanism sees a 1.4× decrease in energy. Although the issue queues increase their energy slightly as a result of the additional vector clusters, the reorder buffer decreases its energy significantly due to the reduction in speculative instructions. The memory-related units reduce their energy consumption by 2.0×. Since the vector unit bypasses the L1D cache, there is a significant decrease in energy for this structure. The memory requests instead go directly to the L2 cache; this structure has an increase in energy, however, this increase is offset by the savings in the L1D. It is important to remember that the cache hierarchy is write through with respect to L1D $\rightarrow$ L2, therefore, the scalar baseline already includes a lot of L2 write activity.

Where the scalar unit may access a cache line multiple times when processing consecutive elements, a unit-stride vector memory instruction can consolidate all these accesses into a single read or write. Additionally, the vector unit avoids the load/store queue since the ISA guarantees that vector memory instructions will not alias with each other. These instructions are instead more efficiently handled by the simple non-associative Vector Memory Request File (VMRF) structure previously described in Section 2.3.3-b. Finally, the energy consumption of the value generation components and register file are reduced 1.3× and 1.1× respectively.

When keeping the MVL fixed, we find that increasing the number of lanes has only a minor effect on energy consumption. The VMRF increases its energy consumption due to the widening of its interface, i.e. each lane needs to write generated addresses in parallel. However, we find this increase is modest with respect to the total energy; in the worst case—$MVL = 64$ and $lanes = 4$—the VMRF uses 8% of the total energy. The energy consumption of functional units and result broadcast also increases. This is caused by multiple factors—firstly, quicksort relies a lot on the `compress` vector instruction; we implement this instruction using a crossbar and this uses more energy as the number of lanes increases. Secondly, there is an increase in the number of misspeculated instructions that occupy the functional units before being squashed. The other structures do not show a notable difference in energy consumption.

When keeping the number of lanes fixed, we find that moving from a MVL of eight to a MVL of sixteen results in further dynamic energy improvements. The front end components see further energy reduction due to each vector instruction now representing up to twice as much work. As shown in Appendix A, there is a significant decrease in both the number of instructions committed and the number of lines fetched from the instruction cache. The memory-related units also decrease in energy usage. The energy of the L1D cache and TLBs is halved due to the reduction of scalar code. Interestingly, the L2 cache energy is also reduced. This is for two reasons—(1) sixteen 32-bit elements use exactly one cache line, thus letting us load one complete line per vector memory request, and (2) after quicksort partitions its input, i.e. compresses, we can store more elements per cache line. We see from the statistics that there are fewer vector memory store requests that write only partial cache lines. The remaining structures also exhibit a reduction in energy consumption. This is principally due to a decrease in instructions needed due to the larger MVLs. Increasing the MVL past sixteen results in further energy reductions in all the components.

5.3.2 Bitonic Mergesort

Figure 5.3 shows the runtime dynamic energy consumption of bitonic mergesort. Note that the scale of its horizontal axis is different to the previous figure. The algorithm exhibits very different behaviour to the vectorised quicksort of Section 5.3.1. We see that the most dominant components are the register file, functional units and result broadcast. Bitonic mergesort exhibits relatively little memory activity since the majority of data movement occurs between vector registers.

Similar to the vectorised quicksort's behaviour, when increasing the MVL of the single-lane configurations we observe a decrease in energy in the front end and out-of-

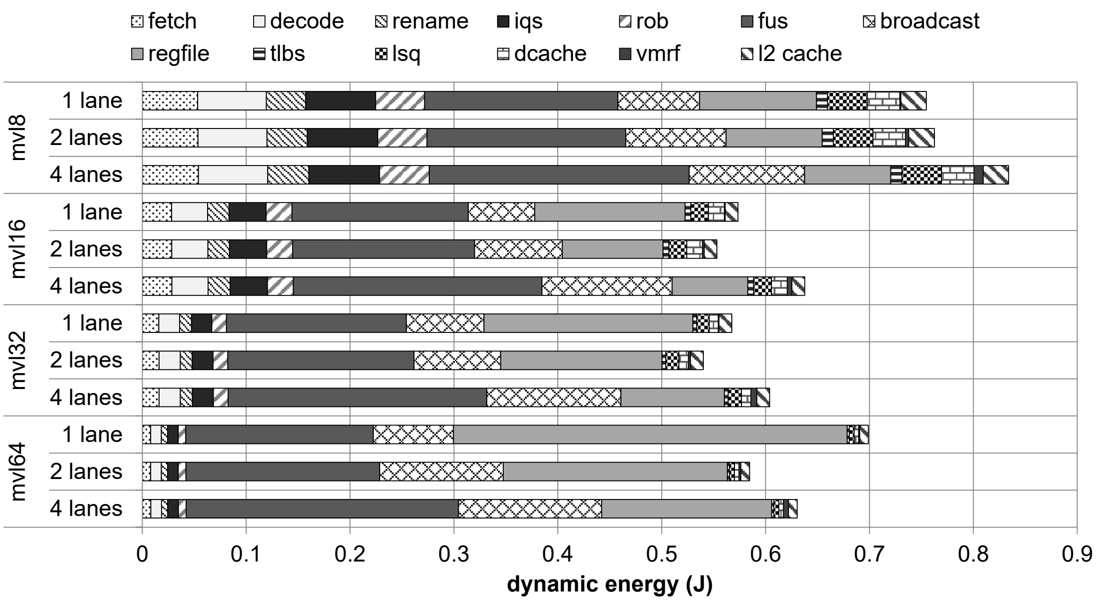


Figure 5.3: Dynamic energy breakdown of bitonic mergesort.

order engine. In contrast—due to the nature of sorting networks—the value generation components and the register file show a growth in energy usage. As the MVL is increased, so is the total number of operations. The reason we can achieve further speedups despite doing more work is because there is a simultaneous growth of the number of operations that can be executed in parallel. This is why lanes are so fundamental to this algorithm's performance. The register file energy consumption also grows with the MVL. This is principally due to an increase in energy per read/write as the structure grows but is also—in part—caused by the extra operations.

Increasing the number of lanes has an interesting effect on dynamic energy. We observe a major reduction in energy consumption of the register file unit. When using lanes, the register file is broken up into homogeneous independent banks that are each strictly coupled with a particular lane. This partitioned design consumes less energy accessing an element than a larger non-partitioned monolithic structure. Table 5.2 shows the per-bank energy consumption of the vector register file when $MVL = 64$.

Table 5.2: Per bank energy of vector register file for bitonic mergesort when $MVL = 64$.

	1 lane	2 lanes	4 lanes
bank 0	0.367 J	0.102 J	0.038 J
bank 1	-	0.102 J	0.038 J
bank 2	-	-	0.038 J
bank 4	-	-	0.038 J
total	0.367 J	0.204 J	0.151 J

In contrast, we find that the functional unit energy usage increases with the number of lanes. This is principally due to this algorithm's reliance on `shuffle` instructions. We have modelled `shuffle` and other permutative instructions using crossbars. The complexity—and consequently energy consumption—increases with the size of the crossbar. While this energy consumption is still modest with four lanes, a crossbar has quadratic complexity and we have observed that it becomes the dominant component when using more than four lanes. Although we do not expect a real `shuffle` implementation to use an unscalable structure such as a crossbar, our assumed model helps highlight the high complexity and large energy requirements of shuffle-based sorting algorithms.

5.3.3 Radix Sort

Figure 5.4 shows the dynamic energy usage of radix sort run with sixteen bins. The scale of its horizontal axis is different to the previous figures. These experiments use at most 40% of the energy required for bitonic mergesort. Radix sort—with an $O(k \cdot n)$ complexity—executes fewer instructions than quicksort and bitonic mergesort which both have an $O(n \cdot log_2 n)$ complexity.

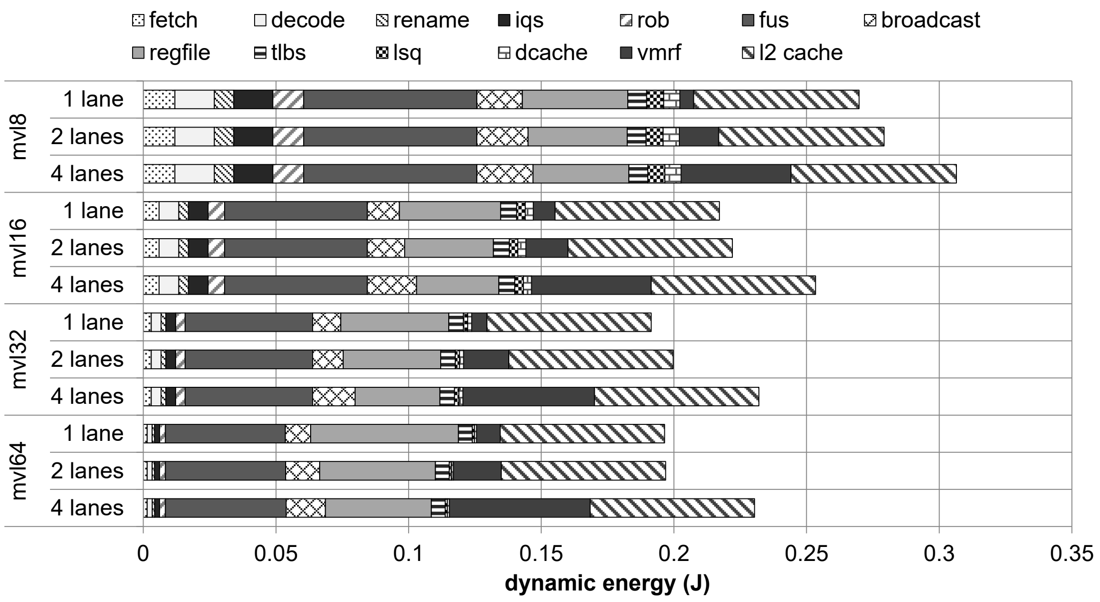


Figure 5.4: Dynamic energy breakdown of radix sort with sixteen bins.

This algorithm's dynamic energy consumption is dominated by the register file, value generation components and memory-related units. Its front end energy usage is small compared to the other algorithms. We observe that enlarging the MVL increases the register file energy usage slightly. Unlike bitonic mergesort—which increases the number of register files accesses as MVL increases—the number of accesses remains the same as the MVL grows. This extra energy consumption alone comes from the structure's increase in size which leads to a larger energy cost per access. In contrast, the functional units decrease their energy usage as the MVL increases, principally due to a reduction in bookkeeping scalar code achieved through the vector SIMD ISA.

The L2 cache activity forms a more significant portion of the overall energy consumption than any of the other algorithms. This is caused by the inefficient strided memory access pattern necessary for radix sort's stability. When loading the input to a vector register, each element is located in a different cache line; additionally, these elements are subsequently scattered to disjoint locations in memory. This is in contrast to quicksort and bitonic mergesort that both load and store their values with a more energy-friendly unit-stride pattern. We also see notable energy consumption caused by the VMRF structure. This is due to an increased number of cache lines being accessed in comparison to the other algorithms. This structure increases its dynamic energy usage as the number of lanes grows since the interface must be widened to write multiple cache line requests each cycle.

5.3.4 VSR Sort

In order to understand the runtime energy behaviour of VSR sort, we first evaluate it with sixteen histogram bins—the optimal number of bins for radix sort—shown in Figure 5.5 using the same scale as the previous figure. The energy consumption is always less than radix sort and the difference in energy between both algorithms grows more as the MVL and number of lanes increase.

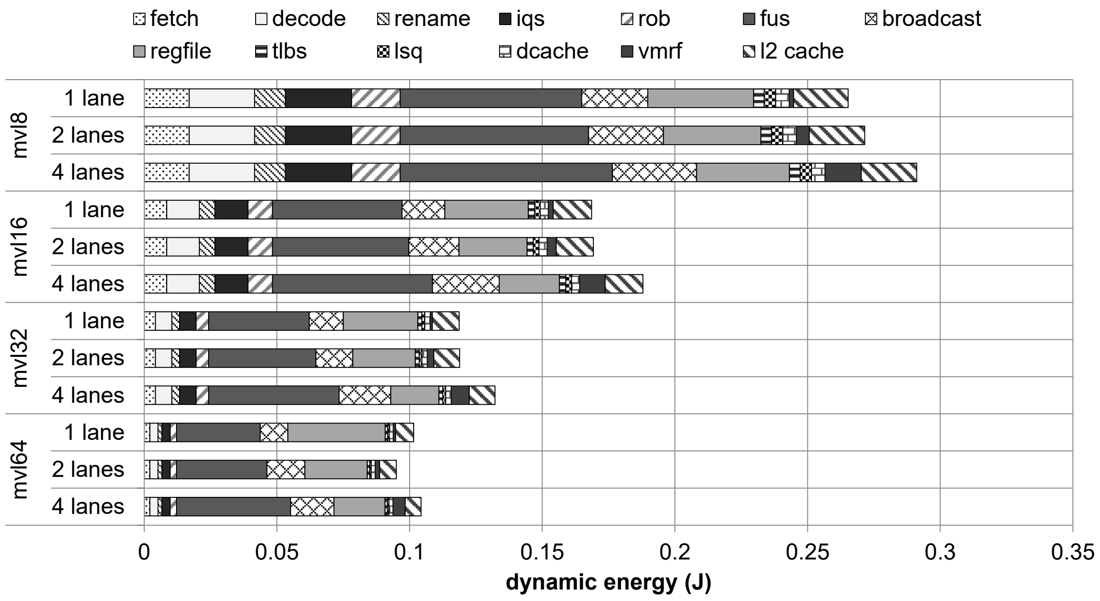


Figure 5.5: Dynamic energy breakdown of VSR sort with sixteen bins.

We find that the front end components and out-of-order engine use more energy than radix sort. This is caused by an increased number of instructions per pass in VSR sort over radix sort. While these structures form a significant percentage of the total dynamic energy when $MVL = 8$, this diminishes to something negligible as we increase the MVL. Functional unit energy consumption exhibits complex behaviour. The single-lane configurations of VSR sort show equivalent behaviour to—or lower consumption than—radix sort. Unlike radix sort, increasing the number of lanes *does* increase the energy consumption as a result of the crossbar and CAM structures which must be multiported—the latter with port conflict detection logic.

Despite the increase in instructions, the register file does not see an increase in energy. This is mainly due to a decrease in the L2 cache to vector register transfers caused by the change from a strided memory access pattern to a unit-stride one. Many values can now be written to a vector register in one access as a result of the improved spatial locality. For the same reason, we observe a significant energy reduction in the memory-related units—above all, the L2 cache and VMRF.

In Section 3.4.3, we found that we can increase the number of histograms bins from sixteen to 256 while retaining locality, allowing us to reduce the number of passes of VSR sort from eight to four. Figure 5.6 shows the dynamic energy consumption of VSR sort run with 256 bins using the same scale as the previous figure. These experiments consume at most 75% of the energy used in the sixteen-bin configurations of VSR sort.

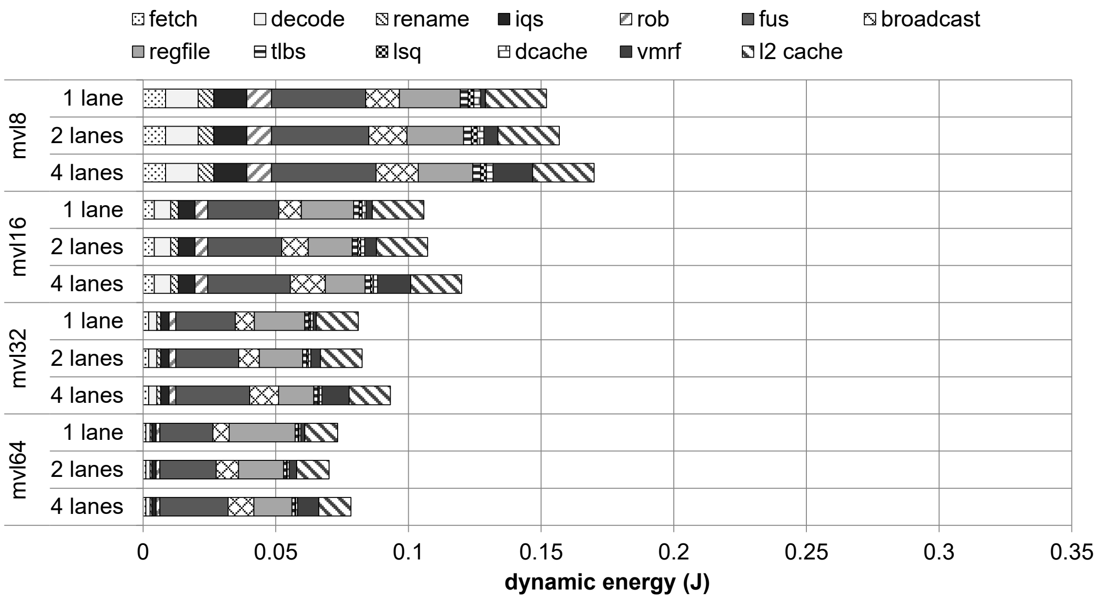


Figure 5.6: Dynamic energy breakdown of VSR sort with 256 bins.

Since the number of algorithm passes is halved, most of the hardware structures reduce their dynamic energy consumption. The only units that increase their energy consumption are the L2 cache and VMRF. Relative to sixteen-bin experiments, the L2 cache energy consumption increases between 1.1× and 2.0×. This growth in energy consumption is directly related to the increase in the number of histogram bins. With sixteen bins, a whole histogram fits in a single cache line. When the histogram is enlarged to 256 bins, sixteen cache lines are required and there is subsequently a higher number of cache accesses when updating the structure. For the same reason, the VMRF structure's energy consumption also goes up. Depending on the configuration, VSR sort exhibits a reduction in dynamic energy over the scalar baseline between 8.2× and 20.1×.

5.4 Power

Figure 5.7 shows the average power consumption for each sorting algorithm run under the various hardware configurations. The results have been normalised to the scalar baseline. Each bar is separated into gate leakage power, subthreshold leakage power and dynamic power. On the far left, the power consumption of the scalar baseline is shown—also marked with a horizontal line.

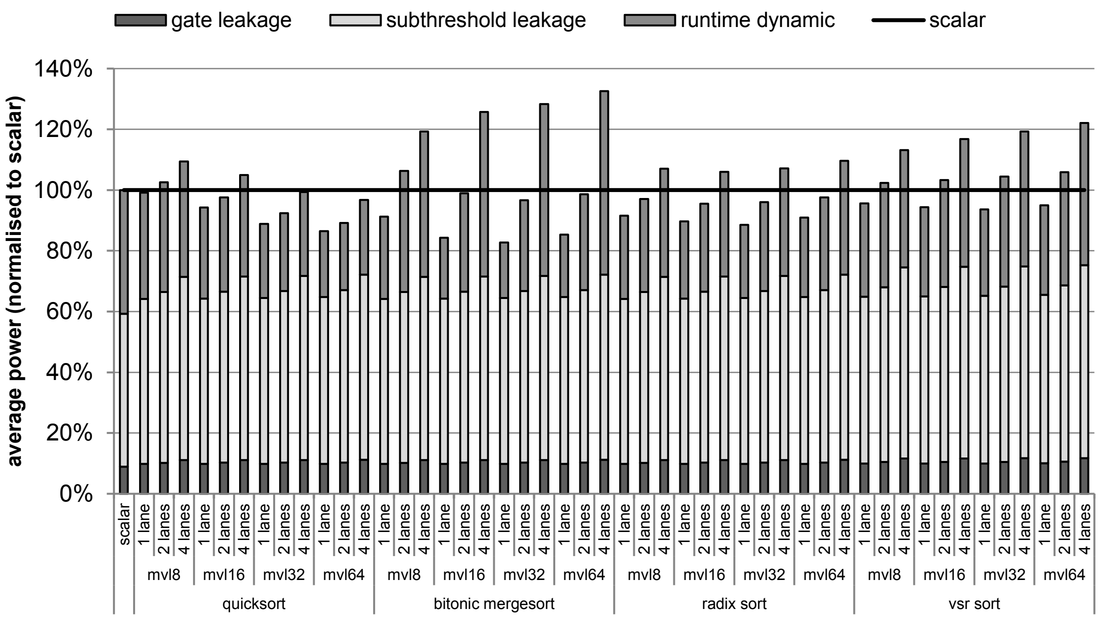


Figure 5.7: Normalised average power consumption of all sorting algorithms.

It can be seen clearly that—depending on the hardware configuration and algorithm—there are experiments that require less power, the same power and more power than the scalar baseline. We first comment on the results of the vectorised quicksort algorithm. When $MVL = 8$ and $lanes = 1$, the power is very close to that of the scalar baseline. Increasing the number of lanes increases both static and dynamic power. Increasing the MVL reduces power; although the static power increases, the dynamic power decreases more. This is because we obtain a substantial reduction in dynamic energy for quicksort when increasing the MVL (Figure 5.2) but not an equivalent decrease in execution time (Figure 3.3). This trend continues and we find the best performing configuration—$MVL = 64$ and $lanes = 4$—uses less power than the scalar baseline.

Bitonic mergesort follows the same pattern, however, we see a much larger increase in dynamic power as the number of lanes increases. This is due to the dramatic decreases in execution time obtained from using lanes in addition to the extra energy required to perform `shuffle` operations in parallel.

Both radix sort and VSR sort exhibit similar trends to each other and are both slightly different to quicksort and bitonic mergesort. Using a larger MVL increases power marginally rather than decreases it and implementing more lanes always increases the power. The single-lane configurations require less power than the scalar baseline whereas the configurations using four lanes always use more power.

VSR sort can achieve significant speedups using hardware configurations that consume less power than the scalar baseline and even better speedups if we have a larger power budget. When $MVL = 64$, the single-lane configuration uses 5% less power yet achieves a 13.5× speedup over the scalar baseline. A four-lane configuration with the same MVL uses 22% more power than the scalar baseline but achieves an impressive 20.6× speedup.

5.5 Related Work

In this section, we discuss work related to the energy efficiency of vector architectures.

Lemuet et al. [LSCJ06] design and evaluate vector extensions for an Alpha 21264. Similar to our findings, they observe that their vectorised kernels exhibit significant speedups and consume less energy than their scalar counterparts. Although less energy is used overall, it is consumed at a faster rate leading to a high power processor. Our work is significantly different to that of Lemeut et al. We make detailed sensitivity experiments varying the MVL and number of lanes, whereas they evaluate only two alternative configurations. Our work focuses on integer applications—specifically sorting—whereas theirs focuses on a selection of FP kernels. In particular, our aim has been to understand the energy/power differences *between* several vectorised sorting algorithms rather than just against a scalar baseline. Finally, our work attempts to push the state of the art of vector design by focusing on irregular DLP, i.e. novel vector instructions and hardware structures. We conclude that VSR sort and its hardware design is significantly more energy efficient than standard SIMD sorting algorithms. Additionally, we find that hardware configurations with a small number of lanes can offer significant performance benefits while actually using less power than the scalar baseline.

Lee et al. [LAB$^+$11] evaluate the relative performance/energy/area benefits of multi-threaded, vector SIMD and GPU SIMT architectures with a range of benchmarks. They conclude that for the majority of their benchmarks, a vector SIMD architecture achieves better performance, energy consumption and area than a multicore architecture with an equivalent parallel factor. They also conclude that a vector SIMD memory system achieves better performance with less energy over a typical SIMT memory system whereby µthreads individually make requests to memory which, if possible, are coalesced. Although they include one sorting benchmark—radix sort—it is not the principal focus of the article and does not appear to have the same optimal implementation as the radix sort used in our evaluation. Finally, Lee et al. do not provide detailed breakdowns of the energy consumption. In contrast, we do provide this because it provides insights into the efficiency of the algorithm and hardware configurations alike. This helps to further motivate our proposed VSR sort algorithm since we can illustrate with detail exactly where the energy savings come from.

5.6 Conclusions

We reach three principal conclusions from our energy and power evaluation. (1) Our vector extensions have a small overhead and consume less energy than the scalar baseline architecture when running vectorised algorithms. (2) VSR sort uses less dynamic energy than any other sorting algorithm evaluated in this work and also outperforms all of them. (3) For VSR sort, the single-lane configurations consume less power than the scalar baseline while still yielding significant performance speedups. Multiple lanes can be used for additional performance but this ends up consuming more power than the scalar baseline. This choice could be useful when deciding the different configuration, e.g. high-performance or low power, while retaining the same ISA and binaries.

CHAPTER 6

Thesis Conclusions

In this thesis we have explored the applicability of vector architectures as a means to accelerate data management. We now summarise and reflect on our research, contributions and results.

6.1 Summary of Achievements

In Section 1.2, we outlined a series of objectives for this thesis. In a nutshell, we intended to (1)—define a concise and optimal ISA in order to vectorise representative operators found in a DBMS; (2)—design efficient hardware extensions that implement this ISA with low area, energy and power overhead costs; and (3)—to effectively transform DBMS operators using these vector extensions—including designing novel algorithms and new instructions to achieve this. We believe that these objectives have been fulfilled by the following achievements of this thesis.

In Chapter 2, we developed a sophisticated integer-based vector ISA and microarchitecture suited for data management. Using our own custom simulation framework, we performed extensive analysis on a hash join algorithm taken from a commercial DSS DBMS which we vectorised using the proposed instruction set. We evaluated various parameters, e.g. the impact of the available memory bandwidth and software prefetching on performance. Furthermore, we proposed relaxing the memory model of the architecture which allowed vector memory instructions to issue earlier without memory alias checks done in hardware. This turned out to be very beneficial with good performance returns. We demonstrated good scalability with an increasing MVL and speedups between 1.94× and 4.56× over the scalar baseline without needing lockstepped parallel lanes.

In Chapter 3, we performed extensive analysis on three diverse sorting algorithms and assessed them using consistent metrics. We learned that all of the algorithms suffer from bottlenecks and scalability problems due to the irregularity of the DLP and the limitations of a standard vector SIMD instruction set. Based on these findings, we proposed VSR sort—a novel way to efficiently vectorise radix sort. To enable this algorithm in our vector architecture, we defined two new instructions—`VPI` and `VLU`. We provided a suitable hardware implementation of these instructions which includes both serial and parallel variants. We demonstrated that the algorithm scales well when increasing the MVL, and works well both with and without parallel lockstepped lanes. VSR sort showed maximum speedups over a scalar baseline between 7.9× and 11.7× when a simple single-lane pipelined vector approach was used, and maximum speedups between 14.9× and 20.6× when as few as four parallel lanes were used. This is contrast to the next best vectorised sorting algorithm—radix sort—which attained maximum speedups between 3.6× and 5.2×, even when lanes were used.

In Chapter 4, we looked at vector instructions as a means to accelerate data aggregation. We found that this is not a trivial task due to the irregularity of the DLP. We made detailed evaluations using multiple algorithms which either evade this irregularity, or confront it through the use of novel vector instructions. We observed that the evasion techniques have limited applicability unless the input is presorted, otherwise, the confrontation techniques prove to be more advantageous. We discovered that `VPI` and `VLU`—along with VSR sort—can aid data aggregation, especially with the realisation that the input need not be fully sorted. With minimal modifications, we extended the logic used for `VPI` and `VLU` in order to further accommodate data aggregation by defining a set of new instructions called `VGAx`. We found that the best algorithm depends highly on both the distribution and cardinality of the input. In most cases, this can be easily detected at runtime and be used to make a choice dynamically. Using a combination of these techniques, we achieved speedups between 2.7× and 7.6× over a scalar baseline for a MVL of 64 and four lockstepped lanes.

In Chapter 5, we assessed the area, energy and power of our proposed vector extensions. With respect to area, we found that the overhead of adding vector support was between 9% and 19%—a modest range considering the performance improvements gained. We used the sorting algorithms from Chapter 3 as a case study to evaluate energy and power. In every instance, the vectorised sorting algorithms consumed less energy than the scalar baseline. A lot of these energy savings came from the decrease in the number of instructions which led to reduced activity in many hardware structures. In the case of our own VSR sort, it used up to 20.1× less dynamic energy than the scalar baseline while still yielding significant performance speedups. We also found that power is dependent on both the algorithm and the hardware configuration. We observed that all of the single-lane vector configurations consumed less power than the scalar baseline. Multiple lanes could be used for additional performance but at the expensive of higher power. We found that for VSR sort run on a hardware configuration with a MVL of 64 and four lockstepped lanes, the power consumption was 22% more than the scalar baseline but achieved a significant 20.6× speedup.

6.2 Relegated Ideas

While this thesis has had many successful outcomes, several ideas were explored that did not produce as significant results. In this section, we outline three of these.

When we first began looking at vectorised sorting algorithms, we formulated an idea for an efficient vectorised implementation of quicksort. We had noted that the implementation proposed by Stone [Sto78] had a deficiency in that it relied heavily on an auxiliary array for temporary results. This means that—on average—every element will be copied to and from the auxiliary array $log_2 n$ times. This copying was often superfluous since a large number of elements were already positioned correctly relative to the pivot of that iteration. We devised our own vectorised algorithm based on an scalar in-place variant of the algorithm. It used two pointers—one at the beginning of the array and another at the end. Using the first pointer, the input was scanned forward until a value larger than the pivot was found. Likewise, the input was scanned in reverse order using the second pointer until a value less than or equal to the pivot was found. These elements were then switched and the process repeated until the two pointers collided. The algorithm was cumbersome to vectorise and required many permutative instructions such as `elemental shift` and `reverse`. In the end, the performance only outperformed Stone's quicksort by a few percent. These performance results didn't merit a detailed explanation of the algorithm and consequently we opted to cite, explain and evaluate Stone's algorithm instead.

In another venture into vectorised sorting algorithms, we designed an algorithm to vectorise a sorting network while avoiding some of the identified deficiencies. As mention in Section 3.3.2, we noted that the depth of a sorting network, i.e. the number of operations, depends on the size of its input which in our case is the vector length. This is the reason why we observed a rapid increase in execution time of bitonic mergesort when increasing the MVL as seen in Figure 3.5. We formulated an alternative approach whereby we emulate a shorter VL in order to use bitonic merging network with a small depth. The trick was to have many disjoint networks within one larger vector, e.g. to have sixteen independent merging networks, each working on four elements, all contained in a vector with a MVL of 64. In order to implement this, we had to define a set of new instructions which we categorised as 'multistream'. For example, a `multistream load` might read four consecutive elements from memory, apply an offset, read the next four consecutive elements and so forth. We encountered similar memory patterns in the works of [CVE99] and [CEL$^+$03], however, the semantics of our idea were significantly more complicated—especially when paired with masks. Furthermore, the permutative instruction `reverse` also needed its own multistream variant. We managed to design and implement a simple working example, however, we quickly realised that there were many corner cases which created a very complicated control flow. The algorithm became increasingly difficult to implement correctly and the idea was eventually shelved.

In our data aggregation work, we also explored an alternative idea not mentioned in Chapter 4. While it was clear from our initial experiments that irregular DLP was the culprit behind the poor vector performance, we originally experimented with a different idea to `VGAx`. We had primarily put our focus on the sorted reduce methods—in particular, ***advanced sorted reduce*** which leverages VSR sort. We observed that when vector reductions are applied to datasets with *high* cardinalities, the average vector lengths were too short and caused poor performance due to serialisation. As a solution, we proposed two instructions—`intradelta` and `multireduce`. `intradelta` was a comparison instruction with one vector input; it compared adjacent elements within a vector register and produced a mask to mark differences. `multireduce` behaved like a typical reduction instruction except that—when used in conjunction with a mask—could reduce several independent segments of the input vector which were marked by bits set in the associated mask. This way, one vector instruction could represent up to MVL separate reduction operations and the average vector length was not compromised. A similar concept was explored in the work of Chatterjee et al. [CBZ90], however, our proposed instructions were significantly more intricate. We modified ***advanced sorted reduce*** to use these new instructions and achieved better performance when aggregating the *high* cardinality datasets. Conversely, we found that when aggregating datasets with cardinalities other than *high*, the modified algorithm actually performed worse than the original. We eventually discovered that ***partially sorted monotable***—using the novel `VGAx` instructions—outperformed this segmented reduction method in all cases so we ultimately scrapped the idea of using `intradelta` and `multireduce`.

6.3 Behind the Scenes

In this section, we reflect on some of the decisions and outcomes of the thesis as a whole.

In Chapter 2, we introduced our idea of relaxing the memory model of the baseline architecture. This allowed the vector memory instructions to issue out of order without hardware checks. We proposed the use of fence instructions to describe dependencies between memory instructions. We compared this technique against a stricter mechanism in which vector memory instructions maintain their relative order and we concluded that the out-of-order mechanism had enough of an advantage to justify the elaborate programming model. The hash join algorithm presented a very clear vector transformation and the placement of the memory fences was quite straightforward. When we moved beyond hash join into other algorithms, we found that this programming model became more challenging and error prone. Even though preserving this programming model presented challenges in later chapters, we opted to maintain it as it gave significant performance advantages.

In Section 2.5.1-b, we reported that there was very little difference between the ooo-customfence-l2cache fence implementation—an intricate mechanism in which the commit logic has a direct path to a fence issue queue—and a simpler alternative—flushed-fence—whereby fence instructions flush the processor's pipeline. Although we did not comment on it, in later chapters we found that the ooo-customfence-l2cache mechanism outperformed the flushed-fence method with significantly more difference than what was measured in the hash join algorithm.

While the relaxed memory model certainly gave performance benefits, in retrospect it seems we could have achieved something similar using a cleaner programming model. One year after our MICRO publication—the basis for Chapter 2—NEC described a solution similar to this for its SX-ACE vector supercomputer [MHIT14]. Although they used hardware to detect memory aliases between non-indexed memory instructions, they agreed with our conclusions that extending this hardware to support gather/scatter aliasing would be complicated. They introduced a 'vector overtake' flag for their vector store instructions which allows younger loads to execute earlier. This way, vector memory instructions will execute in order unless specifically granted permission to do otherwise. Although we haven't attempted to vectorise the data management kernels using such a model, it does seem a little more elegant than our fence mechanism. The upside to this is that the presence of such a mechanism in a vector supercomputer would suggest that the independence between vector memory instructions is a common occurring pattern and has an applicability outside data management. We feel that this is an area that deserves more exploration.

In Chapter 3, we made a study on vector SIMD sorting and—based on the characteristics and performance of the studied algorithms—we proposed VSR sort. While we wrote our HPCA article—which served as the basis for Chapter 3—to make it appear that VSR sort was the logical step forward to solve the deficiencies in other algorithms, the reality was much less tidy. The basic idea for VSR sort came after reading about a method to perform load locked-store conditional operations within a single indexed vector memory instruction [KKS+08]. We understood that this could reduce the size of bookkeeping structures in radix sort but, in that moment, we didn't fully appreciate how much impact this could make on performance. We left this idea dormant for quite some time.

As mentioned in Section 6.2, we explored several other directions first. The results of our own proposals didn't look promising and we were clutching straws by persisting with the topic of sorting. As a last resort, we decided to hack together a proof of concept of radix sort using a single non-replicated histogram. We soon realised that implementing load locked-store conditional in our simulation framework would be a massive undertaking. We began searching for alternative ways of achieving the same effect without touching the memory system and this is how `VPI` and `VLU` were conceived. At that time, it didn't even occur to us that we would need to transform the memory access pattern from a strided one to a unit stride one and this only became apparent after the fact. We did not anticipate how well this algorithm would ultimately perform. The discovery and success of the algorithm and instructions fed into the contents of the succeeding chapters. It was at this point that we started exploring the potential of exploiting irregular DLP through novel instructions and this explain why there is some disparity in style between Chapter 2 and the remainder of the thesis. We realise now that this type of instruction has a lot of potential and we feel that the topic of irregular DLP could become a significant research direction in the future.

Not all of our design decisions can be considered successful. In particular, our decision to allow partially overwriting architectural vector registers was short of a disaster. To expand on this point—if, for example, there is an operation such as $\vec{v1} \leftarrow \vec{v2} + \vec{v3}$ but the VL is set to a value less than the MVL, our ISA guarantees that the previous values of the final $MVL - VL$ elements of $\vec{v1}$ will remain intact. While this may seem like an intuitive design choice from the programmer's perspective, it actually caused several problems in the microarchitecural implementation. It forced us to use an extra operand in the structure of the vector μops which led to larger issue queues. While we observed in Chapter 5 that this wasn't detrimental to the overall energy consumption, its biggest hindrance was to the out-of-order scheduling.

We observed this problem after our work on hash join had already been published. We were benchmarking a SAXPY kernel but the timing results weren't matching our pen-and-paper calculations. Over several days we made a deep dive investigation into all the nooks and crannies of the microarchitecture using very detailed logfiles from PTLsim. We anticipated that since the body of the SAXPY loop is small, the processor should be able to hoist and schedule the loads of the next SAXPY iteration *before* the multiply, add and store have completed. Instead we found that these instructions were not being issued until *after* the arithmetic part had finished. These stalls were caused—in part—by our coding style. We used the same architectural registers for the destination of the vector loads and the destination of arithmetic instructions. Internally, the microarchitecture was putting a physical register dependency between the current iteration's vector load and the previous iteration's vector multiply which serialised everything.

We devised a solution to fix some of these cases without having to rewrite the code of all the benchmarks. Its logic is as follows—when a vector instruction is dispatched to an issue queue, the processor dynamically removes this self-dependency if the VL equals the MVL and masking is not used. This way, it is guaranteed that the entire architectural register will be overwritten with no remnant elements. While this fixed many of the problematic occurrences, some cases still remained. For example, it's not always possible to know the value of the VL register at dispatch time since the `set VL` instructions are often dispatched on the same cycle as dependent vector instructions. This made short stripmined loops more difficult to code efficiently.

The worst part of this ISA feature is the fact that we rarely made use of it. There were only a few cases in our evaluated algorithms where we actually need a vector register's old values and these instances could probably work just as well using `merge` instructions. We conclude that this design decision was not optimal and in hindsight we should have designed the ISA differently.

As a final point, it is worth mentioning the overlap between our proposals and Intel's SIMD extensions. When we began this project as a master's thesis in 2010, the difference between multimedia extensions and a vector ISA was less ambiguous. We could clearly point out deficiencies in technologies like SSE and use these to motivate the adoption of a more sophisticated vector-like ISA. Although there were already dedicated accelerators like FPGAs, CUDA-enabled GPUs and Larrabee, these were different enough to the idea of specialised vector extensions for a microprocessor; we could, therefore, continue motivating our own proposals.

The following year, Intel released the Advanced Vector Extensions (AVX) as an upgrade to their SSE line. The name alone was worrisome since they now explicitly used the word 'vector'. Worst of all was that—with the release of this product—they started a trend of widening their SIMD register width. The last time Intel had done this was in 1999 with the introduction of SSE. These SIMD extensions expanded the 64-bit capabilities of MMX with 128-bit wide SIMD registers. For AVX, they increased the width to 256 bits. An increasing SIMD width—or *MVL* to use the terminology of this thesis—took away one of the more distinguishing features between vector and multimedia extensions. This was exacerbated even more later with the introduction AVX-512 which increased the SIMD register width to 512 bits with previsions to extend this further to 1,024 bits in the future.

The widths of the SIMD registers were not the only concern. With respect to SSE, we could describe quite clearly the functionality that was missing. For example, there was no support for indexed memory instructions, a lack of a variable vector length parameter and very limited support for masks and permutative instructions. Over the course of this thesis, most of these limitations were addressed by Intel. They introduced partial support for indexed memory instructions in AVX2 and full support in AVX-512. With AVX, they introduced a set of permute instructions that allowed the permutation pattern to be variable; before, their shuffle implementation required the pattern to be an immediate operand. In AVX-512, they introduced a set of registers to support masked operations. Each year, Intel's SIMD support edged away from their original MMX multimedia extensions and progressed towards a vector ISA.

The most frustrating instance of this relates to our work on sorting. In the spring of 2014, our article on VSR sort was already written and polished. In the related work section we cited and discussed both Intel's atomic vector instructions and an earlier proposal from Stanford called Scatter-Add. Neither of these solutions resembled our proposal for `VPI` and `VLU` so we remained confident that this work would be seen as novel. One week before submitting the article for peer review, we came across—for the first time—Intel's proposal for AVX-512 CDI conflict detection instructions. Although semantically different to our own proposal, the CDI instructions attempted to tackle a similar problem, i.e. resolving conflicts within vector scatters without interfering with the memory system. Although we didn't find a way to write VSR sort using the CDI instructions, it may very well be possible with a bit of tweaking. Learning about these instructions so late was a large oversight on our part and it ultimately took away from the novelty of the resulting article.

From the point of view of publishing articles, these moves by Intel were problematic, however, they simultaneously helped reaffirm the importance of this topic which was beneficial. When cynical reviewers pointed to GPGPUs as the alpha and omega of DLP-accelerators, we were able to mention that the industry still had a strong interest in enabling advanced SIMD support within the cores themselves. This, in a sense, made publishing somewhat easier.

6.4 Future Outlooks

In this section, we comment on possible future directions for this research topic.

In recent years, we have seen the rise of the multicore microprocessor which now dominates most markets. As mentioned in Chapter 1, we do not believe vectors and multicore to be mutually exclusive; both could be used in tandem to gain more application performance. What is less obvious is how the techniques proposed in this thesis specifically could be incorporated into a multicore architecture. Dealing with a relaxed vector memory model might become considerably more complex when memory consistency and cache coherence are brought into the equation. Furthermore, while we demonstrated in Chapter 2 that a single scalar core cannot easily saturate a system's entire memory bandwidth, the same is probably not true when considering multiple cores. Since vectors require a large sustained memory bandwidth, it could be problematic when multiple cores—each with vector extensions—start competing for the same bandwidth. This leaves an open question as to how well the algorithms proposed in this work will scale using multiple cores in addition to vectors.

This thesis has focused on the performance of vector extensions when integrated into a scalar microarchitecture. As such, our point of reference and comparisons are generally to the same scalar microarchitecture without these additions. What remains to be seen is how our extensions compare against dedicated accelerators such as FPGAs. This is not obvious to estimate. While FPGAs may be able to offer performance advantages over our vector extensions, they still lie off chip, therefore, the processor will have setup and data movement overheads. Our vector extensions are tightly integrated in the processor's pipeline to be able to efficiently handle an intermix of scalar and SIMD code. Furthermore, we have demonstrated that the area overhead needed to implement these extensions is modest—the same may not be true for a dedicated accelerator. Open work includes comparing these extensions with alternative solutions and evaluating the performance, energy and area advantages of each technology.

Computer architecture and data management are both rapidly evolving areas. There are many new and interesting ideas happening in both topics and this may present new opportunities in the future to combine vectors and data management once again. One interesting development that occurred in the lifetime of this thesis is the availability of high bandwidth memory implemented through 3D stacking [LGBT05, JK12, Sta13]. We concluded in Chapter 2 that our vector extensions can easily saturate a system's available memory bandwidth and that adding more bandwidth is generally beneficial. 3D-stacked memory offers the possibility to provide our vector extensions with substantially more bandwidth over what was possible using older DRAM technologies.

While this work has focused on typical operations found in relational database systems, we notice that the landscape of data management is changing. We see that alternative database technologies—often called NoSQL—are garnering more attention these days as a means to service the growing amount of diverse data online which does not fit neatly into the relational model [Cat11]. These include, among others, graph databases, e.g. Neo4j; document stores, e.g. MongoDB; object databases, e.g. db4o and key-value stores, e.g. Redis. The semi-structured nature of these databases might suggest that vector instructions oriented towards irregular DLP could be a good match.

CHAPTER 7

Publications

The work of this thesis has resulted in the following publications.

International Conferences and Journals

[1] T. Hayes, O. Palomar, O. Unsal, A. Cristal and M. Valero, 2012, December. Vector Extensions for Decision Support DBMS Acceleration. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture* (**MICRO**) (pp. 166-176).

[2] T. Hayes, O. Palomar, O. Unsal, A. Cristal and M. Valero, 2015, February. VSR Sort: A Novel Vectorised Sorting Algorithm & Architecture Extensions for Future Microprocessors. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture* (**HPCA**) (pp. 26-38).

[3] T. Hayes, O. Palomar, O. Unsal, A. Cristal and M. Valero, 2016, June. Future Vector Microprocessor Extensions for Data Aggregations. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture* (**ISCA**).

[4] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, E. Ayguade and M. Valero. Runtime-Aware Architectures. (2015). In *Proceedings of the 21st International European Conference on Parallel and Distributed Computing* (**Euro-Par**) (pp. 16-27).

[5] O. Arcas-Abella, A. Armejach, T. Hayes, G. A. Malazgirt, O. Palomar, B. Salami and N. Sonmez. (2016). Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory. *Computing in Science and Engineering, 18(1)* (**CiSE**), 80-87.

Patent Applications

[6] T. Hayes, O. Palomar, O. Unsal, A. Cristal and M. Valero. Methods and devices for discovering multiple instances of recurring values within a vector with an application to sorting. *European Patent PCT/EP2015/052394.* Filed February 5, 2015

APPENDIX A

Algorithm Runtime Characteristics

In this appendix, we provide extra runtime characteristics for the various algorithms evaluated in this thesis. A complete list of all of PTLsim's performance counters for every experiment from previous chapters would require excessive space. Instead, we choose a pertinent selection of performance statistics for a subset of algorithms, datasets and hardware configurations. In order to fit one table of data per page, we choose only the most relevant statistics and opt not to repeat any information that can be found in the original experiments, e.g. execution cycles.

Each table also contains a summarised dynamic instruction mix list with eight entries. Each entry provides an instruction category, e.g. add/sub and a percentage of its presence in the overall µop instruction mix. Unless a category's name is prefixed with 'vec', it refers only to scalar instructions, e.g. load refers specifically to scalar load instructions whereas vec load refers to vector loads. The meaning of most category names should be obvious with the possible exception of lea add/sub. This refers to a three-operand add/sub operation where the third operand is first shifted before being added to the sum of the other two operands. This is typically used for address generation, in particular for x86-64 load-effective address (LEA) instructions.

The vector statistics shown are for single-lane configurations only. While it's true that increasing the number of lanes can impact the interaction between the instructions and the processor's pipeline, the majority of the listed statistics should not change drastically. In some cases, there may be more cache misses due to a decrease in cycles between a prefetch and the subsequent corresponding load. The listings which feature loads/stores (either scalar or vector) exclude prefetch instructions.

For hash join, we provide a separate table for each of the datasets evaluated in Chapter 2. Each table contains statistics for the scalar baseline algorithm in addition to the vectorised implementation run on configurations with a MVL between eight and 64 elements. For sorting, we focus on the large dataset and provide a separate table for each of the vectorised algorithms. Each table also contains the scalar baseline for reference. For aggregations, there is a separate table for each algorithm evaluated with one particular cardinality. For brevity, we choose only two cardinalities—$c = 152$ (*low*) and $c = 625,000$ (*high*). Each table contains the results of all five datasets and the vectorised algorithms are evaluated with a MVL of 64.

The following experiments are run with their optimal configurations and prefetching enabled when useful. Both the scalar and vector versions have been compiled with ICC v14 using the best measured optimisation level.

Table A.1: Hash join run with the **l1r** dataset.

	scalar	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,212,622,847	467,238,909	261,033,270	156,880,169	104,902,359
of which are vector	-	119,492,877	60,971,632	31,688,297	17,165,819
scalar μops	1,784,852,765	419,580,258	242,667,192	153,008,721	108,163,116
insn line fetches -l1	1,062,457,847	153,871,385	94,643,552	59,772,587	41,534,089
-l2	0	1	1	1	1
-mem	38	78	78	78	78
branch predictions -correct	220,881,004	53,937,374	31,157,206	19,657,880	13,912,966
-incorrect	19,454,248	597,713	682,069	485,248	346,314
scalar store/load ratio	0.43	0.22	0.28	0.35	0.45
scalar loads -l1	375,882,227	49,622,495	27,959,936	16,283,267	10,427,027
-l2	21,111,418	3	2	4	6
-mem	9,468,439	52	45	52	56
vector loads -l2	-	86,279,574	76,913,652	70,305,800	63,563,054
-mem		1,706,577	1,531,703	1,467,551	1,380,564
average vector length (stdev)	-	7.67 (1.19)	15.18 (2.69)	29.77 (6.40)	56.90 (16.43)
vector load insns -unit stride	-	19,043,620	9,605,057	4,876,584	2,521,205
-strided		0	0	0	0
-indexed		20,222,951	10,301,752	5,306,429	2,849,578
vector store insns -unit stride	-	16,166,425	8,136,528	4,108,660	2,098,161
-strided		0	0	0	0
-indexed		3,433,269	1,757,718	919,954	508,871
instruction mix	add/sub 25%	add/sub 26%	add/sub 28%	add/sub 31%	add/sub 34%
	lea add/sub 23%	logical 19%	logical 18%	logical 18%	logical 17%
	load 18%	cond. branch 9%	cond. branch 10%	cond. branch 10%	cond. branch 10%
	cond. branch 10%	load 9%	load 9%	load 8%	load 8%
	logical 9%	vec load 7%	vec load 7%	vec load 6%	prefetch 5%
	store 8%	lea add/sub 6%	lea add/sub 6%	lea add/sub 5%	vec load 4%
	prefetch 2%	vec store 4%	vec store 3%	prefetch 3%	lea add/sub 4%
	other 4%	other 20%	other 20%	other 20%	other 18%

Table A.2: Hash join run with the **l2r** dataset.

	scalar	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,206,135,515	465,879,331	260,962,333	157,352,111	105,953,170
of which are vector	-	118,862,962	60,766,611	31,651,833	17,323,340
scalar μops	1,774,616,558	418,743,518	242,906,837	153,710,795	109,338,104
insn line fetches -l1	1,057,590,617	153,104,239	94,081,250	59,625,116	41,679,491
-l2	0	1	1	1	1
-mem	38	78	78	78	78
branch predictions -correct	221,696,316	53,779,900	31,129,088	19,672,454	13,997,404
-incorrect	19,357,712	598,315	686,910	482,904	342,004
scalar store/load ratio	0.43	0.22	0.28	0.36	0.46
scalar loads -l1	340,007,553	49,410,482	28,007,326	16,363,058	10,621,599
-l2	53,439,959	5	3	1	2
-mem	13,120,597	59	65	64	72
vector loads -l2	-	85,970,732	78,104,625	74,209,431	71,684,170
-mem		2,913,252	3,010,969	3,273,415	3,313,969
average vector length (stdev)	-	7.67 (1.21)	15.15 (2.77)	29.66 (6.58)	56.16 (17.53)
vector load insns -unit stride	-	18,976,682	9,576,663	4,867,136	2,531,809
-strided		0	0	0	0
-indexed		20,039,351	10,227,589	5,282,126	2,883,526
vector store insns -unit stride	-	16,131,159	8,123,281	4,104,017	2,100,369
-strided		0	0	0	0
-indexed		3,383,934	1,739,290	913,516	515,644
instruction mix	add/sub 25%	add/sub 26%	add/sub 28%	add/sub 31%	add/sub 34%
	lea add/sub 23%	logical 19%	logical 18%	logical 18%	logical 17%
	load 18%	cond. branch 9%	cond. branch 9%	cond. branch 10%	cond. branch 10%
	cond. branch 10%	load 9%	load 9%	load 8%	load 8%
	logical 9%	vec load 7%	vec load 7%	vec load 5%	prefetch 5%
	store 8%	lea add/sub 6%	lea add/sub 6%	lea add/sub 5%	vec load 4%
	prefetch 2%	vec store 4%	vec store 3%	prefetch 3%	store 4%
	other 4%	other 20%	other 20%	other 20%	other 18%

Table A.3: Hash join run with the **2mb** dataset.

	scalar	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,210,663,951	468,478,267	262,840,092	158,969,821	107,365,058
of which are vector	-	119,487,120	61,146,655	31,954,247	17,555,752
scalar μops	1,781,605,790	421,297,756	244,861,193	155,435,280	110,894,758
insn line fetches -l1	1,082,732,324	153,252,219	93,596,604	59,123,359	41,382,619
-l2	0	1	1	1	1
-mem	38	78	78	78	78
branch predictions -correct	230,500,190	54,056,854	31,305,029	19,818,963	14,140,149
-incorrect	20,781,694	617,204	731,728	490,702	338,529
scalar store/load ratio	0.43	0.23	0.28	0.36	0.46
scalar loads -l1	334,589,423	49,925,598	28,487,895	16,665,861	10,857,189
-l2	51,553,493	5	4	6	3
-mem	35,518,443	66	59	65	68
vector loads -l2	-	69,590,044	59,756,121	54,057,802	48,295,065
-mem		20,248,273	22,553,767	25,087,165	29,213,067
average vector length (stdev)	-	7.66 (1.22)	15.13 (2.80)	29.55 (6.82)	55.81 (17.91)
vector load insns -unit stride	-	19,032,867	9,611,204	4,892,551	2,548,871
-strided		0	0	0	0
-indexed		20,189,444	10,314,366	5,347,833	2,927,362
vector store insns -unit stride	-	16,162,258	8,139,580	4,115,926	2,110,095
-strided		0	0	0	0
-indexed		3,424,678	1,762,400	931,009	528,264
instruction mix	add/sub 25%	add/sub 26%	add/sub 28%	add/sub 31%	add/sub 34%
	lea add/sub 23%	logical 19%	logical 19%	logical 18%	logical 18%
	load 18%	cond. branch 9%	cond. branch 9%	cond. branch 10%	cond. branch 10%
	cond. branch 10%	load 9%	load 9%	load 8%	load 8%
	logical 9%	vec load 7%	vec load 7%	vec load 5%	prefetch 5%
	store 8%	lea add/sub 6%	lea add/sub 6%	lea add/sub 5%	vec load 4%
	prefetch 2%	vec store 4%	vec store 3%	prefetch 3%	store 4%
	other 4%	other 20%	other 20%	other 20%	other 18%

Table A.4: Hash join run with the **huge** dataset.

	scalar	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,238,261,317	477,518,628	267,518,219	161,659,033	108,559,012
of which are vector	-	122,551,259	62,702,574	32,820,508	17,902,804
scalar μops	1,824,782,121	428,755,252	248,771,223	157,725,257	111,987,753
insn line fetches -l1	1,103,050,103	156,410,647	94,422,757	60,059,154	41,530,787
-l2	0	1	1	1	1
-mem	38	78	78	78	78
branch predictions -correct	235,296,140	55,063,831	31,874,791	20,132,074	14,270,005
-incorrect	21,119,587	650,617	699,547	512,816	338,511
scalar store/load ratio	0.43	0.23	0.28	0.36	0.46
scalar loads -l1	343,174,792	51,547,236	29,059,526	17,190,490	11,034,088
-l2	52,261,846	3	2	3	3
-mem	40,700,424	66	62	52	55
vector loads -l2	-	67,887,520	57,766,212	50,947,246	44,157,471
-mem		25,470,149	27,886,444	31,518,277	36,633,474
average vector length (stdev)	-	7.67 (1.19)	15.14 (2.75)	29.52 (6.86)	56.18 (17.23)
vector load insns -unit stride	-	19,333,840	9,767,445	4,975,298	2,578,002
-strided		0	0	0	0
-indexed		21,015,444	10,734,966	5,583,767	3,008,104
vector store insns -unit stride	-	16,317,658	8,214,920	4,156,522	2,131,855
-strided		0	0	0	0
-indexed		3,647,968	1,875,467	992,931	554,531
instruction mix	add/sub 25%	add/sub 26%	add/sub 28%	add/sub 31%	add/sub 34%
	lea add/sub 23%	logical 19%	logical 18%	logical 18%	logical 18%
	load 18%	cond. branch 9%	cond. branch 9%	cond. branch 10%	cond. branch 10%
	cond. branch 10%	load 9%	load 9%	load 9%	load 8%
	logical 9%	vec load 7%	vec load 7%	vec load 6%	prefetch 5%
	store 8%	lea add/sub 6%	lea add/sub 6%	lea add/sub 5%	vec load 4%
	prefetch 2%	vec store 4%	vec store 3%	prefetch 3%	store 4%
	other 4%	other 20%	other 20%	other 20%	other 18%

Table A.5: Hash join run with the **tpch** dataset.

	scalar	mvl8	mvl16	mvl32	mvl64
total instructions retired	918,638,049	365,111,536	210,252,140	131,882,953	92,931,636
of which are vector	-	96,900,838	49,851,133	26,195,837	14,497,196
scalar µops	1,322,180,244	328,345,772	197,345,801	130,836,993	97,704,333
insn line fetches -l1	715,770,620	127,278,943	77,642,278	50,480,145	36,370,798
-l2	1	1	1	1	1
-mem	38	78	78	78	78
branch predictions -correct	181,425,561	38,980,816	23,710,258	15,905,659	12,146,581
-incorrect	12,906,820	583,791	516,553	370,057	233,395
scalar store/load ratio	0.41	0.23	0.29	0.37	0.48
scalar loads -l1	235,953,838	44,492,304	25,021,640	14,959,459	9,865,084
-l2	40,446,821	4	4	4	3
-mem	29,866,555	66	63	64	71
vector loads -l2	-	48,633,528	42,500,640	39,299,159	35,348,285
-mem		23,654,250	25,378,492	27,057,247	30,246,993
average vector length (stdev)	-	7.67 (1.27)	15.01 (3.22)	29.09 (7.74)	54.64 (19.38)
vector load insns -unit stride	-	13,835,875	7,018,452	3,581,236	1,870,339
-strided		0	0	0	0
-indexed		12,694,067	6,558,254	3,446,868	1,933,462
vector store insns -unit stride	-	10,787,685	5,666,536	2,965,180	1,546,867
-strided		0	0	0	0
-indexed		2,825,590	1,465,012	776,944	442,169
instruction mix	add/sub 23%	add/sub 25%	add/sub 27%	add/sub 31%	add/sub 34%
	lea add/sub 23%	logical 19%	logical 19%	logical 18%	logical 17%
	load 17%	load 10%	load 10%	load 9%	cond. branch 10%
	logical 11%	cond. branch 8%	cond. branch 9%	cond. branch 9%	load 8%
	cond. branch 11%	vec load 6%	vec load 5%	vec load 4%	prefetch 5%
	store 7%	lea add/sub 6%	lea add/sub 5%	lea add/sub 4%	store 4%
	prefetch 3%	vec store 3%	vec store 3%	prefetch 4%	vec load 3%
	other 6%	other 22%	other 22%	other 21%	other 18%

Table A.6: Vectorised quicksort run with the **large** dataset.

	scalar quicksort	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,892,178,304	2,345,590,854	1,586,467,473	1,220,371,635	1,045,327,163
of which are vector	-	425,313,887	255,102,563	172,935,018	133,614,999
scalar µops	2,406,349,732	2,553,171,396	1,806,178,749	1,445,999,064	1,273,809,599
insn line fetches -l1	2,244,181,816	712,760,233	511,096,313	414,483,440	369,524,093
-l2	0	1	1	1	1
-mem	21	39	39	43	39
branch predictions -correct	462,840,164	131,297,721	101,488,744	87,030,774	80,091,720
-incorrect	54,122,580	7,291,359	6,817,912	6,334,924	6,086,310
scalar store/load ratio	0.64	0.60	0.72	0.80	0.84
scalar loads -l1	338,765,894	227,489,071	189,171,099	170,450,578	161,480,135
-l2	11,822,077	1,051,760	1,047,356	1,043,846	1,043,932
-mem	8,150,210	7,546	7,538	7,566	7,580
vector loads -l2	-	68,178,057	47,035,979	38,883,237	35,313,276
-mem		8,516,621	8,518,229	6,949,849	5,883,822
average vector length (stdev)	-	6.14 (2.51)	10.45 (5.98)	16.23 (13.17)	23.04 (26.10)
vector load insns -unit stride		56,637,370	34,183,546	23,362,740	18,191,970
-strided	-	0	0	0	0
-indexed		0	0	0	0
vector store insns -unit stride		86,939,812	52,633,964	35,062,456	26,463,434
-strided	-	0	0	0	0
-indexed		0	0	0	0
instruction mix	add/sub 35%	logical 28%	add/sub 27%	add/sub 28%	add/sub 28%
	cond. branch 16%	add/sub 27%	logical 26%	logical 25%	logical 24%
	lea add/sub 14%	lea add/sub 10%	lea add/sub 9%	load 10%	load 11%
	logical 13%	load 7%	load 9%	lea add/sub 9%	store 9%
	load 10%	store 4%	store 6%	store 8%	lea add/sub 8%
	store 7%	vec store 3%	vec store 3%	unc. branch 3%	unc. branch 3%
	simple shift 3%	vec set-vl 3%	vec set-vl 3%	x86 cc ops 2%	x86 cc ops 2%
	other 2%	other 18%	other 17%	other 16%	other 15%

Table A.7: Vectorised quicksort w/ OET cleanup run with the **large** dataset.

	scalar quicksort	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,892,178,304	1,935,726,270	1,028,300,134	571,324,296	350,839,824
of which are vector	-	415,094,904	211,902,709	109,255,164	59,695,145
scalar µops	2,406,349,732	1,929,170,672	1,048,305,717	604,926,032	391,136,567
insn line fetches -l1	2,244,181,816	532,096,939	293,709,739	173,003,328	116,084,501
-l2	0	1	1	1	1
-mem	21	46	44	45	45
branch predictions -correct	462,840,164	85,200,641	48,927,010	30,781,181	21,959,297
-incorrect	54,122,580	2,523,850	2,324,782	2,133,884	1,969,236
scalar store/load ratio	0.64	0.30	0.41	0.55	0.68
scalar loads -l1	338,765,894	137,166,706	83,108,971	55,680,475	42,285,349
-l2	11,822,077	854,175	852,848	851,024	848,824
-mem	8,150,210	8,370	8,464	8,539	8,539
vector loads -l2	-	65,665,653	43,045,819	34,054,834	32,033,238
-mem		22,111,303	21,913,593	19,903,456	16,649,064
average vector length (stdev)	-	7.10 (1.91)	13.85 (4.17)	27.01 (8.78)	50.33 (20.16)
vector load insns -unit stride	-	44,087,572	22,575,290	11,754,484	6,583,714
-strided		20,480,000	10,240,000	5,120,000	2,560,000
-indexed		0	0	0	0
vector store insns -unit stride	-	69,806,916	36,883,992	19,312,484	10,713,462
-strided		20,480,000	10,240,000	5,120,000	2,560,000
-indexed		0	0	0	0
instruction mix	add/sub 35%	logical 29%	logical 28%	logical 27%	add/sub 26%
	cond. branch 16%	add/sub 26%	add/sub 26%	add/sub 26%	logical 25%
	lea add/sub 14%	lea add/sub 10%	lea add/sub 10%	lea add/sub 10%	lea add/sub 10%
	logical 13%	load 6%	load 6%	load 7%	load 9%
	load 10%	vec store 4%	vec store 4%	store 4%	store 6%
	store 7%	vec permute 3%	vec permute 3%	vec store 3%	vec store 3%
	simple shift 3%	simple shift 3%	simple shift 3%	vec permute 3%	simple shift 3%
	other 2%	other 18%	other 19%	other 19%	other 18%

Table A.8: Vectorised bitonic mergesort run with the **large** dataset.

	scalar quicksort	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,892,178,304	1,178,707,185	632,235,466	371,044,454	201,700,184
of which are vector	-	574,597,154	334,878,761	199,713,972	114,434,303
scalar μops	2,406,349,732	927,883,024	462,831,467	259,510,792	130,517,190
insn line fetches -l1	2,244,181,816	461,456,061	245,320,943	137,886,504	70,312,701
-l2	0	0	0	0	0
-mem	21	115	130	135	157
branch predictions -correct	462,840,164	91,338,911	50,011,941	26,697,417	14,181,469
-incorrect	54,122,580	3,835,570	1,831,731	868,495	210,712
scalar store/load ratio	0.64	0.31	0.30	0.41	0.37
scalar loads -l1	338,765,894	164,900,436	75,731,805	34,067,830	15,783,165
-l2	11,822,077	8,999,416	5,716,857	5,435,033	1,931,949
-mem	8,150,210	3,485,581	1,262,215	3,530,478	1,001,413
vector loads -l2	-	33,753,837	18,316,058	27,800,637	21,789,960
-mem		1,968,829	1,435,612	1,517,645	1,580,060
average vector length (stdev)	-	8.00 (0.00)	16.00 (0.00)	32.00 (0.00)	64.00 (0.00)
vector load insns -unit stride		35,402,667	19,431,672	20,771,531	15,386,274
-strided	-	0	0	0	0
-indexed		0	0	0	0
vector store insns -unit stride		25,128,960	11,924,480	5,642,240	2,661,120
-strided	-	0	0	0	0
-indexed		0	0	0	0
instruction mix	add/sub 35%	vec permute 28%	vec permute 32%	vec permute 31%	vec permute 33%
	cond. branch 16%	add/sub 18%	add/sub 16%	add/sub 14%	logical 14%
	lea add/sub 14%	logical 12%	logical 11%	logical 13%	add/sub 12%
	logical 13%	load 11%	load 9%	load 9%	load 7%
	load 10%	lea add/sub 7%	lea add/sub 6%	lea add/sub 5%	vec load 6%
	store 7%	vec comparison 3%	x86 cc ops 4%	vec load 5%	x86 cc ops 5%
	simple shift 3%	store 3%	vec comparison 4%	x86 cc ops 4%	lea add/sub 5%
	other 2%	other 18%	other 18%	other 19%	other 18%

Table A.9: Vectorised radix sort (bins=16) run with the **large** dataset.

	scalar quicksort	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,892,178,304	337,925,934	168,966,578	84,487,862	42,250,422
of which are vector	-	117,760,956	58,881,084	29,441,340	14,721,852
scalar μops	2,406,349,732	256,005,782	128,006,298	64,007,326	32,009,373
insn line fetches -l1	2,244,181,816	107,523,703	53,764,288	26,884,735	13,445,490
-l2	0	5	4	4	4
-mem	21	44	42	42	42
branch predictions -correct	462,840,164	20,480,801	10,240,849	5,120,978	2,561,242
-incorrect	54,122,580	128	133	136	138
scalar store/load ratio	0.64	0.00	0.00	0.00	0.00
scalar loads -l1	338,765,894	46,080,731	23,040,737	11,520,749	5,760,770
-l2	11,822,077	1	1	0	1
-mem	8,150,210	29	26	19	17
vector loads -l2	-	166,868,021	165,876,602	165,879,450	165,874,237
-mem		7,679,837	7,679,865	7,680,084	7,690,639
average vector length (stdev)	-	8.00 (0.00)	16.00 (0.00)	32.00 (0.00)	64.00 (0.00)
vector load insns -unit stride		0	0	0	0
-strided	-	15,360,256	7,680,257	3,840,258	1,920,258
-indexed		10,240,000	5,120,000	2,560,000	1,280,000
vector store insns -unit stride		128	128	128	128
-strided	-	128	128	128	128
-indexed		20,480,000	10,240,000	5,120,000	2,560,000
instruction mix	add/sub 35%	logical 30%	logical 30%	logical 30%	logical 30%
	cond. branch 16%	add/sub 18%	add/sub 18%	add/sub 18%	add/sub 18%
	lea add/sub 14%	load 12%	load 12%	load 12%	load 12%
	logical 13%	vec logical 8%	vec logical 8%	vec logical 8%	vec logical 8%
	load 10%	vec load 7%	vec load 7%	vec load 7%	vec load 7%
	store 7%	vec store 5%	vec store 5%	vec store 5%	vec store 5%
	simple shift 3%	vec add/sub 5%	vec add/sub 5%	vec add/sub 5%	vec add/sub 5%
	other 2%	other 14%	other 14%	other 14%	other 14%

Table A.10: Vectorised VSR sort (bins=256) run with the **large** dataset.

	scalar quicksort	mvl8	mvl16	mvl32	mvl64
total instructions retired	1,892,178,304	273,928,320	136,965,664	68,484,864	34,245,520
of which are vector	-	92,160,984	46,080,600	23,040,504	11,520,648
scalar μops	2,406,349,732	217,608,115	108,805,779	54,405,043	27,205,539
insn line fetches -l1	2,244,181,816	71,683,621	35,843,204	17,922,769	8,963,078
-l2	0	4	4	4	4
-mem	21	49	49	50	51
branch predictions -correct	462,840,164	12,800,758	6,400,583	3,200,561	1,600,637
-incorrect	54,122,580	102	114	102	105
scalar store/load ratio	0.64	0.00	0.00	0.00	0.00
scalar loads -l1	338,765,894	15,360,504	7,680,545	3,840,518	1,920,537
-l2	11,822,077	1	1	0	0
-mem	8,150,210	17	19	21	12
vector loads -l2	-	35,709,920	26,432,723	19,312,359	12,527,646
-mem		3,469,768	1,705,457	464,347	148,685
average vector length (stdev)	-	7.97 (0.19)	15.86 (0.44)	31.45 (1.24)	61.88 (4.16)
vector load insns -unit stride	-	7,680,000	3,840,000	1,920,000	960,000
-strided		256	128	64	33
-indexed		5,120,000	2,560,000	1,280,000	640,000
vector store insns -unit stride	-	128	64	32	16
-strided		128	64	32	16
-indexed		10,240,000	5,120,000	2,560,000	1,280,000
instruction mix	add/sub 35%	logical 29%	logical 29%	logical 29%	logical 29%
	cond. branch 16%	add/sub 21%	add/sub 21%	add/sub 21%	add/sub 21%
	lea add/sub 14%	lea add/sub 6%	lea add/sub 6%	lea add/sub 6%	lea add/sub 6%
	logical 13%	load 5%	load 5%	load 5%	load 5%
	load 10%	vec add/sub 5%	vec add/sub 5%	vec add/sub 5%	vec add/sub 5%
	store 7%	vec load 4%	vec load 4%	vec load 4%	vec load 4%
	simple shift 3%	vec store 3%	vec store 3%	x86 cc ops 3%	x86 cc ops 3%
	other 2%	other 26%	other 26%	other 26%	other 26%

Table A.11: Aggregation ***scalar*** where $c = 152$ (*low*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	150,004,473	150,004,545	150,004,555	150,004,472	150,004,474
of which are vector	-	-	-	-	-
scalar µops	305,007,147	305,007,221	305,007,231	305,007,146	305,007,150
insn line fetches -l1	50,002,659	50,002,644	50,005,234	50,002,699	50,002,711
-l2	4	4	3	4	5
-mem	56	56	55	56	55
branch predictions -correct	15,013,323	15,011,848	15,000,931	15,002,248	15,009,837
-incorrect	101	96	180	101	100
scalar store/load ratio	0.40	0.40	0.40	0.40	0.40
scalar loads -l1	45,412,504	45,392,039	45,371,310	45,352,152	45,392,621
-l2	4,318,870	4,323,392	4,324,571	4,325,019	4,322,671
-mem	337,989	341,889	305,920	333,214	335,579
vector loads -l2 -mem	-	-	-	-	-
average vector length (stdev)	-	-	-	-	-
vector load insns -unit stride -strided -indexed	-	-	-	-	-
vector store insns -unit stride -strided -indexed	-	-	-	-	-
instruction mix	lea add/sub 33% add/sub 23% load 16% prefetch 10% store 7% cond. branch 5% simple shift 3% other 3%	lea add/sub 33% add/sub 23% load 16% prefetch 10% store 7% cond. branch 5% simple shift 3% other 3%	lea add/sub 33% add/sub 23% load 16% prefetch 10% store 7% cond. branch 5% simple shift 3% other 3%	lea add/sub 33% add/sub 23% load 16% prefetch 10% store 7% cond. branch 5% simple shift 3% other 3%	lea add/sub 33% add/sub 23% load 16% prefetch 10% store 7% cond. branch 5% simple shift 3% other 3%

Table A.12: Aggregation ***scalar*** where $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	158,594,870	158,908,633	158,909,024	158,596,140	158,111,608
of which are vector	-	-	-	-	-
scalar μops	320,157,376	320,472,211	320,472,602	320,159,720	319,265,162
insn line fetches -l1	52,662,660	52,658,134	62,158,485	52,658,323	54,814,599
-l2	4	4	4	4	4
-mem	56	54	56	56	55
branch predictions -correct	15,996,519	15,996,206	16,564,778	15,977,061	15,980,994
-incorrect	300	100	313,932	105	88,825
scalar store/load ratio	0.43	0.43	0.43	0.43	0.43
scalar loads -l1	37,590,059	43,585,769	48,530,598	28,044,568	37,953,292
-l2	4,528,769	4,816,444	4,461,901	4,518,291	7,745,547
-mem	9,848,791	3,594,547	219,221	19,313,083	6,349,735
vector loads -l2 -mem	-	-	-	-	-
average vector length (stdev)	-	-	-	-	-
vector load insns -unit stride -strided -indexed	-	-	-	-	-
vector store insns -unit stride -strided -indexed	-	-	-	-	-
instruction mix	lea add/sub 33%	lea add/sub 33%	lea add/sub 33%	lea add/sub 33%	lea add/sub 33%
	add/sub 23%	add/sub 23%	add/sub 23%	add/sub 23%	add/sub 23%
	load 16%	load 16%	load 16%	load 16%	load 16%
	prefetch 10%	prefetch 10%	prefetch 10%	prefetch 10%	prefetch 10%
	store 7%	store 7%	store 7%	store 7%	store 7%
	cond. branch 5%	cond. branch 5%	cond. branch 5%	cond. branch 5%	cond. branch 5%
	simple shift 3%	simple shift 3%	simple shift 3%	simple shift 3%	simple shift 3%
	other 3%	other 3%	other 3%	other 3%	other 3%

Table A.13: Aggregation ***standard sorted reduce*** where $MVL = 64$ and $c = 152$ (*low*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	38,454,665	38,452,374	14,852,147	38,454,520	38,454,404
of which are vector	10,469,871	10,469,555	2,813,091	10,469,851	10,469,835
scalar μops	32,988,360	32,986,227	14,071,842	32,988,225	32,988,117
insn line fetches -l1	10,952,088	10,952,067	3,759,056	10,952,233	10,952,443
-l2	0	0	3	1	1
-mem	125	127	42	124	125
branch predictions -correct	2,346,019	2,345,986	782,208	2,346,048	2,346,069
-incorrect	540	549	360	541	549
scalar store/load ratio	0.00	0.00	0.00	0.00	0.00
scalar loads -l1	4,691,835	4,691,889	783,884	4,691,884	4,691,899
-l2	0	0	2	0	1
-mem	61	62	30	61	61
vector loads -l2	73,652,786	72,532,702	1,774,914	81,077,133	76,029,365
-mem	4,492,310	4,523,701	405,466	4,500,766	4,495,823
average vector length (stdev)	63.99 (0.62)	63.99 (0.57)	63.96 (1.49)	63.99 (0.62)	63.99 (0.61)
vector load insns -unit stride	625,275	625,212	469,052	625,294	625,286
-strided	937,564	937,565	0	937,565	937,565
-indexed	625,000	625,000	0	625,000	625,000
vector store insns -unit stride	334	334	302	334	334
-strided	32	32	0	32	32
-indexed	1,250,000	1,250,000	0	1,250,000	1,250,000
instruction mix	logical 30%	logical 30%	logical 34%	logical 30%	logical 30%
	add/sub 23%	add/sub 23%	add/sub 29%	add/sub 23%	add/sub 23%
	load 11%	load 11%	lea add/sub 7%	load 11%	load 11%
	vec load 5%	vec load 5%	load 5%	vec load 5%	vec load 5%
	vec logical 4%	vec logical 4%	cond. branch 3%	vec logical 4%	vec logical 4%
	lea add/sub 4%	lea add/sub 4%	vec set-vl 3%	lea add/sub 4%	lea add/sub 4%
	vec store 4%	vec store 4%	vec load 3%	vec store 4%	vec store 4%
	other 19%	other 19%	other 17%	other 19%	other 19%

Table A.14: Aggregation ***standard sorted reduce*** where $MVL = 64$ and $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	108,048,298	106,263,160	45,626,509	106,263,160	104,650,957
of which are vector	24,375,304	24,219,922	5,000,009	24,219,922	24,147,520
scalar μops	96,643,006	94,858,453	46,720,886	94,858,453	93,009,886
insn line fetches -l1	29,849,933	29,383,558	11,563,871	29,383,938	29,203,033
-l2	1	1	3	0	0
-mem	131	128	43	133	130
branch predictions -correct	6,954,338	6,877,012	3,125,303	6,877,036	6,685,108
-incorrect	282	259	63	268	12,594
scalar store/load ratio	0.30	0.30	0.68	0.30	0.28
scalar loads -l1	14,492,917	14,494,593	6,367,686	14,494,639	13,850,172
-l2	39,053	39,063	39,064	39,064	34,390
-mem	83	79	51	84	90
vector loads -l2	181,926,137	185,769,659	2,599,994	204,952,607	185,446,329
-mem	11,180,882	11,086,540	16,965	10,997,530	11,292,862
average vector length (stdev)	59.26 (15.61)	58.76 (15.76)	29.33 (25.09)	58.76 (15.80)	59.82 (15.13)
vector load insns -unit stride	1,484,148	1,406,254	937,502	1,406,252	1,444,584
-strided	2,343,910	2,343,910	0	2,343,910	2,343,911
-indexed	1,562,500	1,562,500	0	1,562,500	1,562,500
vector store insns -unit stride	468,918	625,080	312,500	625,080	425,050
-strided	80	80	0	80	80
-indexed	3,125,000	3,125,000	0	3,125,000	3,125,000
instruction mix	logical 30%	logical 30%	logical 28%	logical 30%	logical 30%
	add/sub 21%	add/sub 21%	add/sub 25%	add/sub 21%	add/sub 21%
	load 12%	load 12%	load 12%	load 12%	load 12%
	vec load 4%	vec load 4%	store 8%	vec load 4%	vec load 5%
	vec logical 4%	vec logical 4%	lea add/sub 5%	vec logical 4%	vec logical 4%
	store 4%	store 4%	cond. branch 4%	store 4%	cond. branch 4%
	cond. branch 4%	cond. branch 4%	simple shift 2%	cond. branch 4%	store 3%
	other 21%	other 21%	other 15%	other 21%	other 21%

Table A.15: Aggregation ***polytable*** where $MVL = 64$ and $c = 152$ (*low*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	8,919,244	8,919,244	6,419,231	8,919,244	8,919,244
of which are vector	2,501,284	2,501,284	2,032,531	2,501,284	2,501,284
scalar µops	8,295,721	8,295,721	5,483,217	8,295,721	8,295,721
insn line fetches -l1	2,817,953	2,818,010	1,880,690	2,817,952	2,818,012
-l2	0	1	17	1	0
-mem	85	83	69	83	85
branch predictions -correct	783,054	783,045	470,545	783,045	783,050
-incorrect	138	140	132	138	142
scalar store/load ratio	0.00	0.00	0.00	0.00	0.00
scalar loads -l1	938,747	938,753	313,731	938,752	938,755
-l2	0	0	2	0	0
-mem	45	44	36	43	48
vector loads -l2	12,494,803	21,553,729	2,496,788	20,596,500	16,780,233
-mem	325,715	322,507	4,736	322,806	324,345
average vector length (stdev)	64.00 (0.09)	64.00 (0.09)	64.00 (0.10)	64.00 (0.09)	64.00 (0.09)
vector load insns -unit stride	469,060	469,060	312,810	469,060	469,060
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
vector store insns -unit stride	313	313	313	313	313
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
instruction mix	add/sub 26% logical 22% lea add/sub 10% load 9% vec load 7% vec add/sub 7% cond. branch 6% other 13%	add/sub 26% logical 22% lea add/sub 10% load 9% vec load 7% vec add/sub 7% cond. branch 6% other 13%	logical 25% add/sub 23% lea add/sub 10% vec load 8% vec add/sub 8% cond. branch 4% load 4% other 17%	add/sub 26% logical 22% lea add/sub 10% load 9% vec load 7% vec add/sub 7% cond. branch 6% other 13%	add/sub 26% logical 22% lea add/sub 10% load 9% vec load 7% vec add/sub 7% cond. branch 6% other 13%

Table A.16: Aggregation ***polytable*** where $MVL = 64$ and $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	45,628,647	45,628,647	43,128,634	45,628,647	45,628,647
of which are vector	7,646,513	7,646,513	7,177,760	7,646,513	7,646,513
scalar µops	44,370,334	44,370,334	41,557,830	44,370,334	44,370,334
insn line fetches -l1	12,336,554	12,336,567	11,399,114	12,336,550	12,336,456
-l2	0	0	9	0	0
-mem	73	72	61	75	68
branch predictions -correct	4,561,310	4,561,308	4,248,819	4,561,299	4,561,307
-incorrect	115	118	111	116	109
scalar store/load ratio	1.21	1.21	3.04	1.21	1.21
scalar loads -l1	1,036,236	1,036,216	411,225	1,036,225	1,036,207
-l2	3	5	7	4	3
-mem	80	77	75	80	83
vector loads -l2	5,774,566	4,522,672	4,177,737	4,533,582	10,676,720
-mem	12,434,068	22,430,454	4,571,791	22,419,306	15,260,960
average vector length (stdev)	64.00 (0.03)	64.00 (0.03)	64.00 (0.03)	64.00 (0.03)	63.96 (0.75)
vector load insns -unit stride	1,738,282	1,738,282	1,582,032	1,738,282	1,738,282
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
vector store insns -unit stride	1,279,298	1,279,298	1,279,298	1,279,298	1,279,298
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
instruction mix	logical 26%	logical 26%	logical 27%	logical 26%	logical 26%
	add/sub 26%	add/sub 26%	add/sub 26%	add/sub 26%	add/sub 26%
	lea add/sub 8%	lea add/sub 8%	lea add/sub 8%	lea add/sub 8%	lea add/sub 8%
	cond. branch 7%	cond. branch 7%	cond. branch 7%	cond. branch 7%	cond. branch 7%
	simple shift 5%	simple shift 5%	simple shift 5%	simple shift 5%	simple shift 5%
	mul 5%	mul 5%	mul 5%	mul 5%	mul 5%
	vec load 4%	vec load 4%	vec load 4%	vec load 4%	vec load 4%
	other 18%	other 18%	other 18%	other 18%	other 18%

Table A.17: Aggregation ***advanced sorted reduce*** where $MVL = 64$ and $c = 152$ (*low*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	40,012,207	40,009,916	14,852,147	40,012,062	40,011,946
of which are vector	9,688,318	9,688,002	2,813,091	9,688,298	9,688,282
scalar μops	36,420,130	36,417,997	14,071,842	36,419,995	36,419,887
insn line fetches -l1	10,479,685	10,479,196	3,758,773	10,479,830	10,479,735
-l2	1	1	3	0	0
-mem	121	122	45	123	122
branch predictions -correct	2,189,007	2,188,928	782,172	2,189,005	2,189,014
-incorrect	463	466	363	468	466
scalar store/load ratio	0.00	0.00	0.00	0.00	0.00
scalar loads -l1	2,815,736	2,815,750	783,822	2,815,764	2,815,784
-l2	1	3	1	3	2
-mem	71	74	34	72	68
vector loads -l2	6,669,608	5,268,452	1,774,304	6,959,256	6,330,331
-mem	2,254,298	2,119,499	405,980	2,081,135	2,250,911
average vector length (stdev)	58.72 (12.47)	63.99 (0.63)	63.96 (1.49)	62.20 (4.37)	59.45 (10.76)
vector load insns -unit stride	1,406,543	1,406,457	469,030	1,406,543	1,406,533
-strided	8	8	0	8	8
-indexed	312,500	312,500	0	312,500	312,500
vector store insns -unit stride	312,806	312,806	302	312,806	312,806
-strided	4	4	0	4	4
-indexed	625,000	625,000	0	625,000	625,000
instruction mix	logical 33%	logical 33%	logical 34%	logical 33%	logical 33%
	add/sub 24%	add/sub 24%	add/sub 29%	add/sub 24%	add/sub 24%
	lea add/sub 7%	lea add/sub 7%	lea add/sub 7%	lea add/sub 7%	lea add/sub 7%
	load 6%	load 6%	load 5%	load 6%	load 6%
	vec load 4%	vec load 4%	cond. branch 3%	vec load 4%	vec load 4%
	cond. branch 3%	cond. branch 3%	vec set-vl 3%	cond. branch 3%	cond. branch 3%
	vec set-vl 3%	vec set-vl 3%	vec load 3%	vec set-vl 3%	vec set-vl 3%
	other 20%	other 20%	other 17%	other 20%	other 20%

Table A.18: Aggregation ***advanced sorted reduce*** where $MVL = 64$ and $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	83,823,386	82,038,248	45,626,509	82,038,248	80,426,045
of which are vector	16,874,636	16,719,254	5,000,009	16,719,254	16,646,852
scalar μops	78,355,157	76,570,604	46,720,886	76,570,604	74,722,037
insn line fetches -l1	22,032,868	21,566,659	11,563,881	21,566,749	21,387,205
-l2	0	0	4	0	1
-mem	114	116	44	115	116
branch predictions -correct	5,078,332	5,001,005	3,125,298	5,001,026	4,809,253
-incorrect	193	171	62	174	12,560
scalar store/load ratio	0.47	0.47	0.68	0.47	0.44
scalar loads -l1	9,335,477	9,337,158	6,367,672	9,337,172	8,692,953
-l2	39,052	39,067	39,063	39,064	34,393
-mem	100	89	52	91	110
vector loads -l2	19,216,136	8,455,357	2,599,795	28,936,579	19,503,510
-mem	994,274	918,797	17,164	531,764	1,071,314
average vector length (stdev)	50.27 (22.96)	55.09 (19.74)	29.33 (25.09)	54.65 (19.62)	52.38 (21.56)
vector load insns -unit stride	2,109,150	2,031,252	937,502	2,031,252	2,069,616
-strided	64	64	0	64	64
-indexed	625,000	625,000	0	625,000	625,000
vector store insns -unit stride	156,370	312,532	312,500	312,532	112,502
-strided	32	32	0	32	32
-indexed	1,250,000	1,250,000	0	1,250,000	1,250,000
instruction mix	logical 28%	logical 28%	logical 28%	logical 28%	logical 28%
	add/sub 24%	add/sub 24%	add/sub 25%	add/sub 24%	add/sub 24%
	load 10%	load 10%	load 12%	load 10%	load 10%
	lea add/sub 6%	lea add/sub 6%	store 8%	lea add/sub 6%	lea add/sub 6%
	store 5%	store 5%	lea add/sub 5%	store 5%	store 4%
	cond. branch 3%	cond. branch 3%	cond. branch 4%	cond. branch 3%	cond. branch 3%
	vec load 3%	vec load 3%	simple shift 2%	vec load 3%	vec load 3%
	other 22%	other 22%	other 15%	other 22%	other 22%

Table A.19: Aggregation ***monotable*** where $MVL = 64$ and $c = 152$ (*low*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	8,127,221	8,127,221	5,627,208	8,127,221	8,127,221
of which are vector	2,656,320	2,656,320	2,187,567	2,656,320	2,656,320
scalar µops	7,659,292	7,659,292	4,846,788	7,659,292	7,659,292
insn line fetches -l1	2,814,227	2,814,253	1,876,685	2,814,231	2,814,250
-l2	0	0	3	0	0
-mem	52	53	49	52	53
branch predictions -correct	781,665	781,662	469,169	781,669	781,662
-incorrect	85	85	67	85	85
scalar store/load ratio	0.00	0.00	0.00	0.00	0.00
scalar loads -l1	1,719,467	1,719,469	1,094,432	1,719,480	1,719,469
-l2	0	0	0	0	0
-mem	27	29	26	27	28
vector loads -l2	4,537,764	3,013,499	1,558,409	4,662,438	4,230,232
-mem	327,158	325,335	4,129	325,638	325,416
average vector length (stdev)	64.00 (0.10)	64.00 (0.10)	64.00 (0.11)	64.00 (0.10)	64.00 (0.10)
vector load insns -unit stride	468,756	468,756	312,506	468,756	468,756
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
vector store insns -unit stride	15	15	15	15	15
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
instruction mix	add/sub 24% load 17% logical 14% lea add/sub 9% vec load 8% vec add/sub 8% cond. branch 6% other 15%	add/sub 24% load 17% logical 14% lea add/sub 9% vec load 8% vec add/sub 8% cond. branch 6% other 15%	add/sub 20% load 16% logical 13% lea add/sub 9% vec load 9% vec add/sub 9% cond. branch 4% other 20%	add/sub 24% load 17% logical 14% lea add/sub 9% vec load 8% vec add/sub 8% cond. branch 6% other 15%	add/sub 24% load 17% logical 14% lea add/sub 9% vec load 8% vec add/sub 8% cond. branch 6% other 15%

Table A.20: Aggregation ***monotable*** where $MVL = 64$ and $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	8,869,209	8,869,209	6,369,196	8,869,209	8,869,209
of which are vector	2,822,291	2,822,291	2,353,538	2,822,291	2,822,291
scalar μops	8,391,517	8,391,517	5,579,013	8,391,517	8,391,517
insn line fetches -l1	3,048,568	3,048,565	2,110,999	3,048,552	3,048,561
-l2	0	0	3	0	0
-mem	52	52	48	52	51
branch predictions -correct	850,006	850,008	537,512	850,003	850,010
-incorrect	88	90	74	89	89
scalar store/load ratio	0.00	0.00	0.00	0.00	0.00
scalar loads -l1	1,836,631	1,836,623	1,211,605	1,836,627	1,836,637
-l2	0	0	0	0	0
-mem	27	31	26	27	27
vector loads -l2	2,383,257	1,763,979	1,616,184	2,578,303	9,178,148
-mem	9,883,659	1,595,411	101,452	19,358,619	6,385,278
average vector length (stdev)	64.00 (0.06)	64.00 (0.06)	64.00 (0.06)	64.00 (0.06)	63.90 (1.22)
vector load insns -unit stride	488,282	488,282	332,032	488,282	488,282
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
vector store insns -unit stride	48,830	48,830	48,830	48,830	48,830
-strided	0	0	0	0	0
-indexed	312,500	312,500	312,500	312,500	312,500
instruction mix	add/sub 24% load 16% logical 14% lea add/sub 9% vec load 7% vec add/sub 7% cond. branch 6% other 16%	add/sub 24% load 16% logical 14% lea add/sub 9% vec load 7% vec add/sub 7% cond. branch 6% other 16%	add/sub 21% load 15% logical 14% lea add/sub 9% vec load 8% vec add/sub 8% cond. branch 5% other 20%	add/sub 24% load 16% logical 14% lea add/sub 9% vec load 7% vec add/sub 7% cond. branch 6% other 16%	add/sub 24% load 16% logical 14% lea add/sub 9% vec load 7% vec add/sub 7% cond. branch 6% other 16%

Table A.21: Aggregation ***partially sorted monotable*** where $MVL = 64$ and $c = 625,000$ (*high*).

	hhitter	sequential	sorted	uniform	zipf
total instructions retired	21,374,565	21,374,565	-	21,374,565	21,374,565
of which are vector	8,447,453	8,447,453	-	8,447,453	8,447,453
scalar μops	17,616,977	17,616,977	-	17,616,977	17,616,971
insn line fetches -l1	6,636,150	6,636,114	-	6,636,118	6,636,238
-l2	0	0		0	0
-mem	123	123		122	124
branch predictions -correct	1,788,462	1,788,464	-	1,788,457	1,788,490
-incorrect	211	213		210	220
scalar store/load ratio	0.00	0.00	-	0.00	0.00
scalar loads -l1	2,931,621	2,931,621	-	2,931,608	2,931,651
-l2	0	0		0	0
-mem	65	64		65	67
vector loads -l2	15,122,210	4,761,477	-	24,116,200	17,262,134
-mem	731,043	630,177		549,927	721,731
average vector length (stdev)	58.70 (12.47)	54.25 (22.79)	-	62.21 (4.30)	56.87 (16.62)
vector load insns -unit stride	957,032	957,032	-	957,032	957,032
-strided	11	9		10	10
-indexed	625,000	625,000		625,000	625,000
vector store insns -unit stride	48,835	48,835	-	48,835	48,835
-strided	4	4		4	4
-indexed	937,500	937,500		937,500	937,500
instruction mix	add/sub 19%	add/sub 19%	-	add/sub 19%	add/sub 19%
	logical 17%	logical 17%		logical 17%	logical 17%
	load 11%	load 11%		load 11%	load 11%
	lea add/sub 8%	lea add/sub 8%		lea add/sub 8%	lea add/sub 8%
	vec add/sub 7%	vec add/sub 7%		vec add/sub 7%	vec add/sub 7%
	vec load 6%	vec load 6%		vec load 6%	vec load 6%
	cond. branch 5%	cond. branch 5%		cond. branch 5%	cond. branch 5%
	other 27%	other 27%		other 27%	other 27%

APPENDIX B

Vector Instruction Set Architecture

In this appendix, we provide formal definitions of the instructions used throughout the thesis. These instructions extend the x86-64 ISA, however, their style and format are characteristically different. In a similar vein to VMIPS [HP12, Asa98], we construct our vector ISA extensions in a RISC-like way. This format is flexible, extensible and simple to implement. Where x86-64 uses variable length instructions, all of our new instructions are fixed length. x86-64 operands may come from either registers, main memory or the instruction itself, i.e. immediates, whereas the operands of our new instructions are all registers. Furthermore, x86-64 instructions typically use two source operands where one also serves as the operation's destination, i.e. instructions are destructive to its operands. Our instructions allow up to three encoded source operands and a different destination register, i.e. operations are optionally non-destructive.

B.1 General Format

Each new instruction is encoded using six bytes. Although this might be seen as quite large for an instruction, there are two important points which justify using this size. Firstly, this is an on-going research project and we prefer to have ample room to extend the ISA rapidly in order to prototype new ideas. Having unused bytes in our instructions allows us to easily append new functionality without worrying about backwards compatibility. Secondly, the new vector instructions encode a lot of work, therefore, the semantic density is generally much higher than a typical x86-64 instruction.

To extend the x86-64 ISA, we use a prefix to distinguish our new instructions from existing ones. We have chosen `0xF1` as a single-byte prefix to mark all of the instructions listed in this appendix. This prefix was formerly used for In-Circuit Emulation (ICE) breakpoints, however, it is never used in regular x86-64 applications and is therefore adequate for simulation purposes. The second byte of the new instructions is always an 8-bit opcode while the remaining four bytes are used to encode the instruction's parameters. Each parameter is encoded using one nyble as this makes coding and debugging easier.

B.2 Registers

Using nybles to encode instruction parameters places a restriction on the number of architectural registers that can be included. Each parameter can address one of sixteen registers of each type—scalar, vector and mask. Using sixteen registers aligns nicely with the existing x86-64 ISA which already includes sixteen-general purpose registers. We find sixteen vector registers to be sufficient when vectorising the DBMS kernels evaluated in this thesis which seldom require spill code. With respect to mask registers, sixteen would be overkill—instead, we use four registers and the remaining twelve symbols are used to indicate that the operation is not masked. The registers in the ISA are as follows—

- 16× general purpose (scalar) x86-64 registers – $sr0 \rightarrow sr15$
- 16× vector registers – $vr0 \rightarrow vr15$
- 4× mask registers – $mr0 \rightarrow mr3$
- 1× vector length register – $vlen$

B.3 Datatypes

The new ISA additions currently supports eight datatypes—**signed** and **unsigned byte**, **word**, **long** and **quad** integer types which are one, two, four and eight bytes respectively. Since four bits are reserved for each instruction parameter, it would be possible to easily extend the ISA to include eight more datatypes in the future, e.g. wider integers or even floating point datatypes.

To make the hardware simpler, we do not allow composable vector lengths, hence, there is no subword parallelism. This means that the maximum vector length will be fixed for all datatypes, e.g. being able to operate on sixty-four 64-bit integers does not imply it is possible to operate on 128 32-bit integers. Another motivation for doing this is to simplify the indexed memory instructions which typically require a larger datatype for indices.

B.4 Vector Instruction Listing

This section lists all of the new instructions used to vectorise the applications discussed in this thesis. Each entry provides the name of the instruction, a list of input parameters and pseudocode describing the instruction's semantics. Instructions which partially write to a vector register are non-destructive, therefore, if configuring the $vlen$ to a value less than the MVL or using a mask, the untouched elements of the destination vector register will remain intact. Furthermore, while $vlen$ is used in many instructions, it is an implicit operand and therefore is not encoded in the instruction itself.

Instructions are listed with shorthand notation. `vrd` refers to the destination vector register whereas `vra`, `vrb` and `vrc` refer to operand vector registers a, b and c respectively. Similarly `srd` refers to the destination scalar (x86-64) register whereas `sra` and `srb` refer to operand scalar registers a and b. Likewise, `maskd` refers to the destination mask register whereas `maska` (or simply `mask`) and `maskb` refer to operand mask registers a and b.

The new instructions are classified as follows—

1. Vector Memory
2. Value Initialisation
3. Arithmetic
4. Logical
5. Comparison
6. Permutative
7. Reduction
8. CAM-Based
9. Miscellaneous
10. Mask Manipulation
11. Vector Length

The arithmetic, logical and comparison instructions are listed with two vector source operands, i.e. vector-vector. There are cases where it made sense to also include a vector-scalar variant of some of these instructions; for brevity we don't list these as the semantics should be easily inferred from their vector-vector counterparts.

The base address of vector memory instructions must be in x86-64 canonical form and must be aligned to the datatype of the values being loaded, e.g. it is not legal to load from an array of 32-bit values if the base address if odd.

B.4.1 Vector Memory

vector load: unit stride

```
array ← (dtype*) sra
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← array[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
sra	64-bit base address

vector prefetch: unit stride

```
array ← (dtype*) sra
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        prefetch array[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
sra	64-bit base address

vector store: unit stride

```
array ← (dtype*) sra
for (i ← 0, i<vlen, i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        array[i] ← vra[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vra	vector of source values
sra	64-bit base address

vector load: strided

```
array ← (dtype*) sra
for (i ← 0, i<vlen, i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← array[i·srb]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
sra	64-bit base address
srb	stride of operation

vector prefetch: strided

```
array ← (dtype*) sra
for (i ← 0, i<vlen, i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        prefetch array[i·srb]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
sra	64-bit base address
srb	stride of operation

vector store: strided

```
array ← (dtype*) sra
for (i ← 0, i<vlen, i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        array[i·srb] ← vra[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
sra	64-bit base address
srb	stride of operation
vra	vector of values

vector load: indexed

```
array ← (dtype*) sra
for (i ← 0, i<vlen, i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        if sra = 0 then
            vrd[i] ← *vra[i]
        else
            vrd[i] ← array[vra[i]]
        end if
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
sra	64-bit base address
vra	vector of memory offsets

– If *sra* = 0 then the behaviour is redefined to use absolute addresses.

vector prefetch: indexed

```
array ← (dtype*) sra
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        if sra = 0 then
            prefetch *vra[i]
        else
            prefetch array[vra[i]]
        end if
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
sra	64-bit base address
vra	vector of memory offsets

– If *sra* = 0 then the behaviour is redefined to use absolute addresses.

vector store: indexed

```
array ← (dtype*) sra
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        if sra = 0 then
            *vra[i] ← vrb[i]
        else
            array[ vra[i] ] ← vrb[i]
        end if
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
sra	64-bit base address
vra	vector of memory offsets
vrb	vector of values

– If *sra* = 0 then the behaviour is redefined to use absolute addresses.

– If *vra* contains repeated values, i.e. conflicting indices, behaviour is undefined.

B.4.2 Value Initialisation

vector set: clear

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← 0
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector

vector set: one to all

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← sra
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
sra	source value

vector set: iota

```
iota ← sra
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← iota
    end if
    iota ← iota+1
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
sra	starting value

B.4.3 Arithmetic

vector-vector add

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] + vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	vector of sums
vra	vector of first addends
vrb	vector of second addends

vector-vector subtract

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] - vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	vector of differences
vra	vector of minuends
vrb	vector of subtrahends

vector-vector multiply

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] · vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	vector of products
vra	vector of first factors
vrb	vector of second factors

B.4.4 Logical

vector-vector bitwise: and

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] & vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
vra	vector of first values
vrb	vector of second values

vector-vector bitwise: exclusive or

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] ∧ vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
vra	vector of first values
vrb	vector of second values

vector-vector bitwise: shift right

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] » vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
vra	vector of values
vrb	vector of shift amounts

– Shift is arithmetic or logical depending on whether the datatype is signed or unsigned.

vector-vector bitwise: shift left

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        vrd[i] ← vra[i] « vrb[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
vrd	destination vector
vra	vector of values
vrb	vector of shift amounts

B.4.5 Comparison

vector-vector compare: not equal

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        maskd[i] ← (vra[i] ≠ vrb[i]) ? 1 : 0
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
maskd	destination mask
vra	vector of first values
vrb	vector of second values

vector-vector compare: greater than

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        maskd[i] ← (vra[i] > vrb[i]) ? 1 : 0
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
maskd	destination mask
vra	vector of first values
vrb	vector of second values

vector-vector compare: less than

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        maskd[i] ← (vra[i] < vrb[i]) ? 1 : 0
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
maskd	destination mask
vra	vector of first values
vrb	vector of second values

B.4.6 Permutative

vector compress

```
k ← 0
for (i ← 0,  i<vlen,  i ← i+1) do
    if  mask[i]  then
        vrd[k] ← vra[i]
        k ← k+1
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(required) mask register
vrd	destination vector
vra	vector of source values

vector expand

```
k ← 0
for (i ← 0,  i<vlen,  i ← i+1) do
    if  mask[i]  then
        vrd[i] ← vra[k]
        k ← k+1
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(required) mask register
vrd	destination vector
vra	vector of source values

vector reverse

```
j ← vlen−1
for (i ← 0,  i<vlen,  i ← i+1) do
    vrd[i] ← vra[j]
    j ← j−1
end for
```

vlen	vector length
dtype	datatype of values
vrd	destination vector
vra	vector of source values

vector shuffle

```
for (i ← 0,  i<vlen,  i ← i+1) do
    bitpos ← $log_2$(maxvlen) + 1
    regbit ← getbit(bitpos, vrc[i])
    srcreg ← regbit ? vrb : vra
    srcelm ← vrc[i] & maxvlen-1
    vrd[i] ← srcreg[srcelm]
end for
```

vlen	vector length
dtype	datatype of values
vrd	destination vector
vra	1st vector of source values
vrb	2nd vector of source values
vrc	vector of source register/element

– The least significant log_2(maxvlen) bits are used to determine the source element.

– The bit at position log_2(maxvlen) + 1 is used to determine the source register.

B.4.7 Reduction

vector reduce: sum

```
srd ← 0
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        srd ← srd + vra[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
srd	reduced value
vra	vector of source values

vector reduce: minimum

```
first ← true
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        if first then
            srd ← vra[i]
            first ← false
        end if
        if vra[i] < srd then
            srd ← vra[i]
        end if
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
srd	reduced value
vra	vector of source values

vector reduce: maximum

```
first ← true
for (i ← 0,  i<vlen,  i ← i+1) do
    if ¬mask or (mask and mask[i]) then
        if first then
            srd ← vra[i]
            first ← false
        end if
        if vra[i] > srd then
            srd ← vra[i]
        end if
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	(optional) mask register
srd	reduced value
vra	vector of source values

B.4.8 CAM-Based

Vector Prior Instances

```
for (i ← 0,  i<vlen,  i ← i+1) do
    vrd[i] ← 0
    for (j ← 0,  j<i,  j ← j+1) do
        if vra[i] = vra[j] then
            vrd[i] ← vrd[i]+1
        end if
    end for
end for
```

vlen	vector length
dtype	datatype of values
vrd	destination vector
vra	vector of values

Vector Group Aggregate: Sum

```
for (i ← 0,  i<vlen,  i ← i+1) do
    vrd[i]=0
    for (j ← 0,  j≤i,  j ← j+1) do
        if vra[i] = vra[j] then
            vrd[i] ← vrd[i]+vrb[j]
        end if
    end for
end for
```

vlen	vector length
dtype	datatype of values
vrd	destination vector
vra	vector of group keys
vrb	vector of values to reduce

Vector Last Unique

```
for (i ← vlen−1,  i≥0,  i ← i−1) do
    found ← false
    for (j ← i+1,  j<vlen,  j ← j+1) do
        if  vra[i] = vra[j]  then
            found ← true
        end if
    end for
    md[i] ← ¬found
end for
```

vlen	vector length
dtype	datatype of values
md	destination mask
vra	vector of values
vrb	result of VPU/VGAsum

– Currently, our implementation requires performing VPI or VGAsum before calculating VLU.

B.4.9 Miscellaneous

vector merge

```
for (i ← 0,  i<vlen,  i ← i+1) do
    if  mask[i]  then
        vrd[i] ← vrb[i]
    else
        vrd[i] ← vra[i]
    end if
end for
```

vlen	vector length
dtype	datatype of values
mask	mask register
vrd	destination vector
vra	1st vector of source values
vrb	2nd vector of source values

vector get element

```
srd ← vra[sra]
```

vlen	vector length
srd	destination register
vra	vector of source values
sra	position in vector register

– Undefined behaviour when position is not in range.

vector set element

```
vrd[srb] ← sra
```

vlen	vector length
vrd	destination vector
sra	value to write
srb	position in vector register

– Undefined behaviour when position is not in range.

B.4.10 Mask Manipulation

mask set all

```
for (i ← 0, i<maxvlen, i ← i+1) do
    maskd[i] ← 1
end for
```

maskd	destination mask

mask clear all

```
for (i ← 0, i<maxvlen, i ← i+1) do
    maskd[i] ← 0
end for
```

maskd	destination mask

mask-mask and

```
for (i ← 0, i<maxvlen, i ← i+1) do
    maskd[i] ← maska[i] & maskb[i]
end for
```

maskd	destination mask
maska	first source mask
maskb	second source mask

mask-mask or

```
for (i ← 0, i<maxvlen, i ← i+1) do
    maskd[i] ← maska[i] | maskb[i]
end for
```

maskd	destination mask
maska	first source mask
maskb	second source mask

mask not

```
for (i ← 0, i<maxvlen, i ← i+1) do
    maskd[i] ← ¬maska[i]
end for
```

maskd	destination mask
maska	source mask

mask population count

```
srd ← 0
for (i ← 0, i<vlen, i ← i+1) do
    if mask[i] then
        srd ← srd+1
    end if
end for
```

vlen	vector length
mask	source mask
srd	destination scalar register

B.4.11 Vector Length

vector length: set maximum

```
vlen ← maxvlen
```

vector length: set

vlen ← sra	sra	source value

vector length: get

srd ← vlen	srd	destination register

List of Figures

List of Tables

Bibliography

[AANS$^+$14] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal, and Mikel Luján. An Empirical Evaluation of High-level Synthesis Languages and Tools for Database Acceleration. In *24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014. Referenced on 6.

[ABH$^+$13] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. Referenced on 40.

[ABS$^+$07] Dennis Abts, Abdulla Bataineh, Steve Scott, Greg Faanes, Jim Schwarzmeier, Eric Lundberg, Tim Johnson, Mike Bye, and Gerald Schwoerer. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 17:1–17:12, 2007. Referenced on 16.

[Act11] Actian. Vectorwise. Record Breaking Action Engine for Big Data. `http://www.actian.com/products/vectorwise`, 2011. Referenced on 11.

[AED05] Jung Ho Ahn, Mattan Erez, and William J. Dally. Scatter-Add in Data Parallel Architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 132–142, 2005. Referenced on 58, 83.

[Asa98] Krste Asanović. *Vector Microprocessors.* PhD thesis, EECS Department, University of California, Berkeley, 1998. Referenced on 5, 16, 133.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, 1968. Referenced on 7, 43.

[BBK+68] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, Aug 1968. Referenced on 3.

[BFGS12] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 181–192, 2012. Referenced on 51.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, 1999. Referenced on 1, 5.

[BNE14] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, volume 8391 of *Lecture Notes in Computer Science*, pages 61–76. Springer International Publishing, 2014. Referenced on 61.

[Boh07] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, 2007. Referenced on 2.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, volume 5 of *CIDR*, pages 225–237, 2005. Referenced on 6, 11, 34.

[CAGM04] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering*, pages 116–127, 2004. Referenced on 32, 35.

[Cat11] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):12–27, May 2011. Referenced on 106.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan Primitives for Vector Computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 666–675, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. Referenced on 102.

[CEL+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The Reconfigurable Streaming Vector Processor (RSVP$^{\text{TM}}$). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '03, pages 141–150. IEEE Computer Society, 2003. Referenced on 101.

[CK85] George P. Copeland and Setrag N. Khoshafian. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 268–279, 1985. Referenced on 11, 40.

[CML14] Min Chen, Shiwen Mao, and Yunhao Liu. Big Data: A Survey. *Mobile Networks and Applications*, 19(2):171–209, 2014. Referenced on 1, 61.

[CNL$^+$08] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, August 2008. Referenced on 6, 37, 43.

[CR07] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 339–350. VLDB Endowment, 2007. Referenced on 64, 65.

[Cra96] Harvey G. Cragon. *Memory Systems and Pipelined Processors*. Jones & Bartlett Publishers, Inc., 1996. Referenced on 3.

[CVE99] Jesus Corbal, Mateo Valero, and Roger Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO '99, pages 72–79, 1999. Referenced on 101.

[DDHS00] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, Mar 2000. Referenced on 5.

[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. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974. Referenced on 2.

[EAE$^+$02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 281–292, 2002. Referenced on 16, 18.

[EVS97] Roger Espasa, Mateo Valero, and James E. Smith. Out-of-Order Vector Architectures. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO '97, pages 160–170, 1997. Referenced on 16, 18, 23.

[EVS98] Roger Espasa, Mateo Valero, and James E. Smith. Vector Architectures: Past, Present and Future. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, pages 425–432, 1998. Referenced on 3, 16.

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966. Referenced on 2.

[Fly72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972. Referenced on 3.

[Fog12] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs/An optimization guide for assembly programmers and compiler makers, 2012. Referenced on 22.

[FVS$^+$13] Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An Empirical Study of Intel Xeon Phi. *CoRR*, abs/1310.5842, 2013. Referenced on 44.

[GBY07] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: High Performance Sorting on the Cell Processor. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1286–1297, 2007. Referenced on 6, 37, 43.

[GGK$^+$83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers*, 100(2):175–189, 1983. Referenced on 51.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *International Conference on Management of Data*, SIGMOD, pages 325–336, 2006. Referenced on 6.

[GH11] Chris Gregg and Kim Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 134–144, 2011. Referenced on 6.

[GHF$^+$06] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006. Referenced on 34.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, April 1993. Referenced on 13.

[GP07] Joe Gebis and David Patterson. Embracing and Extending 20th-Century Instruction Set Architectures. *Computer*, 40(4):68–75, 2007. Referenced on 2.

[GVTP97] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating Cache Conflict Misses Through XOR-based Placement Functions. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 76–83. ACM, 1997. Referenced on 63.

[Hab72] A. Nico Habermann. Parallel neighbor-sort (or the glory of the induction principle). Technical report, Carnegie Mellon University, 1972. Referenced on 42.

[Hal56] A. C. D. Haley. DEUCE: a High-speed General-purpose Computer. *Proceedings of the IEE - Part B: Radio and Electronic Engineering*, 103(2):165–173, April 1956. Referenced on 2.

[HB09] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan and Claypool Publishers, 1st edition, 2009. Referenced on 2.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers Two: Architecture, Programming and Algorithms.* IOP Publishing Ltd., Bristol, UK, 2nd edition, 1988. Referenced on 2.

[HLY$^+$09] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems*, 34(4):21:1–21:39, 2009. Referenced on 6, 16, 34.

[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*, DaMoN, pages 4:1–4:6, 2007. Referenced on 6, 34.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962. Referenced on 40.

[HP12] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Elsevier, 2012. Referenced on 2, 133.

[HSU$^+$01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. The Microarchitecture of the Pentium® 4 Processor. In *Intel Technology Journal.* Citeseer, 2001. Referenced on 5.

[HT72] R. G. Hintz and D. P. Tate. Control data STAR-100 processor design. In *Compcon72 Sixth Annual IEEE Computer Society International Conference*, pages 1–4. IEEE Computer Society, 1972. Referenced on 3.

[IMKN07] Hiroshi Inoue, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani. AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 189–198, 2007. Referenced on 6, 37, 43.

[Ing09] Ingres. Ingres/VectorWise Sneak Preview on the Intel Xeon Processor 5500 Series-Based Platform. white paper, 2009. Referenced on 14.

[Int11] International Data Corporation. Worldwide Server Market Accelerates Sharply in Fourth Quarter as Demand for Heterogeneous Platforms Leads the Way, According to IDC. `http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22716111`, February 2011. Accessed on 2011-09-08. Referenced on 14.

[Int14a] Intel. *Intel®64 and IA-32 Architectures Optimization Reference Manual*, March 2014. Referenced on 15, 22.

[Int14b] Intel. *Intel®Architecture Instruction Set Extensions Programming Reference*, March 2014. Referenced on 5, 58, 84.

[JK12] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *Symposium on VLSI Technology (VLSIT)*, pages 87–88, June 2012. Referenced on 106.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 1st edition, 2007. Referenced on 23, 30.

[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. Referenced on 6, 33.

[KKS$^+$08] Sanjeev Kumar, Daehyun Kim, Mikhail Smelyanskiy, Yen-Kuang Chen, Jatin Chhugani, Christopher J. Hughes, Changkyu Kim, Victor W. Lee, and Anthony D. Nguyen. Atomic Vector Operations on Chip Multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 441–452, 2008. Referenced on 58, 84, 103.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison Wesley Longman Publishing Co., Inc., 1998. Referenced on 45.

[Kog81] Peter M. Kogge. *The Architecture of Pipelined Computers*. Hemisphere Publishing Corporation, 1981. Referenced on 3, 16, 63.

[LAB$^+$11] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 129–140, 2011. Referenced on 2, 5, 96.

[LAS$^+$09] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '09, pages 469–480. IEEE, 2009. Referenced on 87.

[LBE$^+$98] Jack L Lo, Luiz André Barroso, Susan J Eggers, Kourosh Gharachorloo, Henry M Levy, and Sujay S Parekh. An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 39–50. IEEE Computer Society, 1998. Referenced on 6.

[LCA$^+$11] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *International Conference on Computer-Aided Design*, ICCAD, pages 694–701, 2011. Referenced on 54.

[Lee95] Ruby B. Lee. Accelerating Multimedia with Enhanced Microprocessors. *IEEE Micro*, 15(2):22–32, April 1995. Referenced on 4.

[Lev90] Stewart A Levin. A fully vectorized quicksort. *Parallel computing*, 16(2):369–373, 1990. Referenced on 37, 42.

[LGBT05] C.C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design Test of Computers, IEEE*, 22(6):556–564, Nov 2005. Referenced on 106.

[LKC$^+$10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460, 2010. Referenced on 6, 51.

[LSCJ06] Christophe Lemuet, Jack Sampson, Jean-Francois Collard, and Norm Jouppi. The Potential Energy Efficiency of Vector Acceleration. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. ACM, 2006. Referenced on 5, 96.

[Mar96] Rich Martin. A Vectorized Hash-Join. IRAM technical report, University of California at Berkeley, 1996. Referenced on 33.

[MCB$^+$11] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H. Byers. *Big data: The next frontier for innovation, competition, and productivity.* McKinsey Global Institute, 2011. Referenced on 1, 61.

[MHIT14] Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara. The Brand-New Vector Supercomputer, SX-ACE. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 199–214, New York, NY, USA, 2014. Springer-Verlag New York, Inc. Referenced on 103.

[MK00] Shintaro Meki and Yahiko Kambayashi. Acceleration of Relational Database Operations on Vector Processors. *Systems and Computers in Japan*, 31(8):79–88, 2000. Referenced on 16, 34.

[Moo65] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965. Referenced on 2.

[MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009. Referenced on 6.

[NH83] Lionel M. Ni and Kai Hwang. Vector reduction methods for arithmetic pipelines. In *IEEE 6th Symposium on Computer Arithmetic (ARITH)*, pages 144–150, June 1983. Referenced on 63.

[OFW99] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48, Mar 1999. Referenced on 5.

[PA11] Davide Pasetto and Albert Akhriev. A Comparative Study of Parallel Sort Algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA, pages 203–204, 2011. Referenced on 6.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 206–218, 1997. Referenced on 5, 12, 16.

[PLH$^+$15] Jason Power, Yinan Li, Mark D. Hill, Jignesh M. Patel, and David A. Wood. Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN, page 11. ACM, 2015. Referenced on 83.

[PM12] Matt Pharr and William R. Mark. ispc: A SPMD Compiler for High-Performance CPU Programming. In *Innovative Parallel Computing*, InPar, pages 1–13, 2012. Referenced on 6.

[PR13] Orestis Polychroniou and Kenneth A. Ross. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 6:1–6:6. ACM, 2013. Referenced on 6, 64, 82.

[PW96] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, Aug 1996. Referenced on 5.

[QCEV99] Francisca Quintana, Jesus Corbal, Roger Espasa, and Mateo Valero. Adding a Vector Unit to a Superscalar Processor. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 1–10, 1999. Referenced on 17, 27.

[Rau91] B. Ramakrishna Rau. Pseudo-randomly Interleaved Memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 74–83. ACM, 1991. Referenced on 63.

[RCBJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architiure Letters*, 10(1):16–19, January 2011. Referenced on 21.

[RGAB98] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André Barroso. Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors. In *ACM SIGPLAN Notices*, volume 33, pages 307–318. ACM, 1998. Referenced on 6.

[Rus78] Richard M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, January 1978. Referenced on 3.

[SBF$^+$12] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 68–70, 2012. Referenced on 51.

[SBM62] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. The SOLOMON Computer. In *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*, AFIPS '62 (Fall), pages 97–107, New York, NY, USA, 1962. ACM. Referenced on 3.

[Sch87a] Paul B. Schneck. *Supercomputer Architecture*, volume 31 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1987. Referenced on 3.

[Sch87b] W. Schönauer. *Scientific Computing on Vector Computers*. Elsevier Science Publisher B.V., 1987. Referenced on 3, 29.

[Sed78] Robert Sedgewick. Implementing Quicksort Programs. *Communications of the ACM*, 21(10):847–857, October 1978. Referenced on 40.

[Sen67] D. N. Senzig. Observations on High-performance Machines. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, AFIPS '67 (Fall), pages 791–799, New York, NY, USA, 1967. ACM. Referenced on 3.

[SFS00] J. E. Smith, Greg Faanes, and Rabin Sugumar. Vector Instruction Set Support for Conditional Operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 260–269, 2000. Referenced on 15.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *International Symposium on Parallel & Distributed Processing*, IPDPS, pages 1–10, 2009. Referenced on 6.

[SKC$^+$10] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, 2010. Referenced on 37, 43.

[SS65] D. N. Senzig and R. V. Smith. Computer Organization for Array Processing. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, AFIPS '65 (Fall, part I), pages 117–128, New York, NY, USA, 1965. ACM. Referenced on 3.

[Sta13] JEDEC Standard. High Bandwidth Memory (HBM) DRAM. *JESD235*, 2013. Referenced on 106.

[Sto78] H. S. Stone. Sorting on STAR. *IEEE Transactions on Software Engineering*, 4(2):138–146, March 1978. Referenced on 37, 40, 43, 101.

[Sut05] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. Referenced on 2.

[SZB11] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. Compilation in Query Execution. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*, pages 33–40, 2011. Referenced on 14.

[SZG$^+$09] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer. CMP Memory Modeling: How Much Does Accuracy Matter? In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, pages 24–33, 2009. Referenced on 30.

[Tra11] Transaction Processing Performance Council. TPC-H Standard Specification v2.14.2. `http://www.tpc.org/tpch/`, 2011. Referenced on 12.

[Wat72] W. J. Watson. The TI ASC: A Highly Modular and Flexible Super Computer Architecture. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), pages 221–228, New York, NY, USA, 1972. ACM. Referenced on 3.

[Wil53] J. H. Wilkinson. The pilot ACE. *Automatic Digital Computation*, pages 5–14, 1953. Referenced on 2.

[You07] Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS '07, pages 23–34, 2007. Referenced on 14, 21.

[YRV11] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable Aggregation on Multicore Processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9. ACM, 2011. Referenced on 64, 82.

[Ż09] Marcin Żukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, 2009. Referenced on 14.

[ZB91] Marco Zagha and Guy E. Blelloch. Radix Sort for Vector Multiprocessors. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 712–721, 1991. Referenced on 37, 45.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156. ACM, 2002. Referenced on 6, 16, 34, 82.

Acronyms

AVL Average Vector Length.

AVX Advanced Vector Extensions.

CAM Content-Addressable Memory.

CPI Cycles per Instruction.

CPT Cycles per Tuple.

DBMS Database Management System.

DLP Data-Level Parallelism.

DSS Decision Support System.

FPGA Field-Programmable Gate Array.

GPGPU General-Purpose GPU.

GPU Graphics Processing Unit.

HT Hash Table.

ILP Instruction-Level Parallelism.

IPC Instructions per Cycle.

ISA Instruction Set Architecture.

L1D Level 1 Data (Cache).

L1I Level 1 Instruction (Cache).

L2 Level 2 (Cache).

LHS Left-Hand Side.

LSQ Load/Store Queue.

MC Memory Controller.

MSHR Miss Status Holding Register.

MVL Maximum Vector Length.

NoSQL Not only SQL.

OET Odd-Even Transposition (Sort).

OLAP Online Analytical Processing.

OLTP Online Transaction Processing.

RHS Right-Hand Side.

ROB Reorder Buffer.

SIMD Single Instruction-Multiple Data.

SQL Structured Query Language.

SS Superscalar.

SSE Streaming SIMD Extensions.

TLP Thread-Level Parallelism.

TPC-H Transaction Processing Performance Council: Benchmark H.

VGA Vector Group Aggregate.

VL Vector Length.

VLU Vector Last Unique.

VMRF Vector Memory Request File.

VPI Vector Prior Instances.

VSR Vectorised Serial Radix (Sort).