Decision Support Database Management System Acceleration Using Vector Processors

Timothy Hayes
Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

A thesis submitted for the degree of

Master in Information Technology

September 12, 2011

Director: Eduard Ayguadé Parra

Codirector: Adrián Cristal Kestelman

Tutor: Oscar Palomar Pérez

Abstract

This work takes a top-down approach to accelerating decision support systems (DSS) on x86-64 microprocessors using true vector ISA extensions. First, a state of art DSS database management system (DBMS) is profiled and bottlenecks are identified. From this, the bottlenecked functions are analysed for data-level parallelism and a discussion is given as to why the existing multimedia SIMD extensions (SSE) are not suitable for capturing this parallelism. A vector ISA is derived from what is found to be necessary in these functions; additionally, a complementary microarchitecture is proposed that draws on prior research done in vector microprocessors but is also optimised for the properties found in the profiled application. Finally, the ISA and microarchitecture are implemented and evaluated using a cycle-accurate x86-64 microarchitecture simulator.

Acknowledgements

I would like to thank Adrián Cristal and Osman Unsal, my two supervisors at the Barcelona Supercomputing Center, and Eduard Ayguadé for directing this master's thesis.

This work would not be made possible without the excellent database research being done by the Ingres VectorWise team. I would like to personally thank Peter Boncz, Marcin Zukowski and Bill Maimone for providing us with an academic licence to their software and always answering our questions and providing useful feedback.

I would like to especially thank my tutor Oscar Palomar for all his support and dedication throughout this work. Oscar has a talent for cutting through my scepticism with logical dialogue; my pessimism with insightful discussions; and my nonsense with a patient demeanour. Despite having to complete his PhD in the past year, Oscar found the time to manage five master's students who are now all on their way to becoming doctoral students. We hope you have been as happy tutoring us as we have been being guided by you.

Finally I would like to thank my family and friends, especially all the people who have come into my life since coming to Barcelona; you have made my stay here all the more worthwhile.

Contents

List of Figures

Chapter 1

Introduction

Database management systems (DBMS) have become an essential tool for research and industry and are often a significant component of data centres. They can be used in a multitude of scenarios including online analytical processing [7], data mining, e-commerce and scientific analysis. As the amount of information to manage grows exponentially each year [19], there is a pressure on software and hardware developers to create data centres that can cope with the increasing requirements. At the same time, there is now also an additional demand to provide greener [34] and more power-efficient data centres without compromising their performance [26].

Modern microprocessors often include SIMD multimedia extensions that can be used to exploit data-level parallelism [46, 51]. There has been some prior work [59, 55] using these extensions to accelerate database software. These works were successful to some extent however multimedia extensions tend to be very limited. They often lack the flexibility to describe non-trivial data-level parallelism found in DBMS software. There is generally an upper threshold of two or four elements that can be operated on in parallel and code must be restructured and recompiled if this threshold should increase in the future. The extensions are typically targeted at multimedia and scientific application with the bulk of support geared towards floating point operations. Integer code found in DBMS software was not its intended purpose, this can be a limiting factor to exploit the parallelism in such applications.

VectorWise [9, 27] performed research into hardware-conscious DBMS systems. Using only commodity processors, they identified where previous database solutions were bottlenecked and structured their own software to exploit the full capabilities of a modern superscalar microprocessor. It transformed the data-level parallelism into instruction-level parallelism to keep the processor utilised fully. While the performance is much better, optimising a database by exploiting the data-level parallelism as instruction-level parallelism is not an efficient solution. Modern microprocessors found in servers are generally superscalar/out of order and can achieve this to some extent; the problem is that the hardware complexity and power consumption of finding more independent instructions this way increases quadratically [42] making this an unscalable solution and not suitable in the long term.

Vector architectures [16] are highly scalable and overcome the limitations imposed by superscalar design as well as SIMD multimedia extensions. They have the flexibility to represent complex data-level parallelism where multimedia extensions are lacking. More importantly, they are energy-efficient [35] and can be implemented on-chip using simple and efficient hardware [3]. In the past, vector processors were also used to tolerate long latencies to main memory. As memory latency has become a significant issue for both computer architects and software developers, vector support could be instrumental in optimising databases which are typically memory-bound [8]. Vector architectures have traditionally been used for scientific applications with floating-point code, however it is felt that a lot of the concepts can be applied to business-domain (integer) applications too.

This work takes a top-down methodology and profiles VectorWise, a database management system optimised for modern out of order superscalar microprocessors, looking for opportunities to use vector technology. From this, software bottlenecks caused by unscalable scalar hardware structures are identified. A discussion is given as to why multimedia instruction sets such as SSE are insufficient to express the potential data-level parallelism in this application; consequently, new vector ISA extensions for x86-64 are proposed as a solution. These ISA extensions have been implemented using a cycle-accurate microprocessor simulator and experiments show speedups in performance between **1.3x** and **6.2x** with the possibility to scale further.

This document is structured as follows. Chapter 2 provides a short overview of technical details relevant to this work. Chapter 3 is an exploration section that analyses VectorWise and discusses bottlenecks due to being run on a scalar processor. Chapter 4 proposes a vector ISA and complementary microarchitecture based on the results of the exploration section. Chapter 5 discusses the issues, choices and decisions of implementing the design using a cycle-accurate microprocessor simulator. Chapter 6 shows experiments and results of running a vectorised version of the application on the modified simulator. Chapter 7 discusses related work regarding database acceleration and vector microprocessors. Finally, chapter 8 concludes this document and hints at ideas for future work.

Chapter 2

Technical Background

This chapter gives a short overview of technical concepts used in this work. It is not intended to be exhaustive and references are given for further reading.

2.1 Vector Architectures

Vector architectures [16] have been around since the early 1970s and were traditionally used in the supercomputer market. They provide instruction set architectures (ISAs) to express data-level parallelism concisely. The main advantages are listed here, but for a comprehensive overview of vector architectures the reader is referred to [24] and [3].

Vector hardware is typically much more simple than their aggressive out of order superscalar equivalents and yet can provide strong computational performance when the application is suitable (vectorisable). Parallelism can be exploited through pipelining or through parallel vector lanes.

Vector architectures have a reputation of being more energy efficient [35] and scalable over scalar microprocessors. There can be a large reduction in the pipeline's fetch and decode stages and scalability is generally not inhibited by associative hardware structure as with out of order superscalar microarchitectures [42].

Vector architectures also have the potential to reduce the penalties of branch mispredictions by predicating operations using masks. It is even possible to accelerate performance further by completely bypassing blocks of masked operands [52]. Additionally, there is less requirement for aggressive speculation and memory prefetching since execution and memory access patterns are fully encoded into a single instruction.

2.1.1 Vectors vs. Multimedia Extensions

Streaming SIMD Extensions [46], the multimedia extensions offered in all the current x86/x86-64, microprocessors can be considered vectors but with limitations. True vectors offer several advantages not found in these multimedia extensions.

At the time of this writing, the latest multimedia extensions offered by Intel are the Advanced Vector Extensions (AVX) [29] which offers sixteen 256-bit SIMD registers. 256 bits can contain four long integers or four double-precision floats. Vectors typically have larger registers which can hold dozens to hundreds of elements.

Traditional vector architectures typically include a programmable vector length register. In the SIMD multimedia extensions, this length is fixed. There is a consequence of this: if the maximum vector length can be detected at runtime through an instruction, manufacturers can vary this length whilst keeping the ISA the same. In contrast, with every generation of the SIMD multimedia extensions that have increased the register width, code must be rewritten and recompiled to take advantage of it.

One the strongest advantages of vector processors is their ability to tolerate long latency instructions, of particular important memory operations. In contrast, the SIMD multimedia extensions have a much more limited access to memory. As an example, Sandy Bridge [28], a microarchitecture that implements AVX, provides a 16-byte bus from the SIMD registers to the level 1 cache. If there is a cache miss, the instruction is treated the same as any scalar load that misses cache. The benefit is that the bandwidth is doubled, but only if the working set is cache resident.

Historically, vector architectures fully pipelined operations through its functional units to hide execution latency. SIMD multimedia extensions are instead executed using very wide functional units. The functional units can be fully pipelined, but with respect to unique instructions rather than elements within the same instruction. Vector architectures have often employed chaining to pipe the output of one instruction into the input of another instruction before the former fully completes. SIMD multimedia extensions don't do this because they don't fully pipeline the elements of their inputs. In fact, their register widths are too short take advantage of chaining.

Vector architectures have often included a separate set of architectural registers to enable masking. Masking allows turning conditional branches into predicated instructions thus eliminating many conditional branches. The SIMD multimedia extensions has some support for masking, although not through separate registers, and is very limited. The most lacking feature in the SIMD multimedia extensions' idea of masking is that it cannot conditionally execute parts of a memory operation. This has changed recently in AVX which allows a limited form of masked load/store instructions.

Vector architectures have had more flexibility when it comes to memory instructions. Vector machines typically support unit-stride memory accesses, strided memory accesses and indexed memory accesses. The SIMD multimedia extensions require that all data be stored in contiguous locations in memory.

Vector architectures have usually included more flexible instructions than those found in the SIMD multimedia extensions. For example, the compress instruction found in many vector architectures [52] allows an input vector register to be rearranged based on a mask. The SIMD multimedia extensions have some position manipulation instructions, but are generally not as powerful as those in their vector counterparts.

Finally, some vector architectures have included support for reductions operations. These allow a vector of values to be aggregated into a single scalar value. The SIMD multimedia extensions offer no equivalent to this.

2.2 Databases

2.2.1 Database Categories

Databases Management Systems (DBMSs) can be broadly categorised into two main categories:

1. Online Transaction Processing (OLTP)
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 typically long and compute-intensive queries. Typically a DBMS will be implemented and optimised targeting either OLTP or OLAP, furthermore 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.

2.2.2 Query Engines

The query engine is the heart of a DBMS. There are three primary ways a query engine can operate on data when executing a query:

1. Tuple at a time processing
2. Column-wise processing
3. Block at a time processing

Tuple at a time processing, also known as the volcano model [18], is a technique to implement a query engine by processing tuples, i.e. rows, one at a time. This essentially means that in a database query, a single row traverses the entire query tree tree before the next row is processed. The primary advantage is that as a row traverses the query tree, there is a lot of temporal locality as the output of one operator is the input to the next one. This can be beneficial for microprocessors with a cache hierarchy. The primary disadvantage is that for every row that passes through every operator in the query tree, there is a function call. This can lead to a large function call overhead and poor performance due to a lack of regularity in the code e.g. no loops.

Column-wise processing [7] is almost the exact opposite. Instead of traversing the query tree one row at a time, all of the rows are sent through a single operator of the query tree together. The output is pruned, stored back into memory and the process starts again for the next operator in the query tree. The advantage here is that the function overhead can be reduced considerably. For all the rows processed, the operator only needs to be called once. The advantage here is that the function call overhead is reduced and there is a lot of regularity in the data access patterns. The disadvantage is that for large databases, the input and temporary results must come from and go to main memory. Main memory is orders of magnitude slower than the cache hierarchy. Data is typically stored in columns [12, 61] rather than rows for accelerated I/O access and may consequently reveal more data-level parallelism to the functions, depending on the implementation.

Block at a time processing [41] is a novel implementation that takes the middle ground between tuple at a time and column-wise processing. Instead of operating on one row at a time per function like the former, it operates on a cache-resident block of rows per function. This means that intermediate results need not go to main memory and remain in the cache between operators. Additionally, the per-operator function call can be amortised if the block size is large enough.

2.2.3 VectorWise

VectorWise [27] is a block at a time query engine based MonetDB/x100 [9]. It is highly optimised for modern superscalar processors. Its operator functions are designed to be instruction cache friendly and as well as to reduce branch misprediction penalties. Its block at a time technique is highly optimised for the data cache [50]. Its functions are written to be data-level parallel and try to expose independent loop iterations to the underlying out of order execution core. SSE intrinsics are used in some parts, but the majority of functions are left to the compiler to autovectorise.

With respect to blocks and memory, VectorWise can store a table in a columnar fashion [12]. This means that arrays in memory contain columns of a table. When the functons access data like this, it can help expose data-level parallelism and generate better hardware prefetching requests.

Chapter 3

Exploration

This chapter describes the process of profiling VectorWise in order to find computational bottlenecks. Later, an analysis is given on the most significant functions. Potential data-level parallelism is identified and a discussion is given as to why SSE is often insufficient to capture this parallelism. This exploration serves as an input to defining an instruction set architecture (ISA) and influences many decisions made in the design phase of the vector microarchitecture.

3.1 Baseline

3.1.1 VectorWise

For our experiments we use a standalone edition of x100 [9] that ultimately serves as the query engine of VectorWise v1.0 [27]. The software is compiled with the Intel C Compiler (ICC) v12. The specific optimisations are chosen by VectorWise's configure script and include an optimisation level of `-O6`.

Although VectorWise offers multithreaded query execution, the objective here is to observe per-thread performance. For this reason benchmarks have been limited to a single thread. It should be noted that while the query-related computations are being performed in one thread, there are still additional helper threads in VectorWise that must co-exist e.g. there is a separate thread in charge of bringing data from disk and uncompressing it.

In all our tests a block length of 1024 is used i.e. each database operator processes 1024 elements before passing the results to the next operator in the query tree.

3.1.2 TPC-H

The standard benchmark used for decision support databases is the Transaction Processing Performance Council benchmark H (TPC-H) [54]. TPC-H consists of 22 ad-hoc queries posing complex business-oriented questions on a large relational database. The benchmark also defines the schema of the database whereas content itself is dummy data generated by the council's official DBGEN tool.

The size of database is determined by a scale factor number; a scale factor of 1 represents an input set of 1 GB however when placed within the database it could be more or less depending on how the vendor decides to store the information. Some databases have a lot of redundancy e.g. storing tables multiple times sorted on different columns; other databases reduce the amount of space required by compressing the tables on disk and only decompressing them on demand. Even with redundancy or compression, the database is expected to stay in the same order of magnitude of its input size.

Typical scale factors are powers of 10, starting from 1 GB and growing as high as 30,000 GB. For our experiments we have chosen 100 GB as 10 GB did not stress the system enough and 1,000 GB would need more disk space than is available on our system.

3.1.3 System Setup

The following is a description of the hardware used to profile VectorWise. It contains the most elemental properties of the system; a more comprehensive and detailed look at the microarchitecture is given in chapter 6.

Microprocessor

All our experiments were conducted on two Intel Xeon X5650s clocked 2.67 GHz. The Xeon's microarchitecture is Westmere which is a derivative of Nehalem fabricated at 32 nm. Each chip multiprocessor (CMP) contains six cores which in turn contain two thread contexts that can perform simultaneous multithreading (SMT). The ISA used is x86-64 with multimedia extensions up to and including SSE4.2.

Memory

There is a three-tiered cache hierarchy per chip with the following properties:

- L3 cache: shared 2 MB local per core and 12 MB overall
- L2 cache: private 256 KB
- L1 data cache: private 64 KB
- L1 instruction cache: private 64 KB

This is backed by a total of 16 GB of main memory. The system has four DIMMs of 4 GB DDR3 1333 memory. These are placed into slots 1, 2, 5 and 6 giving each microprocessor chip local access to two modules each. This configuration is balanced and economical but it does not take advantage of the potential triple-channel memory access that is possible with the Westmere architecture. This would require two extra memory DIMMs located slots 3 and 6.

Disk

The properties of the disk are not given intentionally as these experiments are focused purely on microprocessor utilization; I/O performance is an orthogonal study. Three techniques for reducing I/O impact are: Redundant Array of Independent Disks (RAID) [10]; VectorWise's own ColumnBM buffer manager [9]; and more recently there have been studies into utilizing solid state drives in databases [5].

It should be noted that although the disk performance did not concern these experiments, the disk capacity limited the possible TPC-H scale factors. A scale factor of 100 GB is used as it is the largest size using power of 10 that can be contained on the disk before its capacity becomes a problem.

Operating System

All experiments have been carried out using Fedora 13 x86-64.

3.2 Profiling

In order to get a feeling for the presence and location of bottlenecks, a combination of techniques is employed. First each of the TPC-H queries' CPU usage is measured. This is followed by a breakdown by database operation of the longest query. Finally hardware counters are used to look at the performance of important functions in more detail.

3.2.1 Query Timings

Each query of the TPC-H suite was measured individually for CPU execution time. Intel's VTune Amplifier XE [30] has been used instead of the typical BASH time utility. This is because, as mentioned in subsection 3.1.1, VectorWise is multithreaded and it is necessary to single out the thread assigned to the query's execution. This is made possible by using VTune which has an advanced thread filtering mechanism.

[17] discusses the impact of TLB misses on DBMS software and how to reduce their number by making use of HugeTLBfs (huge pages). VectorWise can optionally use HugeTLBfs if it is supported by the underlying microprocessor. To show how much performance is gained by reducing the TLB miss overhead, all queries are run with and without HugeTLBfs support. Showing that the number of TLB misses can be minimised is an important fact later used in chapter 5.

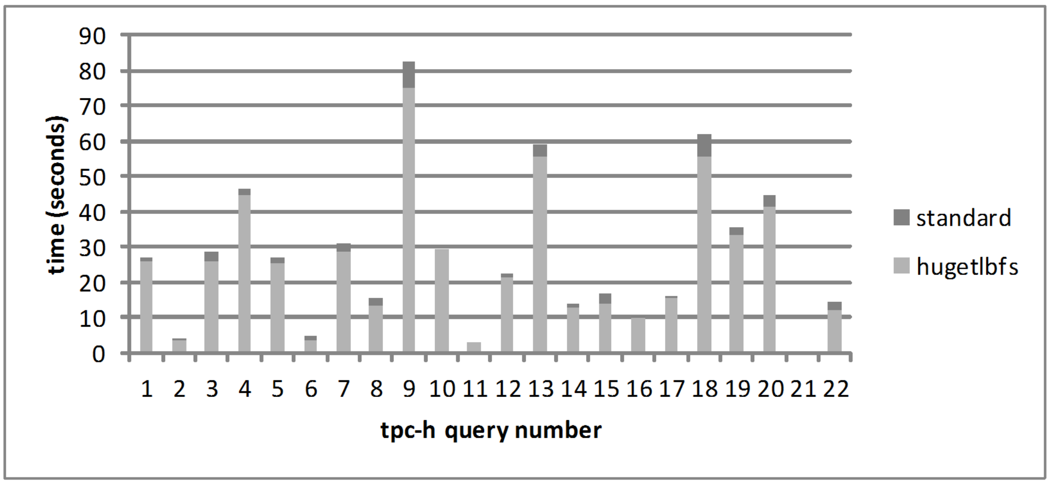


Figure 3.1: TPC-H SF 100 query CPU times

Figure 3.1 shows each query's time spent in the CPU. Of the 22 queries, one of them (21) did not complete successfully due to additional memory requirements. It has been subsequently omitted from further experiments.

It is clear that TLB misses can be a significant contributor to avoidable overhead and has been significantly reduced by using HugeTLBfs. The longest query is number nine which takes 75 seconds of CPU time when HugeTLBfs is enabled. Due to its significance relative to the other times, this query has been chosen for a more detailed analysis.

3.2.2 TPC-H Query Nine

VectorWise is written such that it can optionally profile itself with minimal overhead. Its mechanism uses the x86-64 time stamp counter (RDTSC) to record the current cycle number on entry to and exit from database operators. From this, a difference can be obtained in order to show how much time is spent in any particular operator. The data collected from this profiler has been used to generate a graph of query nine's query tree including a percentage of time spent in each operator.

Figure 3.2 shows a breakdown of query nine. The rectangular boxes represent the input tables of the query. The striped horizontal rectangles display the table name whereas the vertical rectangles display the name of each of the columns needed in the query. The rounded rectangles illustrate an operator of the query tree. These can be unary (single input) or binary (two inputs) and always have one output keeping in line with relational algebra rules. Each operator also contains a percentage of the overall CPU time which is also shown visually by varying the intensity of its fill colour. There are also directed lines connecting the inputs and operators together. Each line has an associated number; this value is the number of rows (or tuples) flowing from one node to the next.

From the diagram it can be seen that of the nine operators of the query tree, five of these are hash joins. These five hash joins account for 93.74% of the total CPU time. Of particular interest is the hash join that takes 63.49% of the CPU time. This is very large operation joining 600 M rows on the left-hand side (LHS) on a hash table containing more than 4.5 M rows. The operator itself uses two columns (keys) for its join criteria i.e. two rows can only be joined if the left-hand side and right-hand side (RHS) both contain matching `partkey` and `suppkey` values.

It is clear that hash join is an important operator in query nine, but what about the other queries? Figure 3.3 shows a percentage of time spent in a hash join operator for each query in TPC-H. Only queries one, six and fifteen don't use a hash join and for the remaining queries, it is evident that it is a very significant function. For six queries alone, it takes more than 90% of the execution.

It is desirable at this point to focus on the hash join operator. The operation itself can be categorised further by describing it as two phases. In order for a hash join to work, first the hash

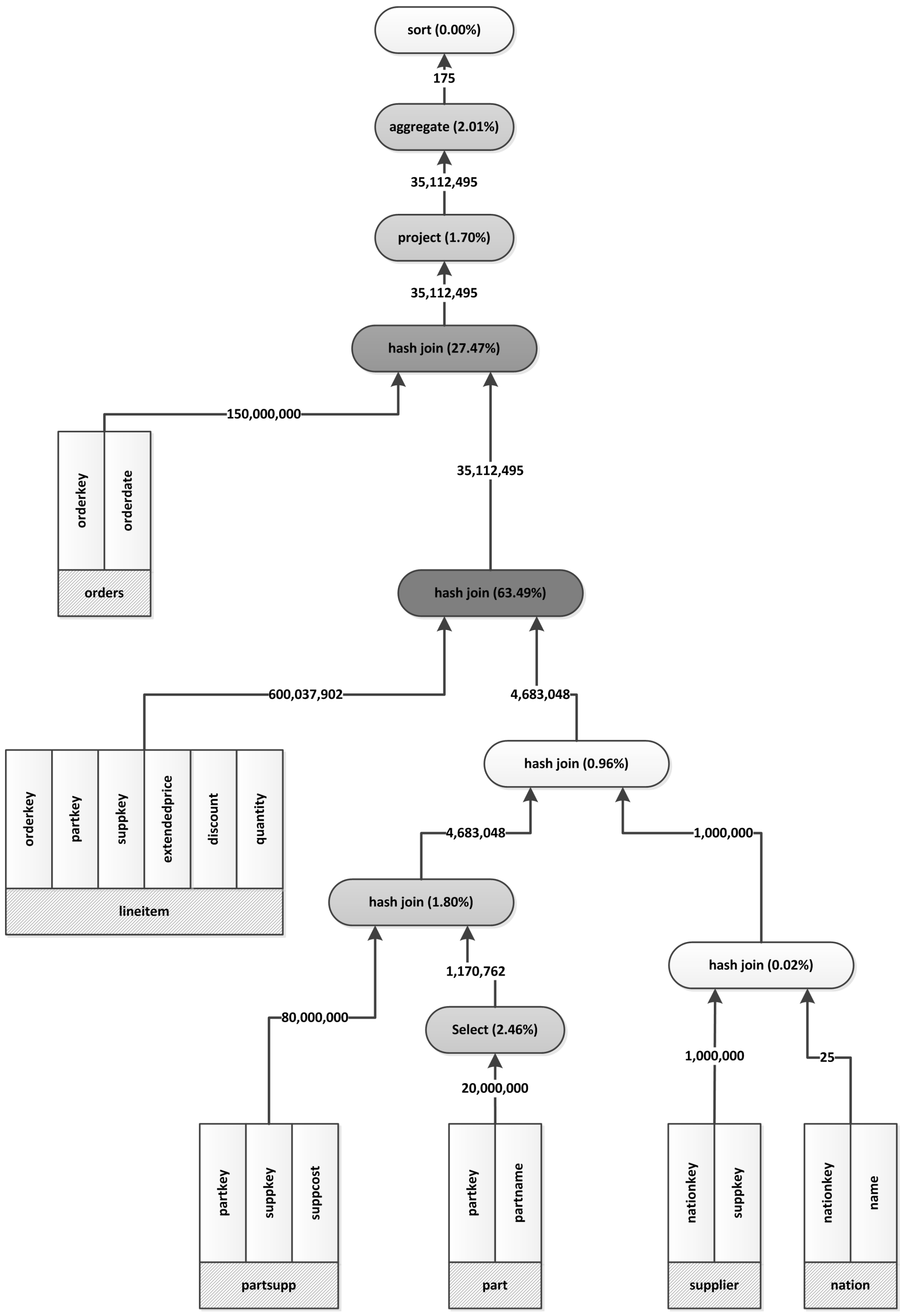


Figure 3.2: TPC-H query nine breakdown

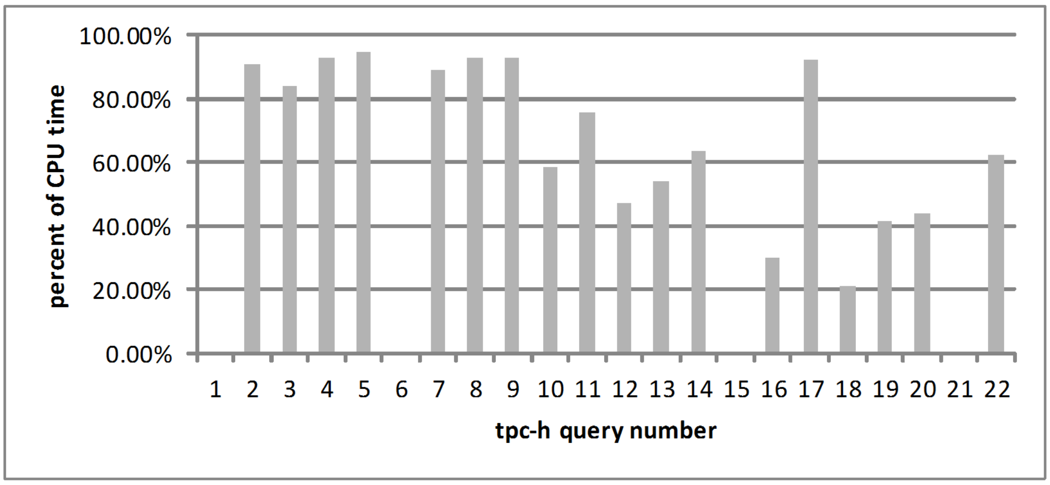


Figure 3.3: TPC-H SF 100 percentage of CPU time spent in hash join

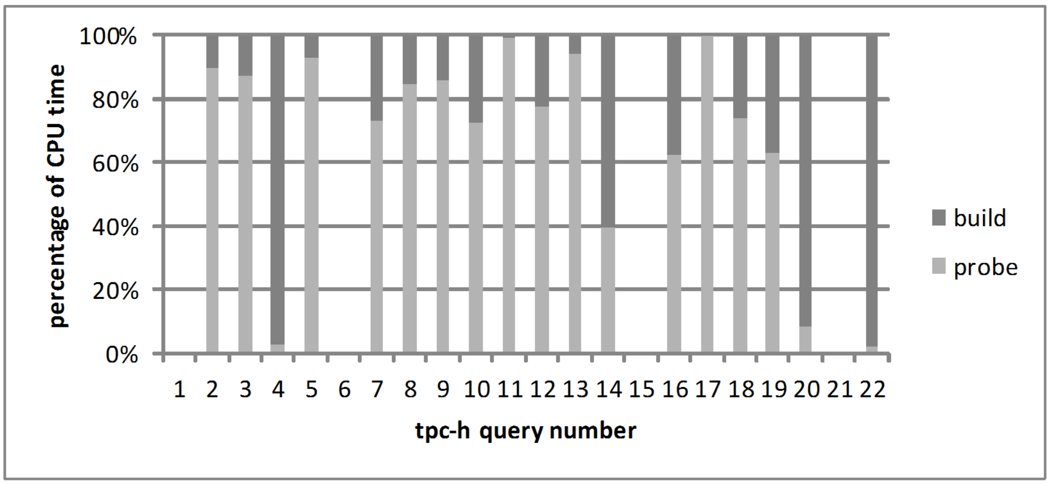


Figure 3.4: TPC-H SF 100 hash join phase breakdowns

table must be constructed using inputs from the RHS; this is called the build phase. After, the table can be queried with values from the LHS; this is called the probe phase. Figure 3.4 shows a breakdown of each query's hash joins by build phase and probe phase. Overall the probe phase is significantly more dominant than the build phase; the notable exceptions to this are queries four, twenty and twenty-two. These three queries are using variants of the join operator called semi-join and anti-join. In both cases the RHS has a larger input than the LHS making the build phase more substantial.

3.2.3 Hash Probing

It has been shown that the probe phase of the hash join operator is very significant and warrants further investigation. This section first explains the VectorWise implementation of hash probing. After, more profiling is performed at the function level using hardware counters. From this, possible reasons for its high CPU usage are given.

Step by Step

To explain the functionality of VectorWise's hash probing mechanism, the dominant join of query nine is used as a reference example. The process is broken into discrete steps which are in turn explained diagrammatically.

The function definitions have been simplified slightly into C-like pseudocode for clarity. Many of the following functions are not written by hand but instead generated from a generic description of the task to be done. This way many variations of the function can be created for different data types. This is analogous to templates used by C++. For the sake of readability, all integer-based data types are represented in the pseudocode using the `int` keyword.

Step 1 The first step is to take the keys on the LHS table and transform them into something that can index into the hash table. Since the query uses two keys in the join, both need to be inputs to the hash function.

```
void hash(int N, int* res, int* col1)
{
  for (i=0; i<N; i++)
  {
    res[i] = HASH(col1[i]);
  }
}
```

Listing 3.1: code for hash()

Figure 3.5 shows how two columns, each containing a key, can be combined into one hash value. First a hash is made from `partkey` using the function `hash()` found in listing 3.1. After, a new hash is generated by combining the first hash and the values from `suppkey` using the function `rehash()` found in listing 3.2. The resulting hash is stored in the temporary space `T0`. The `HASH` macro is omitted for simplicity however it is a simple routine based on prime number multiplication and logical bitwise operations.

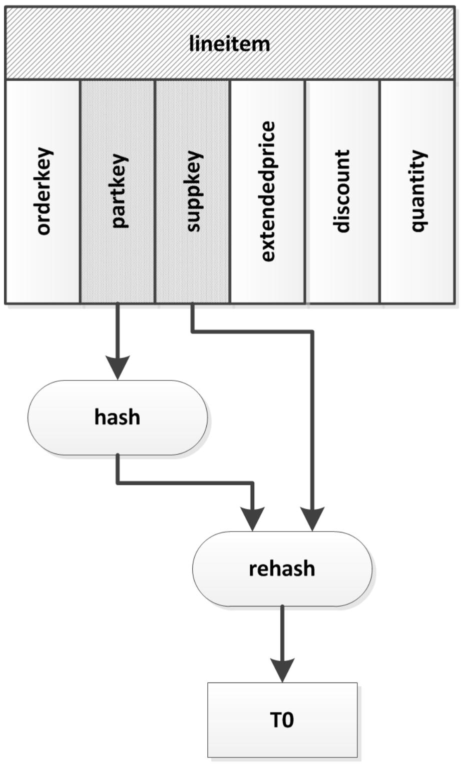


Figure 3.5: hash probing: step one

```
void rehash(int N, int* res, int* col1, int* col2)
{
  for (i=0; i<N; i++)
  {
    res[i] = col1[i] ^ HASH(col2[i]);
  }
}
```

Listing 3.2: code for rehash()

Step 2 The next step transforms the hash into something that can index into the hash table. To avoid expensive modulo instructions, the number of buckets (i.e. size) of VectorWise's hash tables is a power of 2. This way the $numberofbuckets - 1$ can be used to create a bitmask. This is then logically ANDed over the hash which in turn produces a value that can offset into the hash table.

Figure 3.6 illustrates the process. In the reference query the number of buckets in the hash table is 8,388,608. When one is subtracted from this number, the mask is `0x7FFFFF`. Listing 3.3 shows the code of the `bitand()` function.

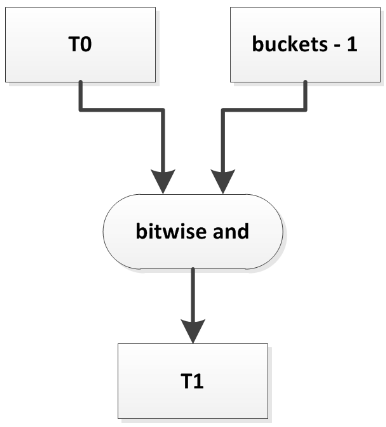


Figure 3.6: hash probing: step two

```
void bitand(int N, int* res, int* col1, int val)
{
  for (i=0; i<N; i++)
  {
    res[i] = col1[i] & val;
  }
}
```

Listing 3.3: code for bitand()

Step 3 The prior step generated an index or offset to the hash table. This step retrieves the value contained at that location and also determines if there is a candidate match or not.

```
int lookup_initial(int N, int* hits, int* hashes,
                        int* buckets, int* rhs_indices)
{
  k = 0;
  for (i=0; i<N; i++)
  {
    int hit = buckets[ hashes[i]];
    rhs_indices[i] = hit;
    if (hit)
    {
      hits[k++] = i;
    }
  }
  return k;
}
```

Listing 3.4: code for lookup_initial()

Figure 3.7 illustrates the process; the corresponding function is `lookup_initial()` contained

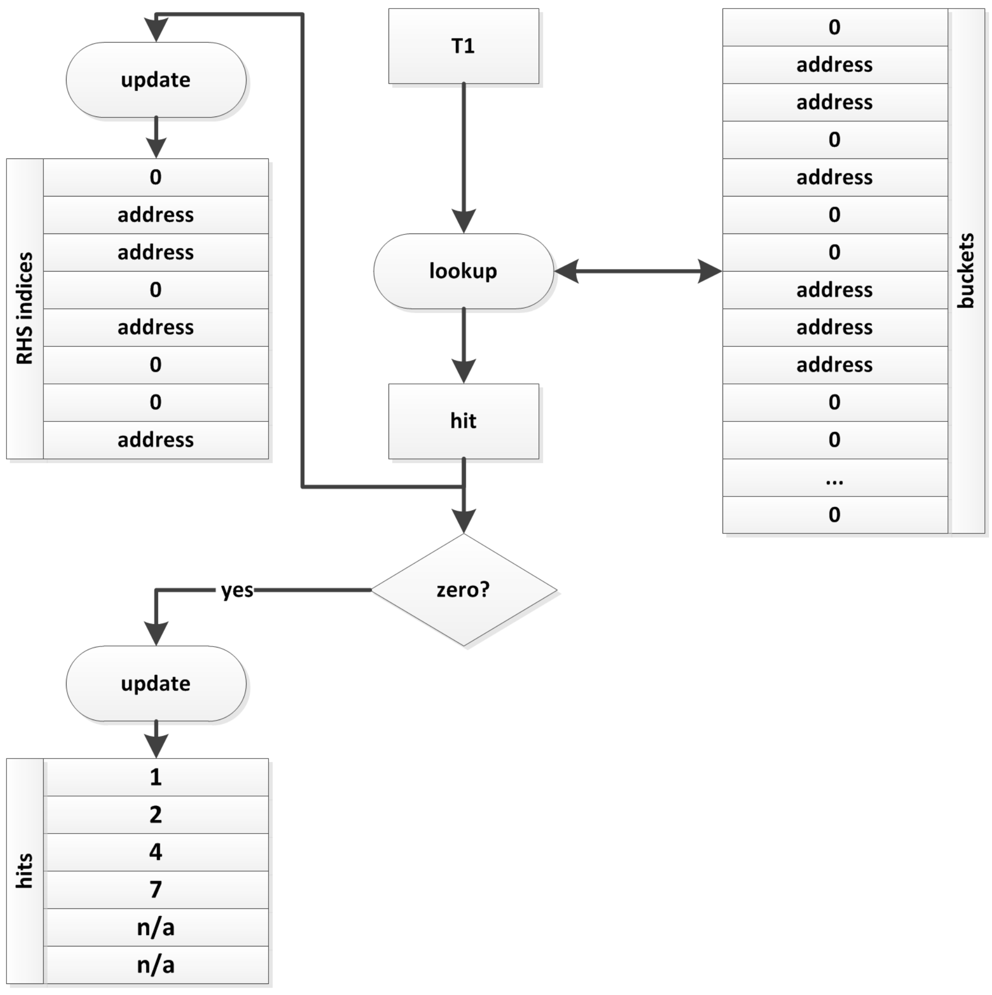


Figure 3.7: hash probing: step three

in listing 3.4. First, `T1` (the previously generated hash) is used to offset into the hash table and retrieve its value. The hash table itself is simply an array named `buckets`; in the example query this is 32 MB. This value is immediately stored in the array named `rhs_indices`. The value is then checked for zero, and if it is non-zero, another array named `hits` is updated. The value that is entered into `hits` is an offset into the array `rhs_indices`. Therefore, all non-zero values can be quickly retrieved using this indirection vector. Zero is the default value for each entry in the hash table. If a bucket contains a zero it implies that nothing from the right-hand side table hashed to that location, ergo it is guaranteed that there will be no match.

Step 4 Step 3 located a set of offsets into the RHS table. This step now verifies if those offsets produce a real match or if they are consequences of hash collisions to the same bucket (false positives). Figure 3.8 illustrates the process. Note that the RHS table is named `temporary table`; this is due to the fact that in the example query, the RHS is materialised from other operators and is not present in the database schema. Similar to hash() and rehash(), there is check() and recheck() shown in listings 3.5 and 3.6 respectively. Again, this is due to the example query using two keys to join the tables together.

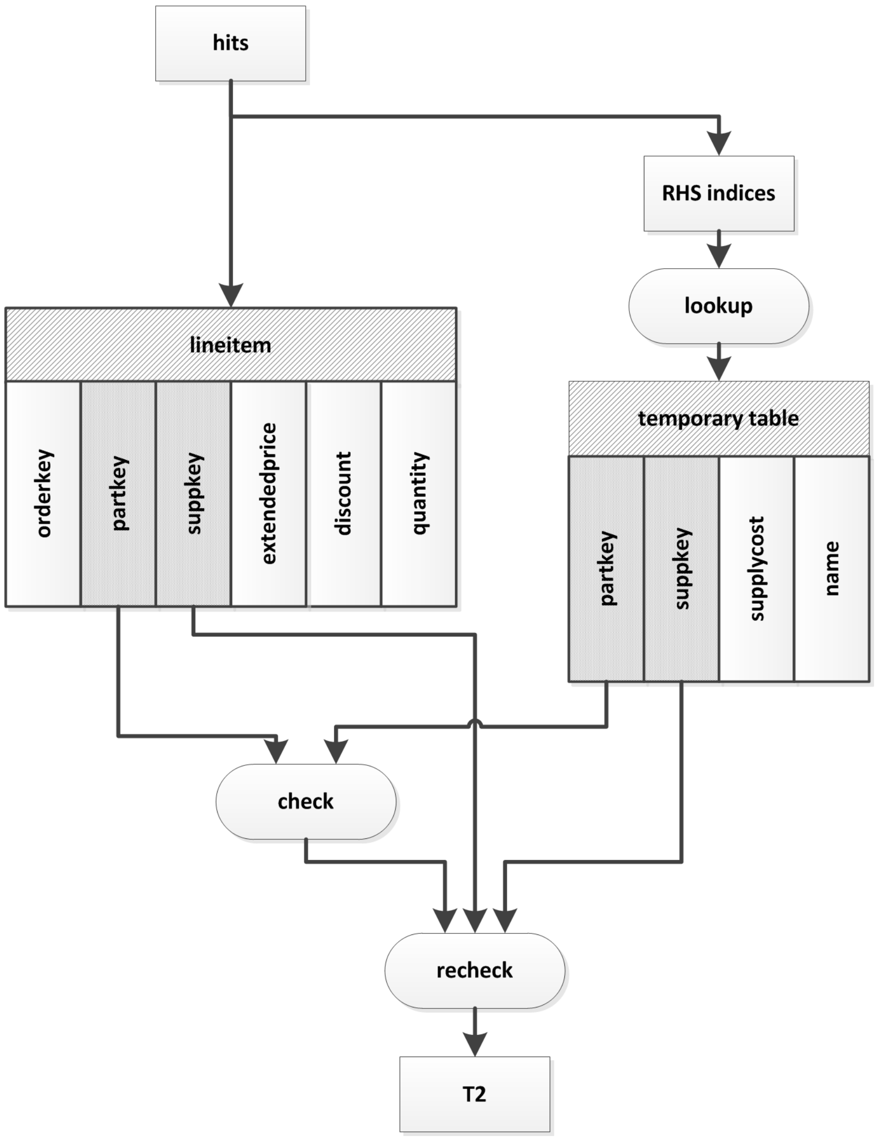


Figure 3.8: hash probing: step four

In each function the hits array is used to avoid trying to match rows that are guaranteed not to match. Its values offset into the LHS table and rhs_indices. The values in rhs_indices are then used to offset into the RHS table (temporary table). The values from the LHS and RHS are then compared against each other and if they are not equal, the result is set to true. This may seem counter-intuitive but its rationale is explained in the next step. recheck() takes the Boolean output of check() and logically ORs its own result onto it. This implies that both keys must match in order for the join to succeed.

```
check(int N, bool* res, int* lhs_key, int* rhs_indices,
                        int* rhs_key, int* hits)
{
  for(i=0; i<N; i++)
  {
    j = hits[i];
    if ( lhs_key[j] != rhs_key[ rhs_indices[j] )
      { res[j] = true; }
    else
      { res[j] = false; }
  }
}
```

Listing 3.5: code for check()

```
recheck(int N, bool* res, bool* col1, int* lhs_key, int* rhs_indices,
           int* rhs_keys, int* hits)
{
  for (i=0; i<N; i++)
  {
    j = hits[i];
    if ( lhs_key[j] != rhs_key[ rhs_indices[j] )
      { temp = true; }
    else
      { temp = false; }
    res[j] = col1[j] | temp;
  }
}
```

Listing 3.6: code for recheck()

Step 5 Step 4 compared keys from both tables and created a list of Booleans specifying whether the match succeeded or not. The reason behind this is because multiple rows on the right hand side may have gone to the same bucket in the hash table causing a conflict. To resolve this, a technique called bucket chaining is used.

VectorWise uses a complementary array called `next` that is used to contain bucket collisions on the main hash table and also on itself. `next` contains as many buckets as there are rows in the RHS table.

Figure 3.9 illustrates the process. Notice it is very similar to the process in step 3. Essentially it is doing the same thing except that `buckets` is now replaced with `next` and here the function only works on a subset of the input. The output of step 4 is fed into the function `select_nonzero()`, shown in listing 3.7, which generates an array of offsets into `rhs_indices`.

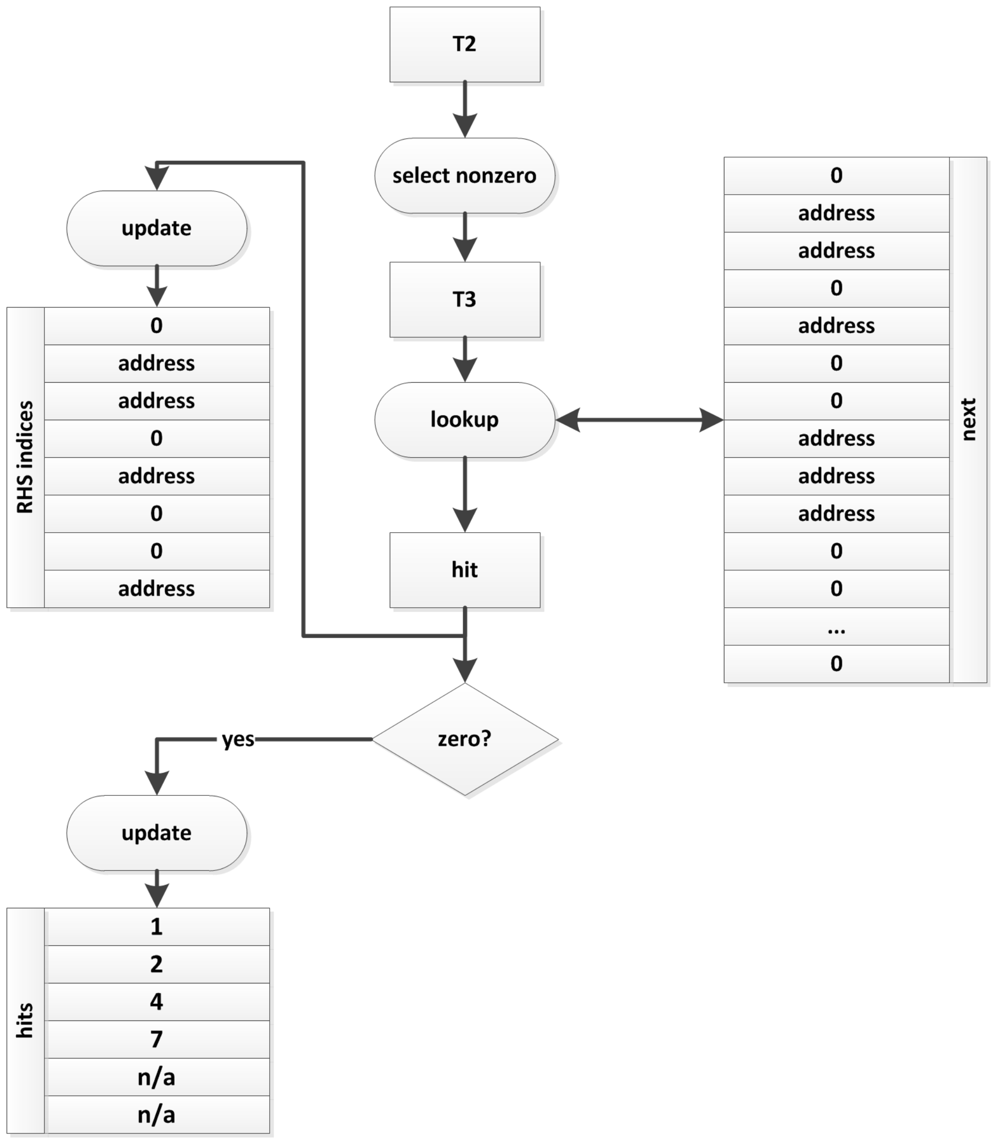


Figure 3.9: hash probing: step five

The values contained at these offsets are the addresses into the RHS table where the check failed. This implies there is still a possibility for a match but the correct row is in another location. These addresses are then used in `lookup_next()`, shown in listing 3.8, which indexes into the `next` array. Here an address is returned which is a new candidate to join the tables.

What happens next is an iterative process. The results of `lookup_next()` go into `check()` and `recheck()` again. After this, `select_nonzero()` and `lookup_next()` are called again. This happens until `lookup_next()` returns all zeros i.e. there are no more hits in the `next` array. Theoretically this looped logic could have many iterations but in practise one to three are typical.

```
select_nonzero(int N, int* res, bool* col1, int* sel)
{
  k=0;
  for (i=0; i<N; i++)
  {
    j = sel[i];
    if (col1[i])
    {
      res[k++] = j;
    }
  }
}
```

Listing 3.7: code for select_nonzero()

```
lookup_next(int N, int* hits, int* next, int* rhs_indices)
{
  k=0;
  for (i=0; i<N; i++)
  {
    int hit = next[ rhs_indices [ hits[i] ] ];
    rhs_indices[ hits[i] ] = hit;
    if (hit)
    {
      hits[k++] = hits[i];
    }
  }
  return k;
}
```

Listing 3.8: code for lookup_next()

Step 6 Step 5 iterates for a number of cycles until all possible matches are found. Step 6 completes the join by retrieving the appropriate input columns from both tables and forming a new table which will be the hash join's output.

```
fetch(int N, int* res, int* col1, int* col2)
{
  for (i=0; i<N; i++)
  {
    res[i] = col2[ col1[i] ];
  }
}
```

Listing 3.9: code for fetch()

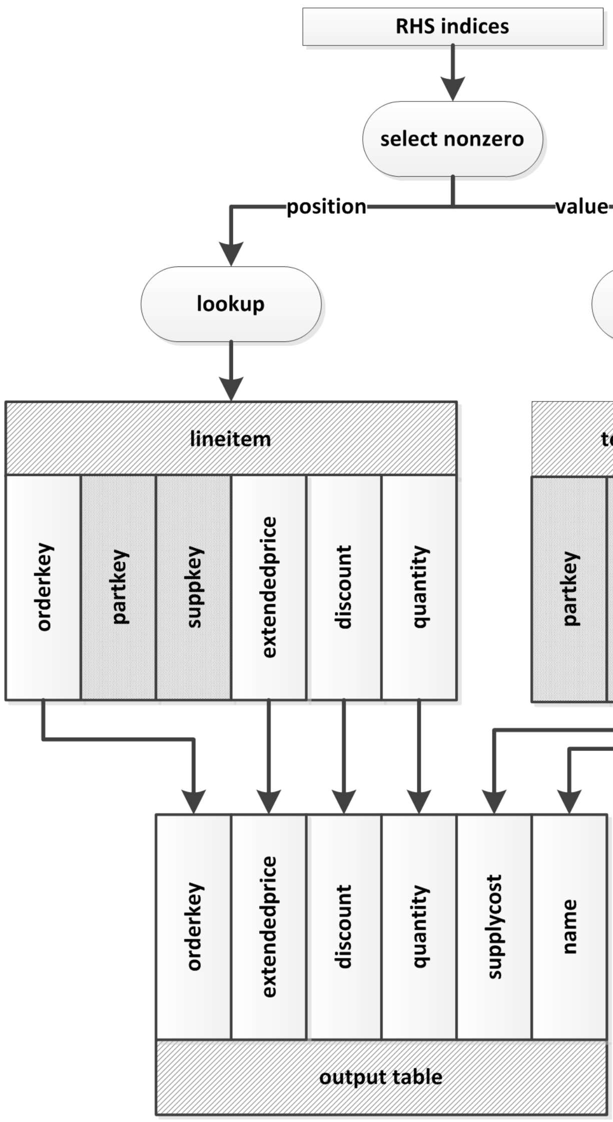


Figure 3.10: hash probing: step six

Figure 3.10 illustrates the process. The `rhs_indices` array can contain an address or a zero. If there is an address, it means the value should be retrieved from that location from the right hand side table. If there is a zero, there was no match and nothing should be retrieved. The array also encodes which rows to take from the left-hand side; the position of a non-zero entry in `rhs_indices` corresponds to the matching position in the left-hand side table. To implement this, the function `fetch()`, shown in listing 3.9, is used and is called once for every column to be retrieved.

Function Analysis

This section looks at the performance of each of the functions that comprise the hash probe phase.

TPC-H query nine is run again, this time collecting various hardware counters using VTune. This can help identify bottlenecks caused by the microarchitecture's structures. Note that in the results there are variations of `fetch()`; this is due to VectorWise's code generation based on templates. `fetch_x()` implies that it is retrieving values from a column of width x.

Function	Reorder Buffer	Reservation Station	Load Queue	Store Queue
bitand	5.56%	20.32%	0.00%	2.41%
check	28.68%	55.19%	0.00%	0.00%
fetch_32	77.26%	0.66%	0.00%	0.00%
fetch_64	81.68%	0.00%	0.00%	0.35%
fetch_8	54.33%	1.96%	0.00%	0.08%
hash	0.16%	63.24%	0.00%	0.00%
lookup_initial	9.89%	30.51%	0.00%	0.03%
lookup_next	27.72%	36.56%	0.00%	0.00%
recheck	7.82%	74.56%	0.00%	0.00%
rehash	0.11%	51.84%	0.00%	0.00%
select_nonzero	0.11%	0.44%	0.00%	0.00%

Table 3.1: TPC-H query nine: percentage of time structure is full

Table 3.1 shows the results of measuring query nine for reorder buffer (ROB) stalls, reservation stations stalls, load queue stalls and store queue stalls. The values shown are percentages of time that the CPU spends in that function where that particular resource is full.

A full ROB and full reservation station is indicative that instructions can't complete as fast as they are arriving into the pipeline. There are many different reasons to explain these resources being full. Since many of the functions address large data structures that don't fit into the cache, it is likely that cache misses is the culprit. Cache misses cause chains of instructions dependent on memory operations to stall and thus filling up the structures quickly.

In any cases where the ROB is full, the oldest instruction in the pipeline is probably a cache miss. Since it can't commit and retire, the reorder buffer becomes full and new instructions can't enter the pipeline until the problematic memory instruction completes. Instructions already in the ROB are free to execute if they are not dependent on the stalling memory operation, but the number of instructions available is limited by the size of the ROB itself. This is a typical structural bottleneck that can hurt scalar code performance.

In cases where the reservation station is full, it means that there is a long chain, or more likely, several short chains, of dependencies that are stalled waiting for some memory operation. This is even worse for performance because there may be instructions in the ROB but not in the reservation station that could complete. If these were to be executed during a cache miss, the penalty could be partially hidden. Unfortunately reservation stations are small and often hurt scalar performance for this reason.

Cache hit/miss ratio has been omitted intentionally. A poor hit/miss ratio implies that execution performance will be bad, but a good hit/miss ratio does not necessarily imply that performance will be good (nor that cache misses are not a problem). It is very difficult to measure the impact of a cache miss as sometimes it can be hidden and sometimes not. The resource stalls in table 3.1 give more of an indication of cache miss penalties than the cache hit/miss rate can.

Discussion

VectorWise is a very good query engine in that it makes a compromise between function call overhead and data/code locality. It takes a block of input values at a time and passes them from the bottom of the query tree all the way to the top. From this there is an advantage of running the query operators over multiple values to amortise the cost and, if the block size is chosen well, there is data locality between the output of one operator and the input to next. This new model works better on modern microprocessors than the tuple-at-a-time model and the column-wise processing model.

The problem is that this philosophy is more difficult to apply to hash joins. Unfortunately it is necessary that the hash table is fully built before it can be probed. This means that for any hash join operator, 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 will become large and, in many cases, not cache resident thus reducing the benefit of data locality.

This is further exacerbated by the fact that if short blocks (e.g. 1024 elements) are used, data locality is sacrificed more. Since the lookup to the hash table is at pseudorandom locations, a lot of effort is spent bringing in various sections of the hash table into the cache. However, when the block moves to the next operator, the whole tree must be traversed again before the hash join operator and its tables are used again. As the query becomes larger and more complex, there is more chance of the hash table being evicted from the cache before it is needed again.

3.2.4 Data-Level Parallelism

For each of the functions in the hash probe operation, all of them are loops with independent operations, thus allowing for data-level parallelism. Even though the iterations are independent, many of the loops contain logic that makes it difficult to parallelise using the current generation of SSE instructions.

Of all the functions that comprise hash probing, only `hash()`, `rehash()` and `bitand()` could be autovectorised with SSE instructions by the GNU and Intel compilers.

`lookup_initial()`and `lookup_next()` share similar characteristics to each other. The main loop is data parallel however it is not suitable to be vectorised with SSE. There is a level of indirection that requires an indexed load instruction and SSE requires that all data is stored contiguously. `check()`, `recheck()` and `fetch()` have the problem of indirection as well. There needs to be an indexed load instruction in order to capture the data-level parallelism.

The conditional part of the loops, which also resembles `select_nonzero()`, cannot be easily vectorised using SSE. It is possible to store the variable `i` in an SSE register by adding the scalar variable `i` to an SSE register containing the constants `[0, 1, 2, 3]` but it is also necessary to reorder the positions of this register dynamically based on a mask. If the register could be rearranged and stored consecutively, it may be possible.

SSE3 introduced the packed shuffle bytes instruction that can permute a source SSE register based on a set of offsets contained in another SSE register. The difficulty lies in generating the offset vector dynamically using only SSE instructions. An alternative approach would be to generate a mask and use it to offset into a lookup table in order to retrieve the permutation input; the consequence of this is having an extra load and more a complex loop body.

SSE is not expressible enough to capture the data-level parallelism in these functions. It should be stated, however, that even if SSE were to introduce instructions like gather/scatter it would not necessarily make them suitable for these kernels. Intel's LRBni [1] instruction set includes gather/scatter instructions however their implementation in the Larrabee microarchitecture [49] is built assuming all values will hit the same L1 cache line. Clearly that isn't the case in hash probe kernels. This assumption also limits the scalability of the architecture.

In contrast, vector architectures have traditionally been built to tolerate the penalty of long latency operations and, depending on the ISA, contain instructions more suitable to capture non-trivial data-level parallelism. What has been seen in the hash probe functions is directly applicable to a vector ISA.

Chapter 4

Design Decisions

This chapter discusses many of the design choices made in order create a vector microprocessor. Design decisions have been made based on what has found to be required in chapter 3. Many ideas are inspired by prior research in vector microprocessors discussed in chapter 7.

4.1 Instruction Set Architecture

This section examines the design choices in the user-visible portion of the architecture.

4.1.1 Base ISA

x86-64 was chosen as the base ISA to extend for several reasons. It is the leading ISA in the server market [13] having roughly 60% of the market share. It is a very universal ISA with mature optimising compilers and toolchains. VectorWise, although not exclusively written for x86-64, has several opimisations made for x86-64 and the Intel Xeon 5500 series [27]. x86-64 is also a large improvement over the archaic 686 ISA and many impovements have been made e.g. the number of general purpose registers is doubled, and some legacy features have been removed e.g. memory segmenting.

Additionally, using x86-64 also allows for any simulated experiments to be compared against modern commodity hardware. Chapter 3 explored VectorWise using an x86-64 microprocessor; using x86-64 and modelling the same microarchitecture would allow a baseline scalar simulation to be verified against real hardware. Furthermore, any improvements made to the architecture would be more transparent when comparing the results of scalar and vector experiments.

Finally, because x86-64 already has SIMD extensions i.e. SSE instructions, it can be interesting to see where these are insufficient to capture the data-level parallelism in the profiled functions of chapter 3. Although different x86-64 incarnations contain varying support of different SSE version, x86-64 has SSE and SSE2 integrated into its core ISA.

4.1.2 Memory or Registers

A major design decision was whether to have a memory-memory based ISA like the CDC-STAR 100 [25] or a vector-based ISA like the Cray 1 [48].

x86-64 has very strict rules regarding its instructions. Instructions must change the state (main memory, architectural registers) in linear order. Additionally, instructions must commit atomically i.e. either the instruction does everything together or it does nothing at all. With these rules in mind, it could be very difficult to implement a memory-memory ISA that also uses many of the optimisations found in current-generation microarchitectures. For example, to allow instructions to issue speculatively is a very useful optimisation; allowing instructions that change the state of the main memory to issue speculatively could be dangerous and very difficult to reason about correctness. Servicing exceptions and interrupts becomes more difficult for the same reasons.

Using vector registers is in-theme with the existing x86-64 ISA. Architectural registers can be part of the user-visible state and rules of atomicity and order can be more easily applied. Aside from the complications of the x86-64 semantics, there are many optimisations that can be performed when vector registers are used, such as speculation, register renaming and out of order execution [15]. For all these reasons, a register-based ISA was chosen for this design.

4.1.3 Registers

The registers used for the ISA have been chosen based on the requirements of functions explored in chapter 3.

Sixteen general purpose vector registers were added. Each vector register is further partitioned into a discrete number of elements; this number is known as the maximum vector length (MVL). The MVL has not been hardcoded into the ISA, instead instructions are introduced to allow the programmer retrieve its value at runtime. This way loops can be vectorised generically using stripmining [3] and the number of iterations needed can be decided dynamically.

To make the hardware simpler there is no subword parallelism i.e. the MVL doesn't change depending on the datatype used. For example, if the MVL is 32 this implies an instruction can operate on 32 qwords, 32 dwords or 32 words. This is in contrast to the SSE instructions that can operate on four qwords, eight dwords or sixteen words. This design decision can be wasteful or not depending on the datatypes being used.

The motivation behind this choice came from the necessity of indexed memory operations. Gather and scatter take a base address plus a vector of offsets. The datatype of the instruction should refer to element being pushed to or pulled from memory. If subword parallelism is allowed, it means that an indexed memory instruction could operate on x qword elements, $2x$ word elements, $4x$ word elements or $8x$ byte elements and thus requires an equivalent number of offsets too. The consequence of this is that the maximum offset decreases by a factor of 2 each time the datatype is shrunk. From the programmer's point of view this can be confusing and if the range of the offsets is not known before runtime, it can lead to bugs in the code quickly. For this reason subword parallelism has been omitted in favour of a more simple model.

In addition to the general purpose vector registers, there are also eight vector mask registers. Each register contains a number of bits that correspond to exactly one element in a general purpose vector register. Some vector architectures, e.g. Tarantula [14], contain a single vector mask register. The reason for introducing multiple registers is to allow the programmer to perform bitwise manipulation instructions on the masks. Some of the kernels found in chapter 4 can benefit from instructions like logical ORing two masks together. The consequence of this is that every maskable instruction in the ISA must encode the mask register to be used. Finally, a vector length register was added, however this can be seen as an additional scalar register to x86-64.

Although the ISA brings sixteen vector registers and eight mask registers, these numbers are a little bit arbitrary. Sixteen was chosen for two reasons. First, x86-64 contains sixteen general purpose registers and sixteen XMM (SSE) registers. Secondly, sixteen can be encoded into a single nyble which allows a much clearer presentation when writing/debugging the vectorised programs. Because there were no assemblers that could handle the new instructions, this had to be done at the binary level. In reality, sixteen and eight are very much in excess of what is required by the vectorised kernels. Six vector registers and three mask registers are all that is really needed, but for simulation purposes this has little bearing.

4.1.4 Instructions

The reader is referred to appendix A for a detailed listing of all the new instructions introduced. In general, the new ISA has a lot more flexibility than the SSE versions. Currently it only works with integer datatyps, as required by the profiled functions. It can easily be further extended to incorporate floating point datatypes. The new ISA includes the following features:

- unit-stride and indexed memory operations
- vector/mask register initialisers
- vectorised logical and arithmetic instructions
- comparison instructions that write to the mask registers
- more suitable position manipulation instructions
- mask-mask logical instructions
- vector register length manipulation instructions
- the majority of instructions can take an optional vector mask

4.2 Microarchitecture

This section examines the design choices of the microarchitecture that implements the vector ISA.

4.2.1 Out of Order Execution

One of the biggest design decisions made was to allow vector instructions to issue out of order. [15] showed that by using register renaming and out of order execution, additional performance can be gained. 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. The decision to allow issuing vector instructions out of order affects many of the subsequent design decisions.

There are also drawbacks to an out of order microarchitecture. The structures used to achieve out of order execution don't scale well and very power-hungry [42]. 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 scalar out of order microprocessors.

4.2.2 Memory

Cache Integration

The block at a time processing technique as used by VectorWise is very cache conscientious. Although as seen in chapter 3 large structures like the hash tables have trouble fitting in the cache hierarchy, many other structures can. Of particular importance are the blocks which flows through various data operators storing intermediate results in cache-resident arrays. For this reason it is highly desirable to take advantage of the cache hierarchy when possible.

[45] proposes a solution to integrating vector support into an existing superscalar processor. Part of the work is integrating the vector units with the cache hierarchy. Their novel solution involves bypassing the level 1 data cache and going directly to the level 2 cache. The main motivation behind this was that adding the logic necessary to support vector loads at the level 1 data cache could compromise its performance for the scalar units.

Because 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 of going to the level 2 cache instead. The level 1 cache is typically banked at the word level as scalar units will load and store single words at a time. In contrast, the level 2 cache is typically banked at the line level as transfers to and from the cache are always entire cache lines making it more ideal for vector access patterns.

The level 2 cache is always larger than the level 1 data cache so there is the additional benefit of the potential to have a larger working set. This design was later used in Tarantula [14] which had a 4 MB banked level 2 cache which the vector unit could take advantage of. For these reasons, this technique is used in our design.

Aliasing

One of the problems of allowing out of order memory operations is that loads and stores may have implicit dependencies, i.e. they operate on the same memory locations in a strict order. This is known as a memory aliasing problem and in modern out of order scalar microprocessors, there is typically an internal structure called the load/store queue that checks for and resolves these scenarios. The problem is that this structure must be fast and for that reason must also be small because it is associative. Checking vector memory operations on top of the scalar ones would be problematic as the vector memory operations address a larger range of memory, and in the case of indexed operations, can be in very random locations.

The profiled application exhibits complete data-level parallelism per function and it is already known there are no carried dependencies, implicit or explicit, within the main-body loops. In contrast, one function's output is very often another function's input and these values are generally held in array structures in memory. If `fun1()` writes its results to memory and then later `fun2()` accesses these these results, there is a dependency that must be honoured.

Adding logic to detect this infrequent cases would be overkill and would hurt performance. Instead, the ISA is appended with a set of vector memory fence instructions. They are categorised as follows:

- vector memory store → vector memory operation
- vector memory store → scalar memory operation
- scalar memory store → vector memory operation

Only the first fence is needed to correctly vectorise the profiled functions with correct memory interaction. The second and third are for scalar/vector interaction and are used primarily here for debugging purpose. It is possible that they could be more useful in other codes and future work. There is no scalar store → scalar memory operation fence as this already included in the x86-64 ISA.

Cache Level Coherency

One of the issues that crops up by accessing the L2 cache directly is keeping the data coherent with respect to the L1 data cache. The vectorised kernels have almost no scalar/vector interaction so there is never a case when a vector store will write to the cache immediately followed by a scalar load reading from the same location. It is still necessary to have some form of coherency as a safety net, so a simple technique is proposed.

The cache hierarchy should be inclusive. The L1 data cache is expected to use a write-through policy meaning that anything that is written to this cache will also be written to the L2 cache. When there is a vector store to the L2 cache and this line is also present in the L1 data cache, it is invalidated in the L1. If a vector store evicts a line from the L2, it must also be invalidated in the L1. This is not an efficient solution, but in practise storing a vector to a location already present in the L1 data cache seldom happens.

4.2.3 Chaining

In the functions listed in chapter 3, there are often chains of dependencies from the head to the tail of the loop bodies. For this reason it is desirable to introduce chaining. If the vector length in a chain of instructions is particularly long, chaining can help to minimise the penalty of each instruction's startup time. Chaining requires that the output of a vector instruction is generated at a regular and predictable rate. For the majority of instructions, this property holds however the exceptions are the memory instructions.

Vector loads pose the most problematic situations. Because of the use the memory hierarchy, it is uncertain if the data is cached or not until the load issues. Even trickier is when some of the data is cached and the rest is in main memory. It could be possible to implement chaining from loads under these circumstances, but the hardware implementation would be excessively complicated. For this reason loads are not allowed to chain.

Vector stores have a different behaviour. The input to a store instruction must already be in a vector register and thus can be fed a new element every cycle without stalling. Stores can therefore be chained in from other vector instructions.

The following is a list of chainable pairs. mask register $\rightarrow$ scalar register (e.g. mask population count) instructions are not chainable as the full contents of the register must be present before an evaluation can take place.

- vector register $\rightarrow$ vector register
- vector register $\rightarrow$ mask register
- mask register $\rightarrow$ vector register
- vector register $\rightarrow$ store

4.2.4 Vector Lanes

A fully pipelined vector instruction without lanes will take $startup + vlen$ cycles where $startup$ is the startup time i.e. the time it takes to produce the first element, and $vlen$ is the vector length of the instruction. Vector lanes allow a simple way to exploit the data-level parallelism in the vector ISA. If there are x lanes then an instruction's time can be reduced to $startup + vlen/x$ cycles.

This is useful for all the vector instructions, but it is the loads that will likely benefit the most. The loads cannot be chained and therefore there is a penalty where dependent instructions cannot issue until the load fully completes. This is exacerbated by the fact that the loads are often keystones of the loop body and must be executed before any other work can begin. Lanes can help reduce this penalty by generating the addresses of indexed loads in parallel. Unit-stride loads don't need this mechanism as they tend to access a small amount of contiguous cache lines.

Chapter 5

Implementation

This chapter takes the ideas from chapter 4 and implements them using a cycle-accurate hardware simulator.

5.1 Choice of Simulator

This section discusses the reasoning behind the choice of simulator to implement the design. It was paramount that the simulator be cycle-accurate and give sufficient statistics about its microarchitectural structures. It was also important that the simulator has some support for the x86-64 ISA. The choice was narrowed down to three simulators.

1. PTLsim
2. MARRSx86
3. M5

Each choice was evaluated considering its advantages and disadvantages. Here the principal pros and cons of each option are given.

5.1.1 PTLsim

PTLsim is a cycle-accurate x86-64 microprocessor simulator. There is an academic paper describing the overview of the project [56] and also a very detailed user manual [57] explaining its internals.

Pros

- Full x86-64 implementation
- SSE implementation up to SSE3
- Relatively fast (between 100,000 and 500,000 instructions retired per second on our test machine)
- Very well documented
- Codebase nicely structured and extensible
- Statistics for relevant microarchitectural structures present
- Good implementation of statistics collection and management

Cons

- Codebase no longer maintained
- Full-system simulation mode uses Xen and is very cumbersome
- TLB simulation is restricted to full-system simulation
- Memory model is highly simplified
- Caches all use a write-through policy
- All configurable parameters must be hard coded thus forcing a recompilation of the binary per simulation

5.1.2 MARSSx86

MARSSx86 is a full-system cycle-accurate accurate simulator based on MPTLsim [58], itself based on PTLsim. The principal differences between it and PTLsim are that it can model a variety of cache-coherency protocols and the full-system simulation mode is removed from Xen and placed into QEMU instead.

Pros

- Based on PTLsim which has many advantages independently
- Replaces Xen with QEMU. Full-system simulation easier to use
- Active albeit slow development with a small community
- Work being done on an Intel Atom core model which could be useful for simulating an in order core
- Some work done integrating DRAMSIM2 [47] which could improve the memory model considerably

Cons

- Its raison d'être is to simulate cache coherency protocols which is not in theme with this project
- All changes to the PTLsim side are undocumented and often difficult to infer a meaning
- Statistics are difficult to interpret as there are many redundant counters from PTLsim that conflict with the new ones
- Its stability is not as good as the original PTLsim
- The official contributions don't change the memory model outside of the caches

5.1.3 M5

M5 [6] is a multi-ISA cycle-accurate full-system simulator. It was originally designed to model networked systems based on the Alpha ISA but has since grown into a rich featured general purpose microarchitecture simulator.

Pros

- Actively developed with a large community of users
- Configurable parameters are input to the simulator at runtime
- Very detailed and intuitive statistics

Cons

- Slower simulation speed than PTLsim
- Documentation very outdated
- The Alpha model is the most stable
  - This architecture is no longer developed
  - Using it would introduce cross-compilation into the experiments
  - Using it makes it more difficult to compare against profiled results on the x86-64 microprocessor
  - There was already a proposal for vector extensions to the Alpha named Tarantula [14]. By reproposing vector extensions for this ISA, the novelty of the project may diminish.
- the x86 model is not nearly as mature as the Alpha model

5.1.4 Verdict

The three options for the simulator presented a difficult choice. Each had their share of advantages and disadvantages. It was decided that simulating an x86-64-compatible microprocessor is very important in order to simulate the same binaries that had been measured in chapter 3. M5's support for x86-64 was too lacking and so was removed from the equation.

The choice between PTLsim and MARSSx86 was also difficult. On one hand, MARSSx86 was actively maintained and solved many the problems involving full-system simulation. On the other hand, its codebase was undocumented and often difficult to follow; additionally, most of the benefits it offered were for modelling cache coherency protocols. The primary reason why full-system simulation is appealing is because the TLBs could be modelled, however it was demonstrated in chapter 3 that using huge pages [17] with VectorWise can reduce TLB misses and hence reduce the requirement of the TLB modelling. The choice was thus narrowed down to PTLsim.

5.2 PTLsim

This section provides an overview of PTLsim's modelling capabilities. For a full explanation of PTLsim's features, the reader is referred to [56] and [57].

PTLsim is intended to model a modern out of order x86-64 microarchitecture. It implements the full x86-64 ISA; fully supports MMX, SSE and SSE2; and contains partial support for some instructions of SSE3. Its modelled microarchitecture is not specific to any x86-64 implementation but instead derives its own based on major features from the then (circa 2007) current-generation x86-64 processors. It is highly configurable meaning that any current x86-64 implementation should be able to be modelled within an acceptable margin of error.

PTLsim is bundled with two models which, in their own terminology, are named cores.

1. *seq* - A functional core intended to gather simple statistics regarding the instruction stream and verify the correctness of a program's output
2. *OoO* - A timing core intended to accurately model the internals of an out of order microarchitecture

5.2.1 Out of Order Core Features

The *OoO* core is separated into an in order pipelined frontend; an out of order execution engine; and an in order retirement stage.

In order Frontend

The frontend models a variety of different features. It models a configurable instruction cache (icache) that fills a fetch buffer structure. The width of the icache to the fetch buffer is configurable. From the fetch buffer, the rest of the pipeline takes instructions as they are available and passes them down the pipeline that:

- decodes instructions into μops [44]
- renames the architectural registers to physical registers
- allocates key structures such as reorder buffer entries and load/store queue entries
- places the instruction in a ready-to-dispatch state

In reality PTLsim has an internal cache filled with basic blocks of pre-decoded instructions in order to speed up simulation. The frontend's pipeline width and length are both configurable. All key data structures accessed here are configurable in size e.g. the physical register file, reorder buffer and load/store queue.

Out of Order Engine

The out of order engine is implemented using using a clustering mechanism [42]. Instead of a unified reservation station, there are several smaller ones named issue queues that can access a subset of the total available functional units. An issue queue coupled with its functional units is called a cluster. Issue queues and functional units are placed into exactly one cluster.

Every cycle, the oldest μop in the ready-to-dispatch state attempts to find a suitable cluster that it can execute on. If one is found, the instruction is copied from the reorder buffer into the issue queue (there is no separate dispatch queue). The μop remains in the issue queue until all its operands are ready and then proceeds to execute on one of the available functional units. The functional units are fully pipelined meaning that a new instruction can issue each cycle irrespective of whether an older instruction is already occupying the unit. In the beginning this was seen as a problem, but when looking through the Intel's documentation [28] it was seen that Westwere can issue a new instruction per cluster per cycle too.

After execution, there is a forwarding stage that feeds the completed instruction back into the clusters before its result is written to a register. Here a bypass network is modelled with configurable parameters for inter-cluster latencies and bandwidth. After, the result moves to a writeback stage and can write to its physical register.

Memory instructions are allowed to execute out of order as well. Every memory request requires an entry in the load/store queue which detects memory aliasing conflicts. In the event of a memory instruction issuing earlier than it should, the load/store queue will detect this and redispatch the problematic instruction also adding the conflicting memory instruction as an extra dependency. There is also a configurable memory alias predictor that can add these extra dependencies in the initial dispatch stage.

Instructions are allowed to execute speculatively using a branch predictor and a speculative register rename table. One downside is that there is no simulation of hardware prefetching. There are several recovery mechanisms in the event of misspeculation: replaying the instruction, redispatching the instruction and annuling the instruction.

In order Retirement

Executed μops remain in the reorder buffer until they can be committed i.e. change the architectural state. μops must commit in order and atomically i.e. if an x86-64 instruction is broken into multiple μops, then all must commit together or nothing can be committed at all. Each cycle the commit stage looks at the head of the reorder buffer to see if the oldest instruction is ready. If this is the case, the commit will retire the instruction and reflect any changes of the architectural register file or main memory to the user-visible state.

Memory

PTLsim models a cache hierarchy of up to three levels. There are separate level 1 (L1) instruction and data caches, a unified level 2 (L2) cache and an optional level 3 (L3) cache. The caches are inclusive meaning that: L1 $\subset$ L2 $\subset$ L3 $\subset$ main memory. The caches use a write-through policy meaning that stores update all levels of the hierarchy and main memory in the same moment. This is modelled with zero cost meaning that stores can write through all the way to main memory with no penalty.

Cache misses are implemented using two structures: the Load Store Request Queue (LSRQ) and the Miss Buffer (MB); these roughly compare with the Miss Handling Status Registers (MSHR) [32] used to implement non-blocking caches. The LSRQ is responsible for individual loads that miss the L1 cache whereas the MB tracks outstanding cache lines at any given level. Multiple missed loads to the same line will take multiple entries in the LSRQ but only a single entry in the MB.

Loads that miss the L1 cache do not remain in the issue queue. Instead, they are considered as complete and the reorder buffer entry is put into a waiting-for-cache-miss state. It is not allowed to commit and any dependent instructions are stalled until the cache miss is serviced. PTLsim uses an asynchronous wakeup mechanism. The caches work autonomously and brings the requested data from the closest cache or memory main memory. When it reaches the L1 data cache, the LSRQ is checked and its corresponding ROB is woken up. At this point the data can be broadcast to any dependent instructions and execution resumes normally.

Two absent features of PTLsim are a) the lack of bus modelling for memory operations; and b) its Miss Buffer cannot be configured/restricted per cache level. If not modelled, both of these things would give vectors an unfair advantage. They have been addressed in the vector implementation but left as is in the scalar implementation, essentially cheating against our favour. Fortunately their impact on the scalar performance is much less profound than what would be observed with unrestricted vector performance.

5.3 Vector Extensions

This section discusses the additions that have been made to PTLsim in order to accommodate the new vector instructions.

It was desirable to reuse as much as possible from the original PTLsim architecture so the additions necessary to implement the vector ISA would be minimal. One of the key design decisions made was integrating the vector units into the core itself, this is in contrast to a design like Tarantula [14] which is more like a co-processor for the Alpha. Almost all the structures of the original microarchitecture were reused with minimal modification. The major changes came from implementing vector memory operations that required two new structures: the Vector Memory Request File (VMRF) and the Vector Memory Fence Queue (VMFQ). First the simple changes are listed and then the new vector memory request structures are explained separately.

5.3.1 Minor Changes

The decode units had to be modified to incorporate the new vector ISA. The changes were minimal because the new instructions all have a fixed length and start with the same prefix.

Each x86-64 instruction is decoded into a single μop. The μop structure itself had to change in order to accommodate the new architectural registers and be expanded to hold more than four operands. The list of all the new instructions can be found in appendix A. Of interest is the indexed memory operations which require five operands in addition to the vector length register (an implicit source). This variation of the μop structure is named a *vop* for vector μop.

The register rename tables had to be changed to rename the new vector/mask/vector length architectural registers. Two new physical register files were added in order to support vectors and masks. Although not modelled in the simulator, it is assumed that these register files will be partitioned depending on the number of lanes used.

New functional units and issue queues were added. The original issue queues can handle up to four operands which is sufficient for the existing x86-64 instructions; in fact, only memory operations utilise the fourth operand. The new vector instructions needed two extra operands on top of the existing four. This is due to three 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 in case masking or a shorter vector length overwrites a part, but not all, of the register. 3) The implementation of vector memory fences (discussed later). The scalar issue queues using four operands can co-exist with the vector issue queues using six operands.

The issuing mechanism must change too. In the original PTLsim model, the functional units are fully pipelined i.e. they can handle one new instruction every cycle regardless of their latency. The vector functional units had to simulate pipelining a single vop at a time. This involved constraining the functional units so that only a single vop could be present in a given cycle. In order to implement chaining, chainable vector instructions dependent on other chainable vector instructions must be allowed to begin issuing before the dependent instruction completes. To accomplish this, another state is added called *first result* which is different from *issuing* or *finished issuing.* The *first result* state allows dependent vops to begin issuing but it does not allow the instruction itself to commit from the reorder buffer.

Finally, the Miss Buffer was modified to restrict the number of outstanding L2 cache misses permitted. In the scalar this is not such a problem. The Load Fill Request Queue limits the number of outstanding L1 data cache misses specifically and the Miss Buffer handles cache misses on all levels. When these structures are properly sized, it can estimate the number of permitted misses at the L1/L2 within a reasonable margin of error. The vector memory instructions bypass the L1 cache so it is not restricted by the LFRQ. The Miss Buffer was modified to have a finite number of outstanding permitted L2 cache misses.

5.3.2 Vector Memory Request File

Vector memory requests had to be handled differently from all other operations. PTLsim can issue memory requests out of order but to achieve this, a complex associative hardware structure called the Load/Store Queue (LSQ) must be used. The LSQ searches for memory aliases i.e. loads and stores that go to the same address, that may cause problems when issued out of order. When an alias is found, a recovery mechanism must be used so the memory operations appear to be ultimately executed in order.

The vector memory operations are known to be data independent at the element level and in most cases also data independent with respect to one another. There is no need to check for memory aliases in these operations and using the LSQ, a small associative structure, would limit the number of in-flight vector memory operations in the core.

It is also advantageous to take advantage of the regular patterns found in vector memory operations. Unit-stride loads/stores use consecutive elements that have a lot of spatial locality. It is therefore preferable to work with whole cache lines when possible. For indexed memory operations with less spatial locality, it is important to reduce the penalty of discarding irrelevant parts of the cache line.

Finally it is also preferable to reuse the existing asynchronous register wakeup mechanism in the PTLsim microarchitecture. This will allow the issue queues to be reused with little modification. It will also allow multiple outstanding vector memory requests and the ability to issue them out of order.

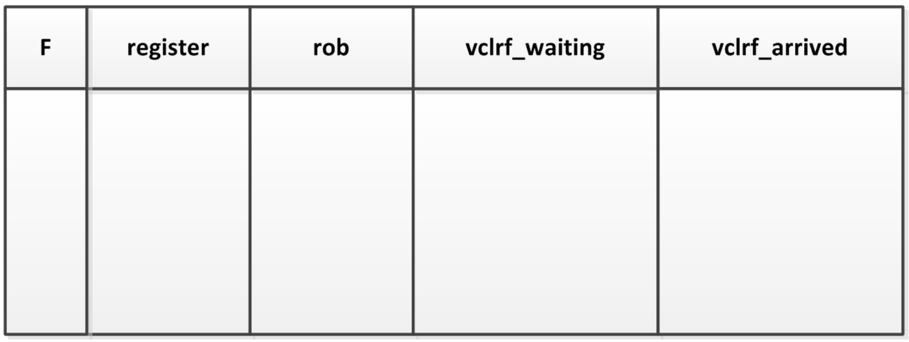
To achieve all this, a structure called the Vector Memory Request File (VMRF) was implemented. This structure itself contains three substructures:

- Vectore Load Request File (VLRF)
- Vector Store Request File (VSRF)
- Vector Cache Line Request File (VCLRF)

The VLRF manages all vector load requests; the VSRF manages all vector store requests; and the VCLRF manages cache line to physical register transfers and is utilised by both the VLRF and VSRF.

Vector Load Request File

Because one load may generate multiple cache line requests, it is desirable to separate the data structures to reduce redundant metadata. All vector loads, no matter how many cache lines they request, will take exactly one entry in the VLRF. The structure is simple and is considerably smaller than the VCLRF.



F	register	rob	vclrf_waiting	vclrf_arrived

Figure 5.1: Vector Load Request File

Figure 5.1 displays the structure of the VLRF. There is one bit **F** that indicates if the line is free or not. **register** is the index of the physical register to write to. **rob** is the index of the reorder buffer entry. **vclrf_waiting** is a bitmap of any entries in the VCLRF used by this load. **vclrf_arrived** is a bitmap of any VCLRF entries the load is using that have already returned and placed the data into the physical register. Not shown is an implicit index of the structure itself, 0 to $size - 1$.

By using this schema, every cycle all the non-free entries in the VLRF can do a bitwise comparison with the two bitmasks. If there is a match, the reorder buffer entry can be woken up and the load can wakeup its dependents and complete.

Vector Store Request File

The VSRF is very similar to the VLRF. Figure 5.2 shows the structure. The biggest difference is that the ROB entry is omitted; this is due to stores not needing a wakeup mechanism like the loads. The bitmaps are still there but renamed to **vclrf_tosend** and **vclrf_sent**. The former records which of the VCLRF are associated with the store and the latter tracks which lines have been successfully sent to the cache. Once again a bitwise comparison is used on the two bitmasks to detect completion.

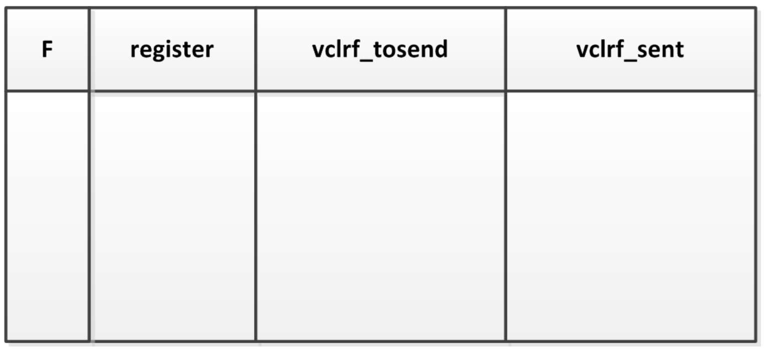


Figure 5.2: Vector Store Request File

Vector Cache Line Request File

The VCLRF is a structure that manages all cache line requests, both loads and stores. Each entry contains all the necessary information to make a request to the cache.

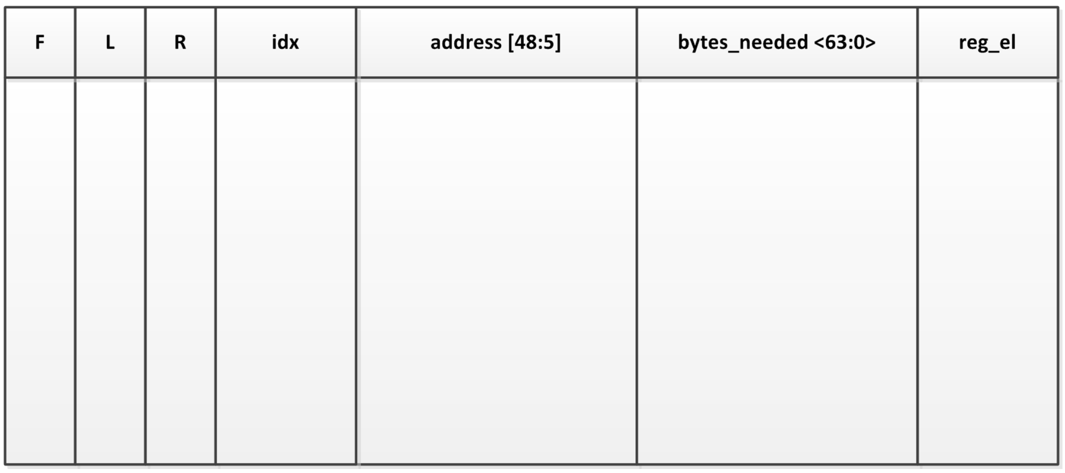


Figure 5.3: Vector Cache Line Request File

Figure 5.3 illustrates the VCLRF structure. There is a 1-bit field **F** to indicate whether the entry is free or not. A 1-bit field **L** to specify if the entry is a load or a store. A 1-bit field **R** that has a different meaning depending on whether it's load or store. For a load this indicates that the memory request has been sent to the cache. For a store, this isn't necessary because the entry can be recycled as soon the line is sent to the cache. Instead entries for stores use this to specify that the instruction has reached the head of the reorder buffer and is safe to commit. **idx** contains the implicit entry number of either the VLRF or VSRF, depending on what the line is used for. This is necessary to update the corresponding entry's **vclrf_arrived** or **vclrf_sent bitmaps**.

address contains the physical address of the cache line, for x86-64 - 42 bits are required.

bytes_needed is a bitmap that has a length the as the number of bytes found in an L2 cache line. **reg_el** is the first element of the physical register where the data should be read/written to. If a load/store single request has a complex memory pattern, multiple entries in the VCLRF must be used. This was necessary in order to avoid complex logic for shifting/masking cache lines to/from the physical register file.

Reorder Buffer Modification

Reorder buffer entries must be modified to contain a Vector Memory Request File identifier: **VMRF_IDX**. This will refer to either the VLRF or VSRF depending on the context. For vector stores, it is necessary to be able to read its entry in the VSRF at commit to ensure all of its data has been successfully written to the cache. Due to i asynchronous wakeup mechanism, loads don't actually need this entry for normal execution however both loads and stores need to have this information when misspeculation recovery occurs. In these cases, the entry in the VLRF or VSRF and all its corresponding VCLRF entries must be annulled and recycled.

Clocking the VCLRF

Once a load or a store has taken entries in the VCLRF, the structure is autonomous and can make forward progress by itself. Every cycle, the VCLRF will check for entries that have the **F** bit unset, the **L** bit set and the **R** bit unset. Of all the candidates, it will choose an entry based on a round-robin scheduling scheme hence making the structure behave like a list. If an entry is found, the Miss Buffer (outstanding cache misses) must be checked for a free entry. This may or may not be used, but the logic is much simpler if the check is made before any request is sent to the cache. Assuming an entry is available, the request to the L2 cache can be sent and the **R** bit is set. From here the cache manages the request and the entry is idle until the cache's data is written to the physical register. At this point the bitmap in the VLRF is updated to indicate that this cache line request has completed and the entry is recycled by setting the **F** bit.

On the same cycle, the VCLRF will also check for entries that have the **F** bit unset, the **L** bit unset and the **R** bit set. These entries belong to stores that are known to be safe to commit. The entry chosen here is less important because they should also belong to the same store instruction i.e the oldest store in the pipeline. The VCLRF will initiate the transfer of data to the cache and on completion update the relevant bitmap in the VSRF. Because of the x86-64 atomic commit property, it assumed that when a vector store is the oldest instruction in the pipeline and without exceptions, the stores from the VCLRF to the cache won't be interrupted.

To model all this, two configurable busses were added that connect the physical register file to the L2 cache. One for load requests, another for store requests and both are managed by the VCLRF. As an optimisation, the structure can handle partial cache line transfers. The **bytes_needed** structure in each entry indicates which bytes within the cache line will be needed. This can be 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. Indexed operations benefit from this especially as their requested bytes from each cache line tends to be small.

When a load request is sent to the L2 cache, it is processed for x cycles where x is the L2 latency minus the time spent on the bus. After, the L2 checks if another request is currently occupying the bus. If not, the transfer can initiate immediately, otherwise the transfer must wait y cycles for the bus to become free. The number of cycles that a request will occupy the bus is determined by the number of sectors requested from the cache line.

5.3.3 Vector Memory Fences

In the design phase of this work, it was decided to allow vector instructions to issue out of order. Observing the profiled kernels in chapter 3, there were few constraints needed on the ordering of memory operations and so these were allowed to be executed out of order as well. In chapter 4 it was decided to give the programmer a set of memory fence instructions so that load/store dependencies could be honoured without resorting to hardware detection. This section looks at how the vector memory fences have been implemented in the PTLsim microarchitecture.

x86-64 already includes fences in its ISA for scalar memory ordering. PTLsim's implementation of these fence instructions involve stalling the frontend when renamed and only continues execution after the fence has commited. Doing this to achieve vector memory ordering would be drastic and detrimental to performance. An alternative solution is provided that has a simple implementation and much less impact on the execution pipeline.

As mentioned in chapter 4, there are three types of fence instruction. Scalar store $\rightarrow$ vector memory; vector store $\rightarrow$ scalar memory; and vector store $\rightarrow$ vector memory. The first two were used primarily for debugging and the third was used to constrain the hash probing functions. For simplicity we discuss the implementation of the third type of fence but the description is just as applicable to the other two. Figure 5.4 is provided to aid the explanation.

To create a fence it was first necessary to keep a record of the newest vector store in flight. This involved adding a register named nvsreg that points to the store's reorder buffer entry. In the rename stage of the frontend, the nvsreg is updated if the current instruction is a store. When the store commits, its ROB id is compared with the contents of the nvsreg. If there is a

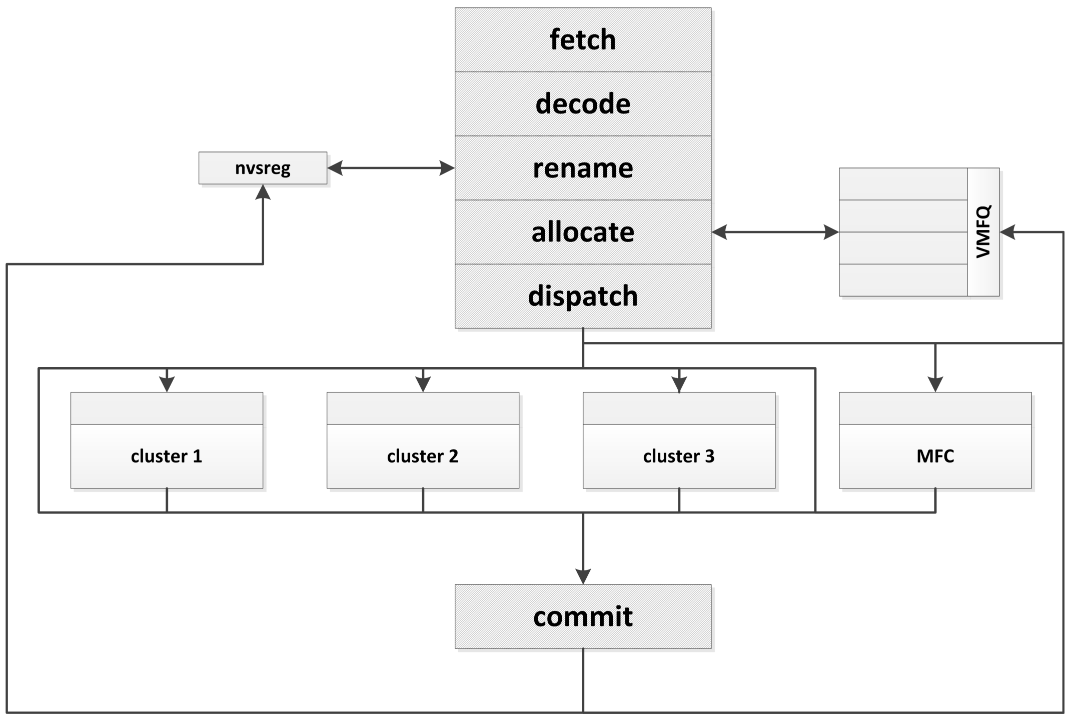


Figure 5.4: Vector Memory Fence Implementation

match, the nvsreg is nullified and if not it is left unchanged. Finding a match means that no additional store entered the frontend during the current store's entire execution.

When a fence instruction enters the pipeline, two additional steps must take place. First, the instruction takes a dependency on the ROB entry pointed to by the nvsreg. This is either the latest in-flight store or NULL. Next, the fetch must be allocated an entry in a custom structure called the Vector Memory Fence Queue (VMFQ). The VMFQ is a simple queue structure that manages any in-flight fences. It is intended to be very small (one to four entries) as the number of expected in-flight fences is quite low. In fact, in our tests no more than two fences are in flight at once. Each entry in the VMFQ contains the value of the ROB id of the associated fence instruction.

The fence instruction proceeds through the pipeline normally until the dispatch stage. Here it must be dispatched to a special cluster called the Memory Fence Cluster (MFC). The reasoning behind this is that in PTLsim, a store is considered "complete" after it has issued, and any dependents will start issuing at that point. For the memory fences to be effective, dependent instructions should not issue until stores older than the fence have committed to memory and retired from the instruction pipeline. To solve this, the issue queue in the VMFQ is disconnected from the main operand broadcast network; instead, whenever a store commits to memory it also broadcasts its ROB id to the VMFQ. Now a fence cannot issue until the store has fully retired from the pipeline. When the fence instruction commits, it frees its entry in the MFC.

The next change is concerning vector memory operations that are newer than an in-flight fence. When a vector memory instruction is allocated its ROB entry, the MFC is also checked for emptiness. If it is not empty, the current instruction must take the most recent fence as a dependency. This means that the current memory instruction can only issue after the fence has issued. If there is a misspeculated stream of instructions, during their annulment any MFC entries must be freed and the nvsreg must be corrected. Other than this, the recovery mechanism is the same as scalar instructions.

Chapter 6

Experiments

This chapter discusses the experiments and results done using the extended version of PTLsim.

6.1 Reducing VectorWise

6.1.1 Benchmark for Simulation

Simulating the entire VectorWise software would be complicated; error-prone; very slow; and the results would be difficult to interpret. [31] discusses many of issues and difficulties that arise from simulating an entire database management system. VectorWise is a multithreaded application and is broken down into server/client executables. PTLsim run in userspace mode does not support multithreaded simulations. Furthermore, this study is particularly interested in the hash join implementation discussed in chapter 3. For this reason it was decided to build a reduced standalone version of the VectorWise hash join implementation.

All of the functions and macros that comprise the VectorWise hash join were extracted into a separate program. The large hash join in TPC-H query nine was created as a benchmark for the simulation. To achieve this, the VectorWise was traced in order to obtain any input related to this query. In total, 11 GB of raw data was extracted. The caveat of doing this is that when the hash join is called in isolation, there is less interference with its working set. This could mean that the cache performance is better. Experiments varying the cache size are run later to address this.

6.1.2 Vectorising the Benchmark

In order to make a vectorised version of the benchmark, it was necessary to use the new ISA defined in appendix A. The problem is that no existing assemblers recognised this new ISA. To circumvent this, instructions were inlined into the C source code in their hexidecimal form. Listing 6.1 gives an example of the *set maximum vector length* encoded into the source file.

```
asm volatile (".byte 0xF1, 0x26, 0x00, 0x00, 0x00, 0x00");
```

Listing 6.1: inlining hexidecimal

Some vector instructions interact with the existing x86-64 general purpose registers. This means that it is necessary to control the placement of variables in the registers. GCC allows this using the register keyword. Listing 6.2 illustrates a value being assigned to the variable `t0` that is held in the register `r8`.

```
register unsigned long t0 asm ("r8")  = 1234;
```

Listing 6.2: register definition

It was also necessary to stop GCC's optimisations from eliminating registers/variables from the source code. This can happen because GCC has no notion as to what the new vector instructions actually do. To solve this, GCC's assembly input/output lists are used. This allows the programmer to inline assembly and explicitly specify what registers will be read and what registers will be written to. GCC's register allocator takes this information into account when assigning variables to registers. Listing 6.3 shows and example of the instruction *set vector length* and uses the value stored in `r11` as input. When compiled, GCC will make sure that `r11` is not modified until after this instruction.

```
asm volatile (".byte 0xF1, 0x24, 0xB0, 0x00, 0x00, 0x00"
 : /* no output */
 : "q" (r11) );
```

Listing 6.3: register dependency modelling

The reduced scalar and vector versions of the VectorWise hash join was compiled with GCC 4.0.3 with `-O6` optimisations and loop/function alignment turned on. Additionally, all the input data arrays were aligned to 64-bytes i.e. the size of a cache line. This reduces the number of cache line accesses for unit-stride vector loads/stores.

6.2 Baseline Parameters

This section lists the parameters of the baseline setup. In all the proceeding experiments, the variables are taken from these tables unless explicitly stated otherwise. The parameters of the scalar baseline are based on the Intel Westmere microarchitecture used to profile VectorWise in chapter 3. Values were obtained from the Intel Optimisation Reference Manual [28].

parameter	value
fetch width	4
frontend width	4
frontend stages	7
dispatch width	4
maximum issue width	6 (1 per cluster)
writeback width	4
commit width	4

Table 6.1: superscalar width parameters

Table 6.1 lists the superscalar parameters. These are correct except that the Westmere architecture can use micro-op fusion which occasionally allows decoding five microps into the frontend pipline. This is not modelled in PTLsim.

parameter	value
reorder buffer	128
issue queue	8 (per cluster)
load queue	48
store queue	32
fetch queue	28
outstanding L1 misses	10
outstanding L2 misses	16

Table 6.2: out of order structure parameters

Table 6.2 shows the sizes of various structures in the microarchitecture. The only difference is the issue queue configuration. In Westmere the equivalent to an issue queue is the reservation station and it is a single structure with 36 entries shared by all the clusters. It is not possible to model clusters as well as a shared issue queue in PTLsim. To solve this, the reservation station is divided into six parts (there are six clusters) and each given an extra two each to compensate. Subsequent experiments will increase each issue queue size and measure its impact on performance.

Table 6.3 shows the cache hierarchy and main memory parameters. The main memory latency was chosen based on a benchmark run the system used to profile vectorwise in chapter 3. LMbench [37] was used to generate an estimate of main memory latency. The size of working

cache level	size	latency	line size	ways	sets
level 1 instruction	32 KB	1	64	4	128
level 1 data	32 KB	4	64	8	64
level 2	256 KB	10	64	8	512
level 3	12 MB	35	64	16	12288
memory	∞	357	n/a	n/a	n/a

Table 6.3: cache hierarchy parameters

set of the application is taken into account with this.

parameter	value
physical vector registers	32
physical mask registers	32
lanes	1
chaining	on
L2 $\rightarrow$ vector register bus width	32 bytes
vector load request file entries	12
vector store request file entries	8
vector cache line request file entries	128

Table 6.4: vector parameters

Table 6.4 shows the default configuration of the vector parameters. The number of physical vector registers has been made double the amount of architectural vector registers. This is based on [15] that states for register renaming to be effective, there should be at minimum twice as many physical registers to architectural registers. We have noted that sixteen architectural vector registers is far too excessive for our workload. Six architectural registers would be enough meaning that the physical register file could be reduced to 12 entries.

The level 2 cache to vector register file bus width was made 32 bytes; this the bus width that connects the level 1 data cache to the level 2 cache in Westmere. The Vector Cache Line Request File was made 128 entries. This number was chosen to allow two indexed memory operations in flight when the maximum vector length is 64. The VCLRF is not associative and should be allowed to grow larger if space permits.

Table 6.5 shows the configuration of the clusters. The Westmere has six clusters (ports 0 to 5) and four additional clusters have been added for vector support. The scalar clusters attempt to model the Westmere as close as possible, however it is not precise due to the difference in μop generation in PTLsim and the Westmere (which is not publicly available). The scalar functional units are fully pipelined meaning that a new instruction can issue every cycle. The vector functional units are fully pipelined but with respect to a single instruction. They cannot issue a new vector instruction until the occupying instruction has fully completed. The vectors clusters vexec0 and vexec1 are able to execute any vector instructions bar memory requests and

cluster	functional unit	functional unit	functional unit	functional unit
port0	int alu	fp mult	divide	sse alu
port1	int alu	fp add	complex int	sse alu
port2	load	-	-	-
port3	store	-	-	-
port4	store	-	-	-
port5	int alu	branch	sse alu	-
vmem	memory: unit-stride	memory: indexed	-	-
vexec0	vector alu	mask alu	vector length	mask pop
vexec1	vector alu	maskalu	vector length	mask pop
vfence	fence	-	-	-

Table 6.5: cluster parameters

fences. Chaining can occur between the two exec clusters and between functional units in the same cluster.

6.3 Reducing Simulation Time

One run of execution of the scalar implementation using PTLsim takes over 110 hours, roughly five days long. This is impractical when doing many experiments that vary a single parameter of the configuration at a time. In order to reduce the time spent simulating, the number of tuples processed needed to be reduced first. This is possible because the most active parts of the code are repetitive functions executed in a loop.

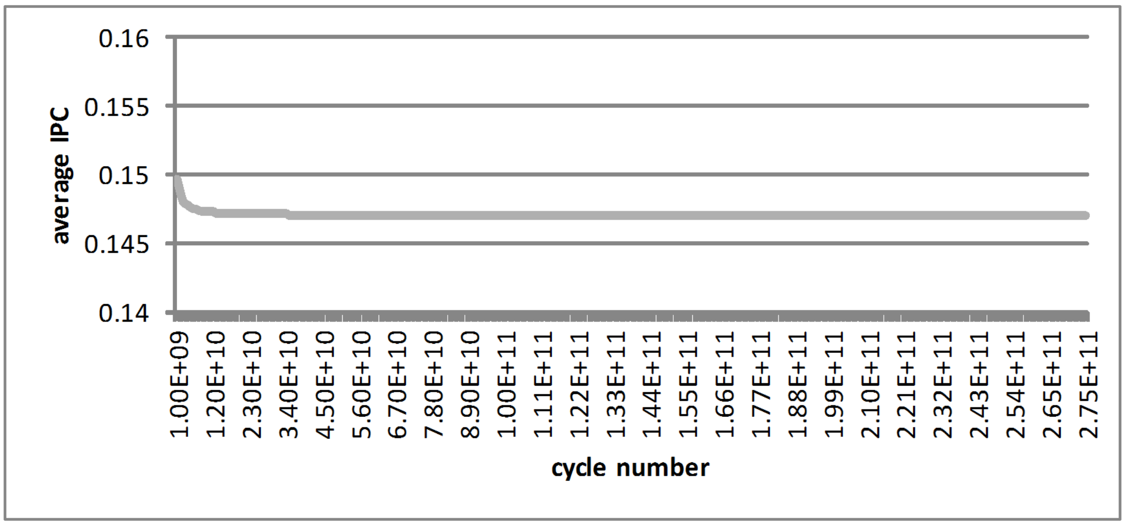


Figure 6.1: accumulated IPC over the application's full run

It was important first to verify that this is true. To do this, a full simulation was run taking a snapshot of the accumulated statistics every 1,000,000,000 simulated cycles. Each snapshot contains the average instructions per cycle at that moment. Figure 6.1 shows the average IPC plotted against the cycle number. It can be seen that the IPC is very stable and doesn't fluctuate after several billion cycles. The binary was changed from operating on 600 million tuples to 12 and a half million instead. This reduced the scalar simulation time to roughly two hours.

6.4 Results

This section covers all the simulated experiments. The VectorWise block length is fixed at 1024 until the final experiment. It should be noted that whenever the term performance is used, it is referring to a relative measure of execution time; shorter is better and longer is worse. Speedup refers to the reduction of execution time related to the scalar version of the code simulated using the default configuration.

6.4.1 Average Vector Length

The average vector length is an important statistic. It can show the (lack of) scalability of vectorised code and if there can be additional benefits increasing the maximum vector length (MVL) i.e. the size of the vector registers.

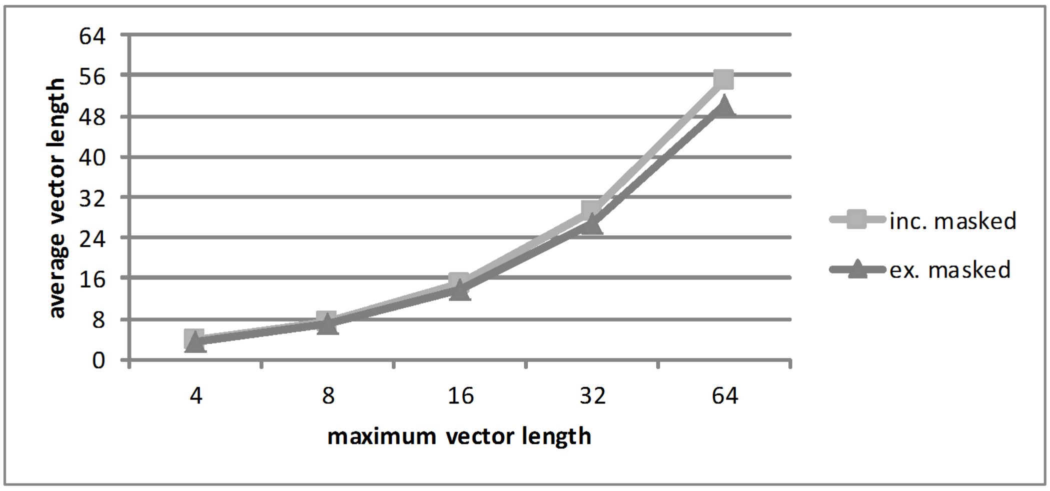


Figure 6.2: average vector length

Figure 6.2 shows the results. The horizontal axis varies the MVL whereas the vertical axis shows how many elements were operated on by a single vector instruction on average. There are two lines shown: one uses the vector length of the instruction as its reference [inc. masked];

the other uses the vector length but subtracts any elements that have been masked out [ex. masked]. The results are good and show that the application scales well whilst varying the maximum vector length. In VectorWise, the scalability should sustain until the MVL exceeds the block length.

6.4.2 Maximum Vector Length

Figure 6.3 shows the results of an experiment that varies the maximum vector length. There are two runs per MVL value, one with chaining disabled and another with chaining enabled. The results are relative to the baseline scalar performance and display the vectorised version's speedup.

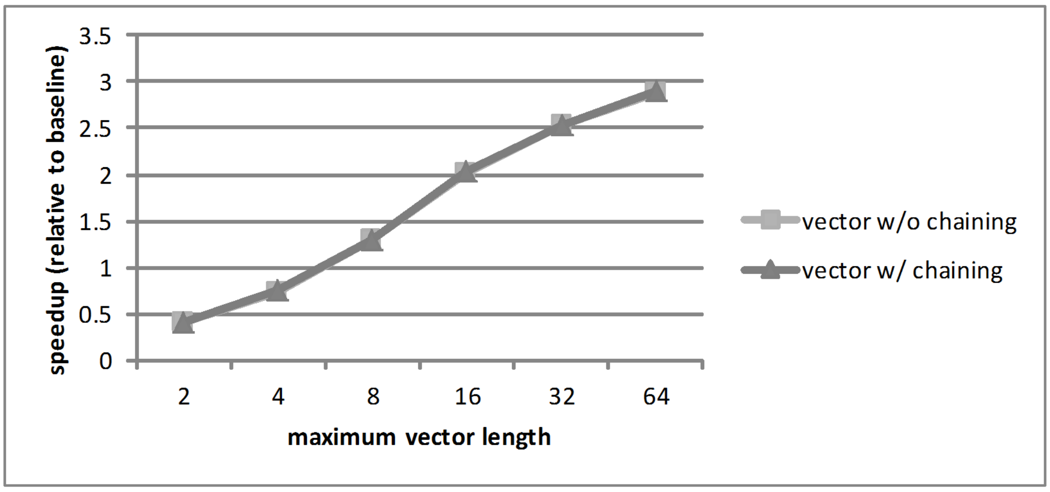


Figure 6.3: varying the maximum vector length

From the diagram it can be seen that performance is handicapped when the MVL is two or four. This most likely happens due to the extra overhead associated with accessing the L2 cache in lieu of the L1. Additionally, the advantage of accessing entire cache lines is weakened; a maximum vector length of four can address a maximum of 32 bytes (4x an 8-byte datatype) which is only half of a line cache. The performance benefits kick in when an MVL of eight is used. The performance grows steeply until an MVL of sixteen is used. After this point, performance continues to increase but at a steadier rate. With an MVL of 64 there is a performance gain of 2.9x.

It can be seen that the use of chaining offers very little benefit. There are three reasons that can explain this. The first reason is that the loads cannot be chained into other operations. The loads are keystones in the loop bodies and are often the root of dependency chains. The second reason is that there just isn't enough functional units in the configuration. The third reason is that by employing an out of order execution engine, the functional units can be kept busy without resorting to chaining.

6.4.3 Vector Lanes

Figure 6.4 shows the results of an experiment that varies the number of vector lanes. Here the MVL is fixed at 64. Like before, there is a run with chaining disabled and another run with chaining enabled.

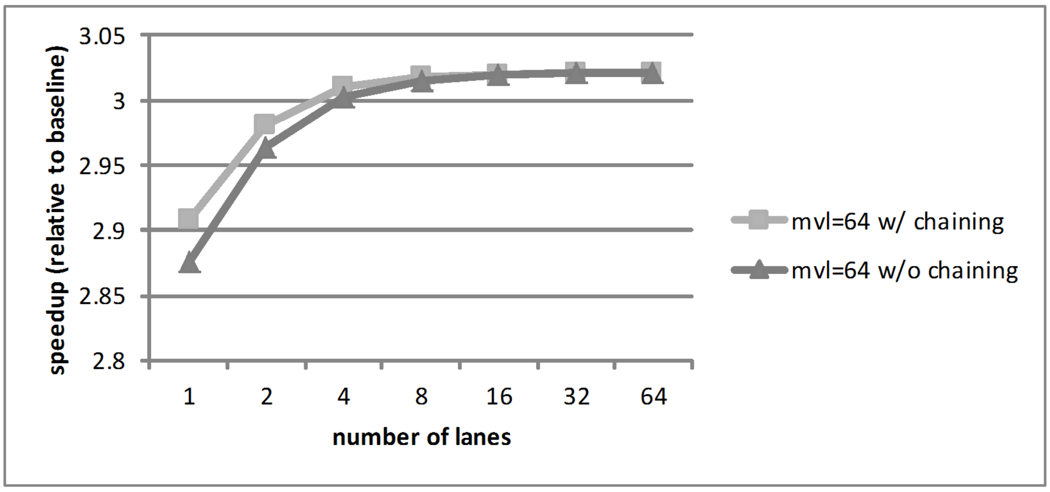


Figure 6.4: varying the number of vector lanes

The results show that lanes can bring some performance advantages. There is a steep slope between one lane and four lanes, there doesn't appear to be much more advantage when the number of lanes is greater than sixteen. The indexed memory operations can take advantage of lanes to generate multiple addresses in parallel and reducing the drawback of not allowing loads to be chained.

It should be noted that the scale of the vertical axis of this diagram is much smaller than the one in figure 6.3. Using 64 lanes over one lane will give an extra 1.3% of performance when chaining is enabled or 1.6% when chaining is disabled.

6.4.4 Cache Size

VectorWise is very conscientious of the cache hierarchy, however it was analysed in chapter 3 that the hash probing operation has a working set larger the any of the cache levels. To see how the size of the cache can impact both the scalar and vector implementations, an experiment varying the cache sizes was performed. In each experiment, the number of sets of each cache level is doubled. For simplicity the L3 has been removed from this experiment. Table 6.6 shows the configurations and figure 6.5 shows the results.

configuration	L1 size	L2 size
x	32 KB	256 KB
x2	64 KB	512 KB
x4	128 KB	1 MB
x8	256 KB	2 MB
x16	512 KB	4 MB
x32	1 MB	8 MB
x64	2 MB	16 MB
x128	4 MB	32 MB
x256	8 MB	64 MB

Table 6.6: cache hierarchy configuration

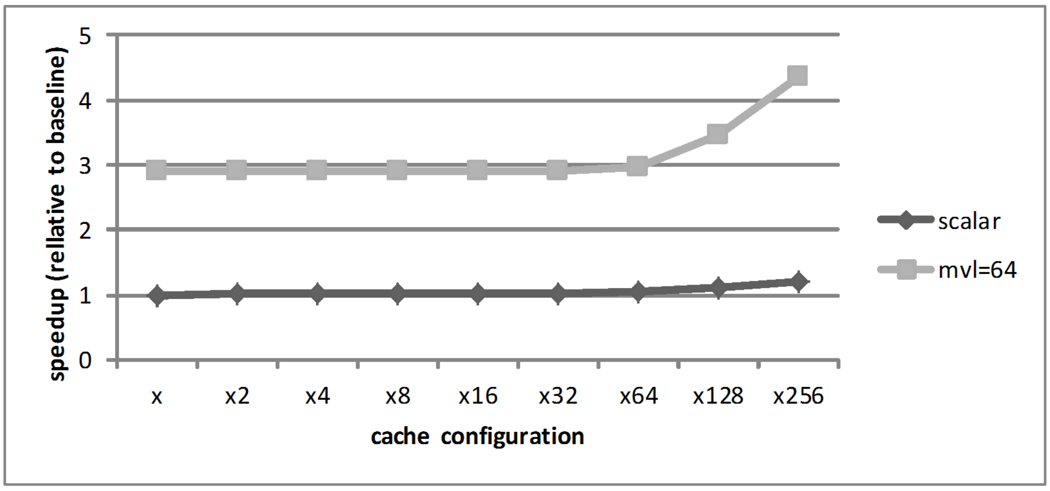


Figure 6.5: Increasing the cache size

The results show that increasing the cache size has little benefit for either the scalar or vector versions of the application. When the caches become exceedingly large and begin to comfortably fit the hash table and bucket chain (32 MB and 17 MB respectively), the performance starts to increase. Both the scalar and the vector versions of the application exhibit a similar trend except that the vector implementation benefits a bit sooner as the L2 cache is larger than the L1 cache size. There is little benefit in increasing the cache sizes unless the extra space allows the entire working set to become resident.

6.4.5 Outstanding Cache Misses

It has been demonstrated that the cache hierarchy must be unrealistically large before it can offer additional performance gains to either the scalar and vector versions of the application. It is therefore desirable that the application can issue many requests to main memory in parallel to tolerate this cache behaviour. An experiment has been performed varying the number of outstanding misses allowed in the L1 and L2 caches. Table 6.7 shows the configurations and figure 6.6 shows the results. The experiment was run for the scalar implementation and also for the vector implementation with MVLs of 16 and 64.

configuration	L1 misses	L2 misses
x	10	16
x2	20	32
x4	40	64

Table 6.7: outstanding misses configuration

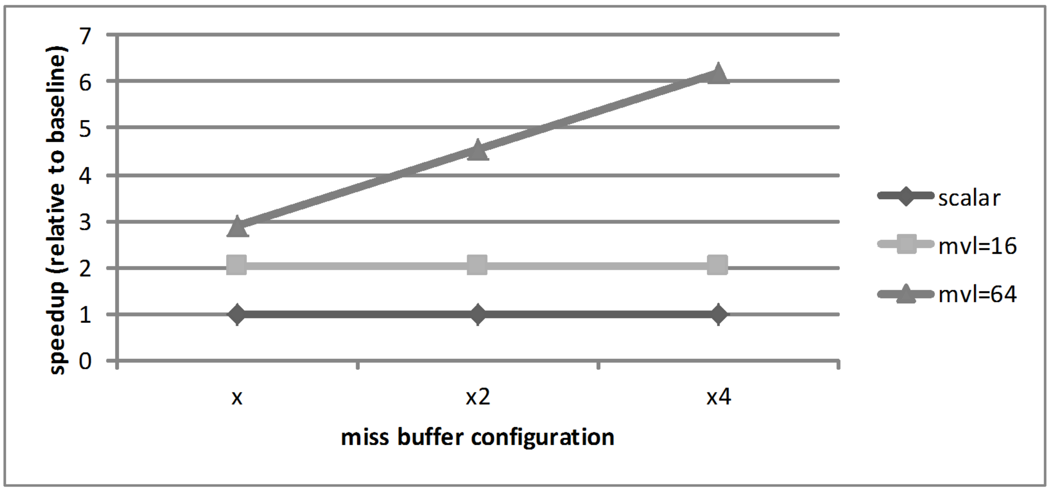


Figure 6.6: Increasing the amount of outstanding cache misses

The diagram shows that increasing the number of cache misses allowed has no influence on the scalar code's performance. The scalar implementation cannot generate cache misses fast enough to saturate the Load Fill Request Queue (LFRQ) or Miss Buffer (MB). The same can be said regarding the vector implementation when the MVL is sixteen. Although the performance is better relative to the scalar implementation, it also cannot saturate the LFRQ or MB. When an MVL of 64 is used, it generates a lot more misses in parallel. This is likely due to the indexed memory operations generating many cache line requests.

6.4.6 Latency Tolerance

Vector processors have a reputation for being more tolerant of long latency operations than scalar processors. This experiment varies the main memory latency in multiples of two, both shorter and longer than the baseline. The experiment has been run with the scalar implementation and the vector implementation using MVLs of 16 and 64. Figure 6.7 shows the results. It should be noted that the vertical axis has been changed from speedup to relative time.

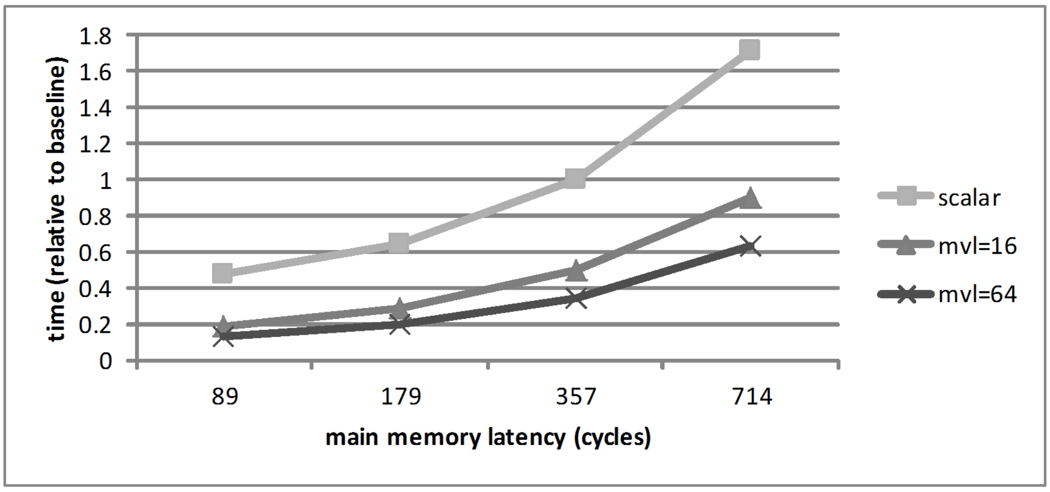


Figure 6.7: varying main memory latency

It can be seen that the main memory latency contributes quite a lot towards both the scalar and vector implementation's performance. When the baseline's main memory latency is halved, the scalar performance is 1.54x better and both vector versions perform roughly 1.7x better. When the graph is examined in the other direction and memory latency is made longer, it can be seen that the scalar implementation suffers a lot more than the vector implementation. The slope of the scalar implementation is much steeper than either of the vector implementations. It can further be observed that using a larger MVL, latency can be tolerated even more as the slope when the MVL is 16 is steeper than the slope when the MVL is 64. It can also be observed that when the latency is doubled, the vector implementations still perform better than the original baseline.

6.4.7 Number of Physical Registers

In this experiment the number of physical registers is increased. Because there are two sets of new architectural registers (16 vector registers, 8 mask registers) both of these numbers are increased in multiples of two. It should be noted that there is no equivalent experiment for the scalar version because the baseline microarchitecture does not use physical registers and instead utilises the reorder buffer for the same purpose. Table 6.8 shows the configuration figure 6.8 shows the results.

configuration	vector registers	mask registers
x	16	8
x2	32	16
x4	64	32
x8	128	64
x16	256	128

Table 6.8: physical registers configuration

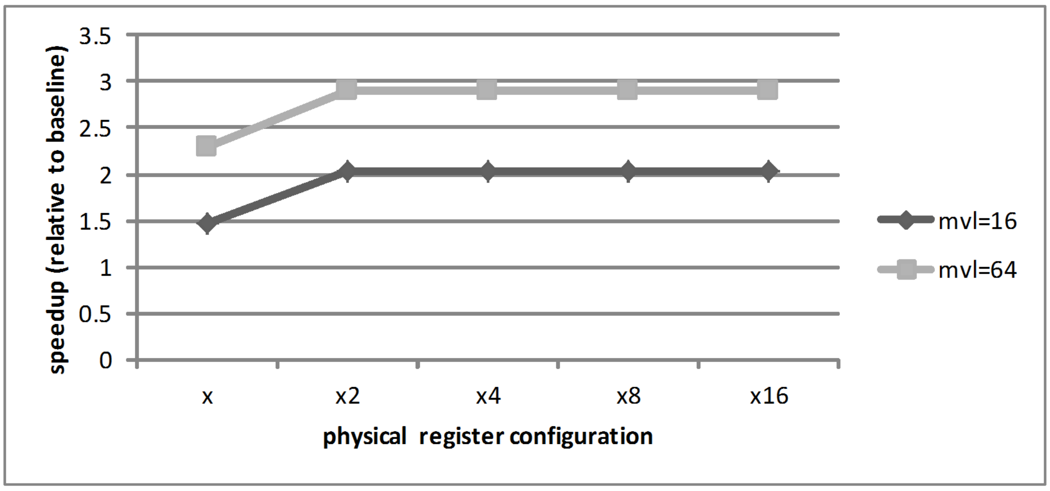


Figure 6.8: varying the number of physical registers

It can be seen a large number of physical registers isn't necessary. Double the number of architectural registers appears to be a good choice which is a similar configuration used in the vector baseline. This is primarily due to the reorder buffer becoming a structural bottleneck when the physical register file becomes excessively large. Figure 6.9 shows the same experiment but this time the vertical axis shows the number of cycles the frontend is stalled due to the reorder buffer being full.

6.4.8 Reorder Buffer Size

This experiment varies the number of entries in the reorder buffer. It is performed with the scalar implementation and the vector implementation with MVLs of 16 and 64. Figure 6.10 shows the results.

It can be seen that the baseline value (128 entries) is suitable in all cases. There is very little benefit increasing it larger than this. As mentioned in the previous section, the frontend may be

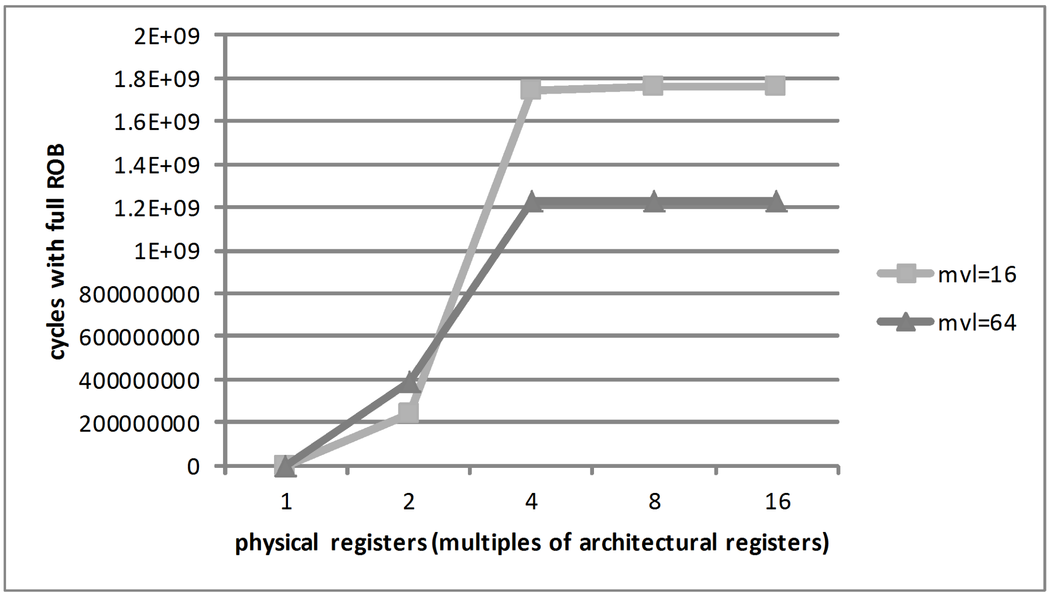


Figure 6.9: ROB bottleneck when varying the number of physical registers

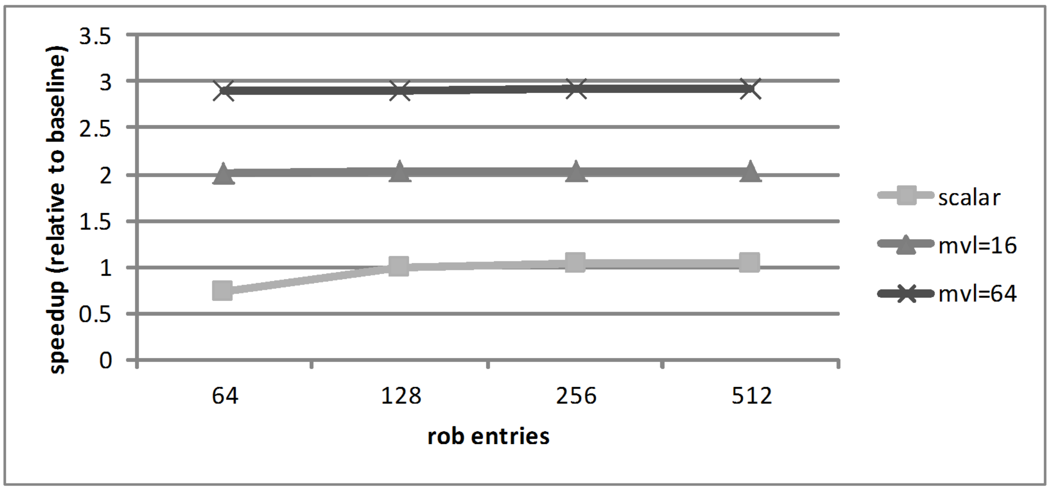


Figure 6.10: Increasing the reorder buffer size

stalling for lack of a vector physical register, a lack of a reorder buffer entry or a combination of both. An experiment is performed increasing both of these. Table 6.9 shows the configuration and figure 6.11 shows the results.

configuration	rob entries	vector registers	mask registers
x	64	16	8
x2	128	32	16
x4	256	64	32
x8	512	128	64

Table 6.9: rob and physical registers configuration

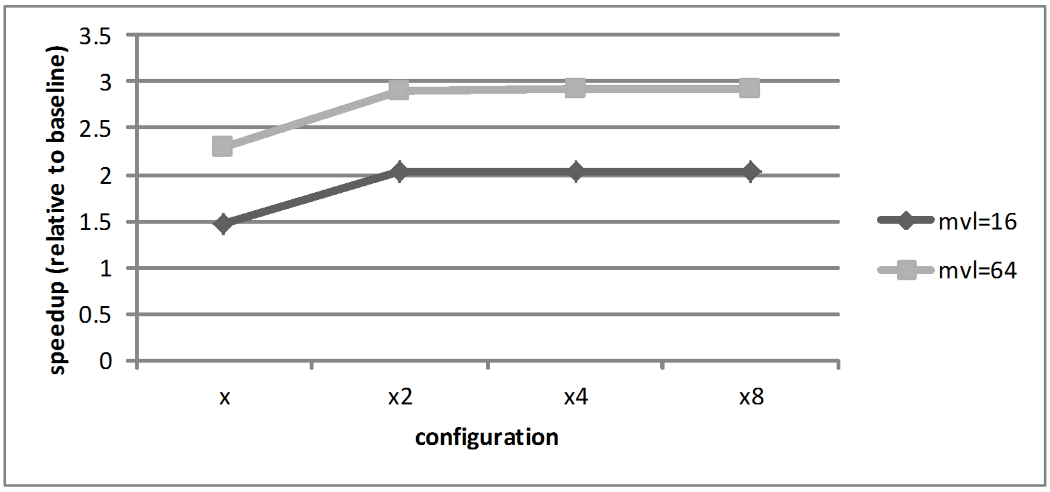


Figure 6.11: Increasing the reorder buffer size and physical registers

It is seen that even increasing the physical registers and reorder buffer together has little benefit. At this point the bottleneck becomes the number of outstanding misses allowed. It has already been shown in section 6.4.5 that for an MVL of 64, the number of cache misses can be increased without increasing the size of the reorder buffer or physical register file.

6.4.9 Issue Queue Size

This experiment varies the size of the issue queues. The baseline uses clusters to partition work and each cluster has an issue queue with eight entries. The Westmere microarchitecture used to profile VectorWise is chapter 3 has a single shared issue queue called a reservation station with 36 entries. In order to model the Westmere's clusters using PTLsim, it was necessary to partition this. This experiment increases the issue queue size of *each* cluster, therefore any benefits of using a large shared issue queue should be revealed. Figure 6.12 shows the results.

It can be seen that there are some additional scalar performance by increasing the issue queues from eight to sixteen, but it is relatively small at 1.12x. After sixteen entries there is no additional performance gains. The vector implementations show no improvements from increasing the issue queues.

6.4.10 Load Units

One feature that may be seen as unfair when comparing the vector baseline to the scalar baseline is the distribution of memory functional units. The scalar baseline has one load and two store

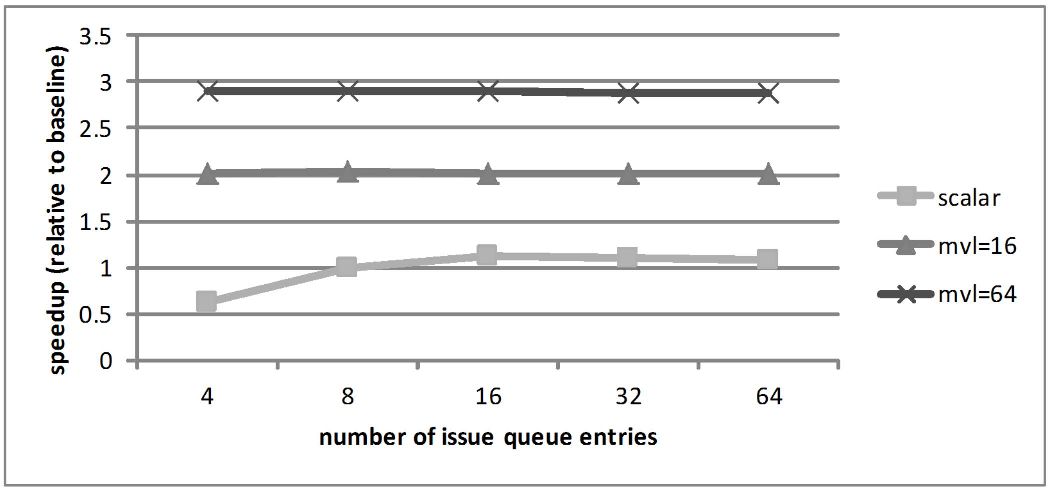


Figure 6.12: varying the size of the issue queues

functional units as found in the Westmere microarchitecture. The vector baseline has two memory functional units: one for unit-stride requests and another for indexed requests, these are used by both loads and store instructions. The ability to issue two loads in parallel might be seen as an unfair advantage. To address this, an experiment is performed increasing the number of load functional units in the scalar baseline. Figure 6.13 shows the results.

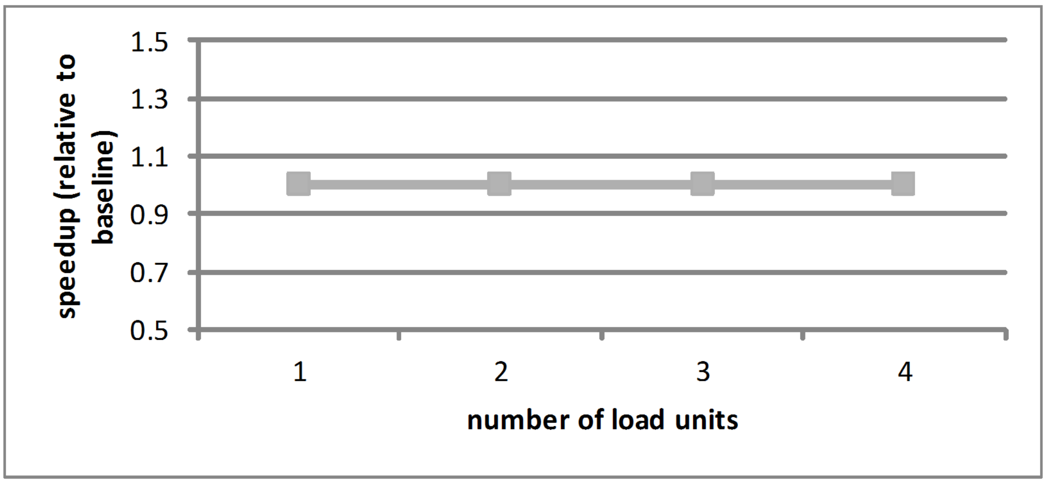


Figure 6.13: increasing the number of scalar load functional units

Any benefit is barely measurable and too small to be justifiable. The problem is that the issue queue can't find enough loads to issue in the same cycle. Figure 6.14 shows a breakdown of the number of loads that can be issued in each cycle. One load unit is sufficient and two units can give a very small amount of extra benefit. Anything more than two is overkill. This could be due to the fact that the load units are fully pipelined and so multiple units won't have a major impact on performance.

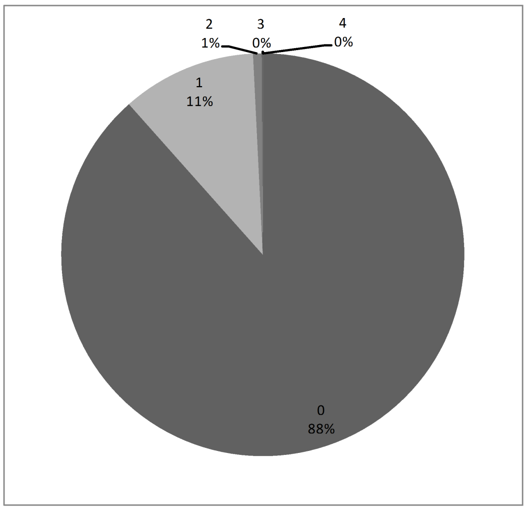


Figure 6.14: number of loads issued each cycle

6.4.11 Super Superscalar

This experiment tries to resolve the scalar baseline's bottlenecks by increasing all of its major structures at once. Here the following structures are increased in multiples of two:

- reorder buffer
- issue queues
- load queue
- store queue
- outstanding L1 misses
- outstanding L2 misses

Table 6.10 shows the configuration and figure 6.15 shows the results. Doubling the amount of resources available does boost the performance, but quadrupling the resources doesn't have the same impact. With four times the resources, there is only an improvement of 1.56x. This is in stark contrast to the vector performance increase of 6.18x in section 6.4.5. It should be noted that many of these structures are associative and cannot be scaled anyway. This demonstrates that a vector architecture can scale much better with fewer resources than a scalar architecture.

6.4.12 VectorWise Block Length

In this experiment we vary the VectorWise block length i.e. how many tuples are processed together in a group. Figure 6.16 shows the results.

configuration	rob	issue queue	load queue	store queue	outstanding L1	outstanding L2
$x\frac{1}{2}$	64	4	24	16	5	8
x	128	8	48	32	10	16
x2	256	16	96	64	20	32
x4	512	32	192	128	40	64

Table 6.10: major structure configurations

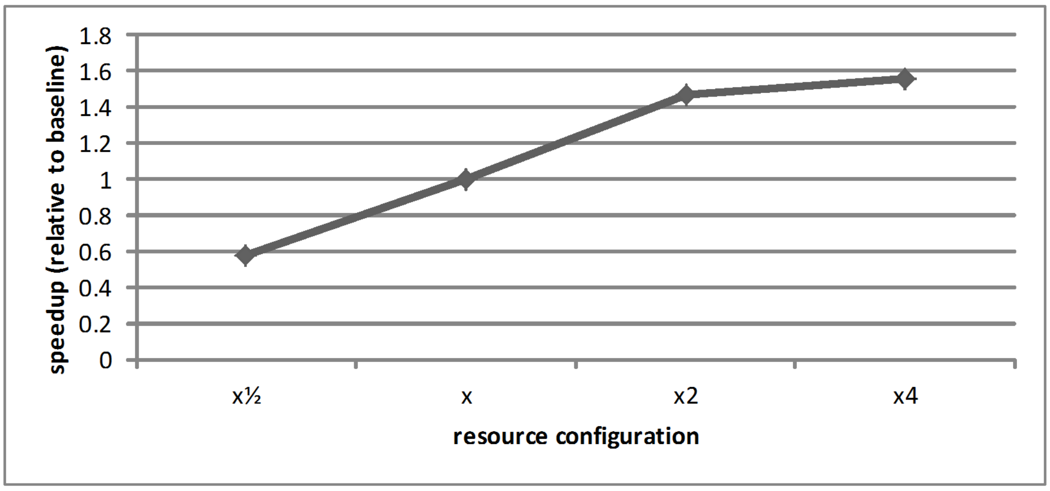


Figure 6.15: increasing multiple scalar structures at once

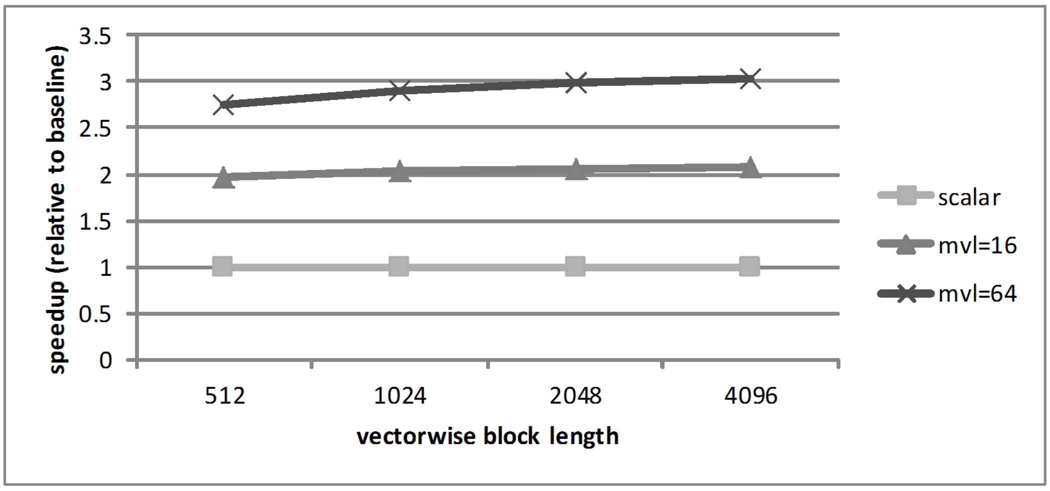


Figure 6.16: increasing the vectorwise block length

It can be seen that the scalar implementation has no additional benefit from increasing the block size larger than its default of 1024. For the vector implementations, there is an additional 2% performance increase when the MVL is 16 and 4.4% when the MVL is 64.

Chapter 7

Related Work

This chapter lists various pieces of research that share something in common with our work. First we list various research related to database acceleration through data-level parallelism. After, a list of prior vector research is given. For all related works, a list of commonalities and differences are given.

7.1 DBMS Acceleration

This section examines a list of works related to database acceleration through data-level parallelism.

7.1.1 DBMS and Vector Architectures

[36] is the earliest work found looking at database operator implementations on vector processors. The report looks specifically at the hash join operator and implements a vectorised version for the Cray C90 [39].

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 joins with no reference to a real DBMS. Additionally, we are proposing vector extensions to an ISA that already dominates the server marker whereas this work is done exclusively on a supercomputer. Additionally, their reference scalar implementation is quite naive whereas ours is the basis for a commercial product.

In a similar vein to [36], [38] also looks at vectorising database operators. This time the list is expanded to selection, projection and join operators however the methodology is still the same. Naive scalar implementations are run against vectorised versions on a supercomputer and so the same arguments still apply.

7.1.2 DBMS and SIMD Multimedia Extensions

[59] is a broad study accelerating various database operators using the existing SSE instruction extensions for x86. The work investigates the benefits of data-level parallelism and reduction of conditional branches in implementations of scans, aggregations, indexed operations and joins. Since our work is primarily focused on joins, a comparison of this feature is given.

The principal difference between our vectorised join and their SIMDised join 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 left hand side table with every row from the right hand side table. This is not a problem when the tables are small, but if they are large this is a very inefficient join algorithm. In contrast, we look at a hash join implementation which is suitable for large tables typically found in decision support systems.

7.1.3 DBMS and the Cell

[23] is a study that ports a DBMS to the Cell Broadband Engine [20], an architecture abundant with data-level parallelism capabilities. What is interesting is that the query engine used in the study is MonetDB/X100 [9] which is an earliy 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 one which lacks a join operation. Our work is primarily focused on joins so it is difficult to make a comparison.

7.1.4 DBMS and GPGPU

[22] is a very comprehensive work investigating the performance benefits of running DBMS operations on graphics processing units (GPUs) [40]. The study includes a hash join implementation that runs on a GPGPU coprocessor. There are some performance benefits but 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.

7.2 Previous Vector Architectures

The ISA extensions designed and implemented in this work were customised to accelerate hash joins when implemented using a block at a time query engine. Many of the design decisions were inspired by previous work in vector architectures. This section details a handful of vector microprocessors from the past fifteen years which were considered relevant to our work.

7.2.1 Torrent-0

The Torrent-0 (T0) [3] was the first single-chip vector microprocessor. It introduced the VMIPS ISA, a set of vector extensions for the RISC-like MIPS ISA. It used vector registers, a mask register and a vector length register. Its pipeline was minimal and in order. There was a single scalar unit, a single vector memory unit and a two vector execution units that could support chaining; all of these shared the same fetch and decode stages. It had eight lanes to support parallel execution of vector instructions. It could support both strided and indexed memory accesses.

The T0 was a very ambitious piece of work and many features such as vector registers, masking, chaining, parallel lanes and non unit-stride memory accesses have been incorporated into our design. The major difference is that we have taken an existing scalar microarchitecture and modified it to support vectors. In contrast, the T0 was designed from scratch with a focus on vector processing and consequently the scalar capabilities were made very simple. Additionally, the T0 uses an in order pipeline whereas our design is out of order.

7.2.2 Out of Order Vector Execution

[15] applies the concepts of register renaming and out of order execution to vector processing. The advantages of this are 1) a reduction in anti-dependencies, 2) a good mechanism for implementing precise interrupts and 3) that the functional units can be kept busy with useful work a larger proportion of the time than with an equivalent in order model. The principal advantage is that an out of order model can keep the memory port busy and hide long memory latencies even further.

Our design draws on this work primarily for the benefits of out of order memory scheduling. The main differences are that our work takes an existing out of order superscalar microarchitecture and appends vector support to it. In contrast, [15] modifies a Convex C3400 - a vector supercomputer architecture. It also focuses on the SpecFP92 and Perfect Club benchmarks, whereas our work is primarily concerned with decision support acceleration.

7.2.3 Scalar Processor Integration

[45] takes an existing superscalar microarchitecture, the MIPS R10000, and extends it with a vector register file and complementary instruction set. Its primary contribution is the integration of vector support with the existing cache hierarchy. Its novel approach is to bypass the level 1 data cache and go directly to the level 2 cache. This way the level 1 cache access times won't be compromised thus not interfering with the existing scalar performance. There is extra latency when accessing the level 2 cache in lieu of the level 1, but this can be hidden with a vector instruction's natural latency-tolerant properties. It takes advantage of the line-based banking scheme found in many level 2 cache designs. Using the level 2 cache over the level 1 cache also implies the application's working set can be larger.

Our work uses the level 2 cache idea from this research. [45] makes uses of cache banking to access multiple cache lines in parallel; in contrast our design is more simple and accesses a maximum of one cache line per cycle. [45] supports only strided memory accesses whereas our work supports indexed memory accesses.

7.2.4 Alpha Tarantula

Tarantula [14] was a proposed extension to the Alpha 21464 that would have appended the Alpha ISA with a rich set of vector instructions. Its ideas drew on those presented in [15] and [45] and added a lot of new functionality. Its vector support was fully integrated with the Alpha's virtual memory cache coherency scheme. It supports very fast non unit-stride memory accesses and indexed memory operations.

Our design is very much inspired by Tarantula. There are many features in common including the scalar core integration, out of order vector execution, indexed memory operations and L2 cache access. Tarantula is integrated into the existing 21464 without compromising its performance; our design also integrates the vector functionality into an existing x86-64 microprocessor, but we attempt to reuse as much as possible.

Our study focuses on specifically on decision support systems, a class in the business applications domain, which exhibits different properties to multimedia and scientific applications. Tarantula was benchmarked against multimedia and scientific applications that can benefit from many arithmetic functional units. Our design does not require so many functional units as our application is mostly memory bound. We demonstrated that adding parallel lanes to our design only makes a marginal difference whereas with Tarantula and its benchmark suite, the difference is more profound.

7.2.5 Intel Larrabee

[49] is a many-core x86 architecture which supports vectors through Intel's new LRBni instruction set [1]. Larrabee had been intended to compete with graphics processing units by offering similar performance whilst minimising the amount of fixed-function logic thus allowing the programmer to configure and fine-tune their own algorithms. Larrabee contains 32 512-bit vector registers and 8 16-bit mask registers. LRBni supports many useful instructions not found in Intel's mainline SIMD multimedia extensions such as gather/scatter and compress/expand.

There are several differences between our work and Larrabee. First, Larrabee's target market was principally the graphics/gaming domain whereas our work is focused on business applications. Intel has since decided to retarget Larrabee for the supercomputing market [53] however the same argument still appleis as multimedia and scientific applications tend to exhibit different behaviour to business applications [4].

Larrabee's cores are based on a modified P5 microarchitecture [2] and are entirely in order. Our microarchitecture is out of order and have more in common with the P6 microarchitecture [43, 21]. Larrabee's maximum vector length is limited to sixteen 4-byte values or eight 8-byte values. Our work experiments with different maximum vector lengths and demonstrates performance gains when the value is large (64 values). Larrabee's vector memory operations go the level 1 data cache and are optimised when all the elements hit the same cache line. Our implementation works with the level 2 cache and it is expected that our memory operations will address several cache lines in a single instruction. Additionally it uses subword parallelism where we omit this feature from our design.

Chapter 8

Conclusion

8.1 Overview of Work

This work took the application VectorWise, a highly-optimised OLAP DBMS, and used it as a vehicle to investigate vector microarchitecture. First, the application was profiled for bottlenecks. We found that performance was often stifled due to the ineffectiveness of exposing data-level parallelism through instruction-level parallelism. In particular, we looked at the hash probe phase of the hash join operator. We found that due to the data structures being larger than the cache hierarchy, many cycles are stalled waiting for cache misses.

From this, a set of vector instruction extensions were proposed for the x86-64 ISA. There was discussion as to problems of the existing SIMD multimedia instructions and their lack of expressibility to capture the available data-level parallelism in the profiled part of the application. Next, a microarchitecture was conceived, designed and implemented using PTLsim, a cycle-accurate x86-64 microprocessor simulator. One of the challenges was to get as much reuse as possible from the existing microarchitecture without compromising the existing scalar performance.

After all these steps, a rich set of experiments were performed comparing the original scalar version of the program against a hand-coded vectorised version. Many configurations were tested and ultimately the consensus is that the scalar microarchitecture doesn't scale well even when increasing all the major hardware structures. On the other hand, the vector microarchiecture exhibits enormous amounts of scalability and performance benefits. Our results show that using a vanilla vector baseline, there are performance speedups of **1.3x** for a short maximum vector length (8) and **2.9x** for a longer maximum length (64). When variables such as the number of outstanding cache misses are increased proportionally, the vector units can saturate these structures and increase the speedup to **6.2x** with room for additional scaling. In contrast, the scalar unit couldn't take advantage of this at all. From the perspective of the original goal, to accelerate a decision support system using vectors, this was successful.

8.2 Reflection on DSS Acceleration

Looking for problems in database software and trying to fix them with hardware alone comes with its own set of issues.

8.2.1 Software Versions

The life cycle of software is generally more rapid and dynamic than that of hardware. Hardware designers have generally employed benchmarks such as SPEC to evaluate their ideas. SPEC has a new version approximately every five years which gives the computer architecture research community something stable to work with. Using commercial software with frequent updates makes it harder to evaluate a piece of research in this.

When we began working with VectorWise, we chose a stable edition and stuck with it out of familiarity. As time passed and our research progressed, there were several version updates to VectorWise. These updates brought new functionality and features to the software, but in order to retain stability and consistency in our experiments we continued to work with the initial stable version. It is possible that some of the bottlenecks we found had been addressed in other ways using only software. We know that the VectorWise team have had some success decreasing the impact of cache misses in hash joins by using bloom filters, purely a software optimisation. We intend to continue looking at VectorWise and pay more attention to the features in more recent releases.

8.2.2 Query Plans

Aside from software updates, there is another variable that contributes to a DBMS's performance. These are the query plans that are executed by the DBMS. A problem can be solved in many ways, and in the case of query plans, a very large number of ways. When we started this piece of work, we benchmarked VectorWise with a set of query plans included in the software package. Much like the software version, these query plans became static in our research. We looked at VectorWise's research and saw that they could get radically different performance results by changing the query plan. The following is a short list giving examples of some of the things that can affect a query plan's behaviour.

Query tree arrangement: A query node's branches can be arranged so that they are spread out and balanced or heavier on one side.

Query tree order: The order in which individual operators are resolved with respect to each other. For example, if there are two criteria in a selection it makes sense to evaluate the one with the lowest selectivity first and filter the remainder out using the other with high selectivity.

Hash join variations: There are different ways of implementing hash joins. There are variations for when the left hand side can have an abitrary number of matches in the right hand side; other variations for when the left hand side has exactly one match in the right hand side; and others where the left hand side side can have exactly one or zero matches on the right hand side. Choosing the correct version can impact performance.

Alternative joins: Sometimes hash joins are not even the most suitable join to begin with. Sort-merge joins are another major algorithm used in decision support systems and have very different features and requirements to that of hash joins.

8.3 Reflection on Microarchitecture Simulation

The biggest challenge with hardware simulation is knowing when you are modelling something correctly and knowing when you are cheating. In our approach to design and implementation, simple designs were used in favour of overly-ambitious complicated ones. The latter would be far too difficult to verify with simulators such as PTLsim.

That being said, the opposite is also true. Hardware simulators, in our case PTLsim, often oversimplify their hardware. In PTLsim we ran our experiments with a very uncomplicated memory model. Main memory has one configurable parameter: latency; TLBs aren't modelled; and there is no simulated hardware prefetcher. We tried to be as careful as possible to ensure that if the scalar performance would suffer due to one of these abesent features, then the vector performance should as well. Because of the significance of memory in our profiled application, the impact of a very incorrect memory model would be catastrophic.

8.4 Future Work

8.4.1 Simulator

On the simulator side of the research, we plan to improve the cache to register file interface. Right now the design is very simple and will sometimes serialise memory operations unnecessarily. Future work will investigate the potential of parallel accesses to the cache hierarchy and more advanced forms of cache line coalescing. Additionally it would be very beneficial to look at a simulator such as DRAMSim2 [47] in order to build a more realistic memory model. Finally, many statements were made regarding vector technology and energy/power savings. Power estimate functionality should be added to the simulator so we can measure the impact of our new hardware additions.

8.4.2 DBMS

Although this work looked at a very specific part of VectorWise, many ideas related to DBMS acceleration were generated in the process. To make this study complete, we would also like to look at the build phase of hash joins and investigate the possibility of using atomic vector instructions [33] for acceleration. We also would like to look at alternative join algorithms, for example the sort-merge join and best-effort partitioning hash joins [60], and investigate if vector architectures can aid their performance too.

Bibliography

[1] Michael Abrash. A First Look at the Larrabee New Instructions (LRBni). `http://drdobbs.com/high-performance-computing/216402188`, 2009. accessed on 2011-09-08.

[2] Donald Alpert and Dror Avnon. Architecture of the pentium microprocessor. *IEEE Micro*, 13:11–21, May 1993.

[3] Krste Asanović. *Vector Microprocessors.* PhD thesis, University of California at Berkeley, 1998.

[4] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, ISCA '98, pages 3–14, Washington, DC, USA, 1998. IEEE Computer Society.

[5] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler. Flashing databases: expectations and limitations. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 9–18, New York, NY, USA, 2010. ACM.

[6] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[7] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.* Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[8] 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, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[9] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[10] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185, 1994.

[11] Robert R. Collins. In-Circuit Emulation: How the Microprocessor Evolved Over Time. `http://www.rcollins.org/ddj/Sep97/`, 1997. accessed on 2011-09-11.

[12] 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, New York, NY, USA, 1985. ACM.

[13] 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.

[14] 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, Washington, DC, USA, 2002. IEEE Computer Society.

[15] 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 30, pages 160–170, Washington, DC, USA, 1997. IEEE Computer Society.

[16] 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, New York, NY, USA, 1998. ACM.

[17] Vivek Gite. Linux HugeTLBfs: Improve MySQL Database Application Performance. `http://www.cyberciti.biz/tips/linux-hugetlbfs-and-mysql-performance.html`, 2009. accessed on 2011-09-08.

[18] G. Graefe. Volcano&#151 an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6:120–135, February 1994.

[19] Lynn Greiner. What is Data Analysis and Data Mining? `http://www.dbta.com/Articles/Editorial/Trends-and-Applications/What-is-Data-Analysis-and-Data-Mining-73503.aspx`, 2011. accessed on 2011-09-11.

[20] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, 26:10–24, March 2006.

[21] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[22] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34:21:1–21:39, December 2009.

[23] 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 '07, pages 4:1–4:6, New York, NY, USA, 2007. ACM.

[24] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition.* Morgan Kaufmann, San Diego, 2007.

[25] R. G. Hintz and D. P. Tate. Control data star-100 processor design. In *IEEE Computer Society Conf.*, COMPCON 72, pages 1–4, 1972.

[26] 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.

[27] Ingres. Ingres/VectorWise sneak Preview on the Intel Xeon Processor 5500 series-based platform. Technical report, Ingres, 2009.

[28] Intel. Intel®64 and ia-32 architectures optimization reference manual, June 2011.

[29] Intel. Intel®advanced vector extensions programming reference, June 2011.

[30] Intel. Intel®VTune$^{\text{TM}}$Amplifier XE, 2011.

[31] Kimberly Keeton and David A. Patterson. Towards a simplified database workload for computer architecture evaluations. In *In Workload Characterization for Computer System Design, edited byh*, pages 115–124. Kluwer Academic Publishers, 2000.

[32] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 195–201, New York, NY, USA, 1998. ACM.

[33] 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, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Patrick Kurp. Green computing. *Commun. ACM*, 51:11–13, October 2008.

[35] 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, New York, NY, USA, 2006. ACM.

[36] Rich Martin. A Vectorized Hash-Join, 1996.

[37] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[38] Shintaro Meki and Yahiko Kambayashi. Acceleration of relational database operations on vector processors. *Systems and Computers in Japan*, 31(8):79–88, 2000.

[39] Wilfried Oed and Martin Walker. An overview of cray research computers including the y-mp/c90 and the new mpp t3d. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, SPAA '93, pages 271–272, New York, NY, USA, 1993. ACM.

[40] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[41] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society.

[42] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25:206–218, May 1997.

[43] David B. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, 16:8–15, April 1996.

[44] Y. N. Patt, S. W. Melvin, W. M. Hwu, M. C. Shebanow, and C. Chen. Run-time generation of hps microinstructions from a vax instruction stream. In *Proceedings of the 19th annual workshop on Microprogramming*, MICRO 19, pages 75–81, New York, NY, USA, 1986. ACM.

[45] 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, New York, NY, USA, 1999. ACM.

[46] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20:47–57, July 2000.

[47] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10:16–19, January 2011.

[48] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21:63–72, January 1978.

[49] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[50] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[51] Nathan T. Slingerland and Alan J Smith. Multimedia instruction sets for general purpose microprocessors: a. Technical report, Berkeley, CA, USA, 2000.

[52] 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*, ISCA '00, pages 260–269, New York, NY, USA, 2000. ACM.

[53] John Stokes. Intel takes wraps off 50-core supercomputing coprocessor plans. `http://arstechnica.com/business/news/2011/06/intel-takes-wraps-off-of-50-core-supercomputing-coprocessor-plans.ars`, 2011. accessed on 2011-09-11.

[54] Transaction Processing Performance Council (TPC). TPC-H Standard Specification v2.14.2. Technical report, 2011.

[55] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2:385–394, August 2009.

[56] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *in ISPASS 07*, 2007.

[57] Matt T. Yourst. PTLsim Users Guide and Reference. Technical report, 2007.

[58] Hui Zeng, Matt Yourst, Kanad Ghose, and Dmitry Ponomarev. Mptlsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH Comput. Archit. News*, 37:2–9, July 2009.

[59] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[60] Marcin Zukowski, Sándor Héman, and Peter Boncz. Architecture-conscious hashing. In *Proceedings of the 2nd international workshop on Data management on new hardware*, DaMoN '06, New York, NY, USA, 2006. ACM.

[61] Marcin Zukowski, Niels Nes, and Peter Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th international workshop on Data management on new hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. ACM.

Appendix A

Vector ISA

This is a formal definition of the vector instruction set architecture used in this project. This can be seen as an extension to the x86-64 ISA. There is a very important difference here between the format of x86-64 instructions and the format of our extensions. The former works with two operands where one of the sources must also be a destination (instructions are destructive to its operands). This work creates a VMIPS-style ISA [3] for extra flexibility which can have a different destination register than one of our source operands.

A.1 Registers

The ISA description refers to architectural registers using the following shorthand:

- `vl`: vector length register
- `sr[0]` → `sr[15]`: general purpose scalar registers (these are the existing sixteen 64-bit registers in x86-64)
- `vr[0]` → `vr[15]`: general purpose vector registers
- `vm[0]` → `vm[7]`: vector mask registers

A.2 RISC or CISC

x86-64 (and to a larger extent x86) is a CISC ISA for three reasons

1. the instructions are variable length necessitating complex decoding logic
2. work (e.g. arithmetic) instructions can take operands from registers and/or main memory
3. many instructions express a large amount of work e.g. repeated string instructions

Which category do these vector instructions fall into? The three points are addressed as follows:

1. the vector ISA will be fixed length in order to have a simple encoding mechanism for our assembler (an added bonus is simple decoding logic)
2. the vector ISA will require all work instructions to read from and write to architectural registers
3. a vector instruction represents a lot of work, a lot more than a μop, but the work is regular and simple

It's difficult to place the vector ISA into either group, so a new term is invented: **R**educed **C**omplex **I**nstruction **S**et **C**omputing - **RCISC**.

A.3 Masks

There are eight mask registers but in all the vectorised kernels the maximum ever used is three, why so many? It's nice to have some flexibility with mask manipulation and give the ISA more than one mask. Eight mask registers can be addressed with 3 bits and then one more bit can be used to determine if the mask should be used or not. This is because a mask is optional for most of the instructions. It is far easier to deal with nyble than two or three bits when using a disassembler.

code	mask	code	mask
0x0	vm0	0x8	NOMASK
0x1	vm1	0x9	NOMASK
0x2	vm2	0xA	NOMASK
0x3	vm3	0xB	NOMASK
0x4	vm4	0xC	NOMASK
0x5	vm5	0xD	NOMASK
0x6	vm6	0xE	NOMASK
0x7	vm7	0xF	NOMASK

Table A.1: mask encoding

Logical bitwise operations as well as POPCNT already exist in x86-64. Why are there unique instructions for masks? Because the size of the mask relates to the size of a vector register. Right now, the largest datatype for x86-64 is 64 bits. If the maximum vector length is increased to > 64 the masks cannot not be stored in a single general purpose register (GPR) thus complicating things quite a bit for the programmer/compiler.

A.4 Datatypes

A nyble is used to encode the following datatypes into the instructions. Using a nyble allows the ISA to be extended later when non-integer datatypes are needed.

symbol	type	size	code
sc	signed char	1	0x0
ss	signed short	2	0x1
si	signed int	4	0x2
sl	signed long	8	0x3
uc	unsigned char	1	0x4
us	unsigned short	2	0x5
ui	unsigned int	4	0x6
ul	unsigned long	8	0x7

Table A.2: datatype encoding

In x86-64 there is implicit zero extension of 32-bit values loaded into 64-bit registers Because there is no subword parallelism in this vector ISA, integer datatypes can be sign extended.

A.5 General Format

To extend the x86-64 ISA, a prefix is used to distinguish the new instructions from existing ones from Intel/AMD. The following single-byte prefix is used: `0xF1`. This can work however the prefix actually is reserved for In-Circuit Emulation [11] breakpoints. PTLsim doesn't decode any instruction with this opcode/prefix so it is safe to use for simulation purposes.

The longest instructions in the ISA (vldi and vsti) require five nybles for to encode operands. These together with the single-byte prefix and single-byte opcode suggest a four and a half byte wide instruction. This is extended to six bytes to reserve space for future instructions and, additionally it will be easier to view six-byte instructions in a disassembler over four and a half.

```
[0xF1] [opcode] [0x??]  [0x??]  [0x??]  [0x??]
```

A.6 Dependencies

In the instructions that follow, there is a table that lists various properties of the instructions. Among them are *src* and *dst*. These stand for source register and destination register respectively. The purpose is to highlight anywhere there is a data dependency between architectural registers to avoid various hazards when pipelining and out of order execution is introduced. Datatypes and immediates are not considered sources in this sense however the vector length register is considered a source even though it is never encoded into the instruction itself.

One important observation to make is that the destination is generally considered a source as well. When there are operations which are masked or where the vector length is less than MVL, the orthodox behaviour is that the old values of the architectural registers will remain intact. Unfortunately this implies slightly more complicated hardware and in some cases may hamper out of order execution.

A.7 Instructions

The new instructions are classified into nine categories

- memory A.7.1
- initialisation A.7.2
- arithmetic A.7.3
- logical A.7.4
- comparisons A.7.5
- position manipulation A.7.6
- mask A.7.7
- vector length A.7.8
- memory fences A.7.9

A.7.1 memory

- The base address must be aligned to the datatype.
- The base address must be in x86-64 canonical form.

vector load: unit stride

mnemonic	opcode
vldus	0x01

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values to load
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
scalar	y	y	n	1	base	64-bit base address

```
dtype* array = (dtype*) base;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = array[i];
  }
}
```

vector store: unit stride

mnemonic	opcode
vstus	0x02

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	n	1	$\vec{vra}$	vector of source values
scalar	y	y	n	1	base	64-bit base address

```
dtype* array = (dtype*) base;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
     array[i] = vra[i];
  }
}
```

vector load: indexed

mnemonic	opcode
vldi	0x03

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values to load
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
scalar	y	y	n	1	base	64-bit base address
vector	y	y	n	1	$\vec{vra}$	vector of memory offsets

```
dtype* array = (dtype*) base;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
     vrd[i] = array[ vra[i] ];
  }
}
```

vector store: indexed

mnemonic	opcode
vsti	0x04

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values to store
mask	n	y	n	1	$\vec{mask}$	optional mask register
scalar	y	y	n	1	base	64-bit base address
vector	y	y	n	1	$\vec{vra}$	vector of memory offsets
vector	y	y	n	1	$\vec{vrb}$	vector of values

```
dtype* array = (dtype*) base;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
     array[ vra[i] ] = vrb[i];
  }
}
```

A.7.2 initialisation

scalar-vector set: one

mnemonic	opcode
svso	0x05

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source value
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
scalar	y	y	n	1	val	source value

```
dtype* array = (dtype*) base;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = val;
  }
}
```

scalar-vector set: iota

mnemonic	opcode
svsio	0x06

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source value
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
scalar	y	y	n	1	val	starting value

```
iota = val;
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = iota;
  }
  iota++;
}
```

vector set: clear

mnemonic	opcode
vsc	0x27

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source value
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = 0;
  }
}
```

A.7.3 arithmetic

vector-vector add

mnemonic	opcode
vvadd	0x07

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of sums
vector	y	y	n	1	$\vec{vra}$	vector of first addends
vector	y	y	n	1	$\vec{vrb}$	vector of second addends

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] + vrb[i];
  }
}
```

vector-vector subtract

mnemonic	opcode
vvsub	0x08

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of differences
vector	y	y	n	1	$\vec{vra}$	vector of minuends
vector	y	y	n	1	$\vec{vrb}$	vector of subtrahends

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] - vrb[i];
  }
}
```

vector-vector multiply

mnemonic	opcode
vvmul	0x09

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of products
vector	y	y	n	1	$\vec{vra}$	vector of first factors
vector	y	y	n	1	$\vec{vrb}$	vector of second factors

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] * vrb[i];
  }
}
```

vector-scalar add

mnemonic	opcode
vsadd	0x0A

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of sums
vector	y	y	n	1	$\vec{vra}$	vector of first addends
scalar	y	y	n	1	val	second addend

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] + val;
  }
}
```

vector-scalar subtract

mnemonic	opcode
vssub	0x0B

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of differences
vector	y	y	n	1	$\vec{vra}$	vector of minuends
scalar	y	y	n	1	val	subtrahend

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] - val;
  }
}
```

vector-scalar multiply

mnemonic	opcode
vsmul	0x0C

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of products
vector	y	y	n	1	$\vec{vra}$	vector of first factors
scalar	y	y	n	1	val	second factor

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] * val;
  }
}
```

scalar-vector subtract

mnemonic	opcode
svsub	0x0D

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	vector of differences
vector	y	y	n	1	$\vec{vra}$	vector of subtrahends
scalar	y	y	n	1	val	minuend

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = val - vra[i];
  }
}
```

A.7.4 logical

vector-vector bitwise: or

mnemonic	opcode
vvbor	0x0E

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
vector	y	y	n	1	$\vec{vrb}$	vector of second values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] | vrb[i];
  }
}
```

vector-vector bitwise: and

mnemonic	opcode
vvband	0x0F

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
vector	y	y	n	1	$\vec{vrb}$	vector of second values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] & vrb[i];
  }
}
```

vector-vector bitwise: exclusive or

mnemonic	opcode
vvbxor	0x10

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
vector	y	y	n	1	$\vec{vrb}$	vector of second values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] ^ vrb[i];
  }
}
```

vector-vector bitwise: shift right

mnemonic	opcode
vvbshfr	0x11

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of values
vector	y	y	n	1	$\vec{vrb}$	vector of shift amounts

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] >> vrb[i];
  }
}
```

The shift is arithmetic or logical depending on whether or not the datatype is signed or unsigned.

vector-vector bitwise: shift left

mnemonic	opcode
vvbshfl	0x12

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of values
vector	y	y	n	1	$\vec{vrb}$	vector of shift amounts

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] << vrb[i];
  }
}
```

vector-scalar bitwise: or

mnemonic	opcode
vsbor	0x13

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
scalar	y	y	n	1	val	second value

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] | val;
  }
}
```

vector-scalar bitwise: and

mnemonic	opcode
vsband	0x14

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
scalar	y	y	n	1	val	second value

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] & val;
  }
}
```

vector-scalar bitwise: exclusive or

mnemonic	opcode
vsbxor	0x15

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of first values
scalar	y	y	n	1	val	second value

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] ^ val;
  }
}
```

vector-scalar bitwise: shift right

mnemonic	opcode
vsbshfr	0x16

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of values
scalar	y	y	n	1	val	shift amount

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] >> val;
  }
}
```

The shift is arithmetic or logical depending on whether or not the datatype is signed or unsigned.

vector-scalar bitwise: shift left

mnemonic	opcode
vsbshfl	0x17

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of values
scalar	y	y	n	1	val	shift amount

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = vra[i] << val;
  }
}
```

The shift is arithmetic or logical depending on whether or not the datatype is signed or unsigned.

vector bitwise: not

mnemonic	opcode
vbnot	0x18

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values
mask	n	y	n	1	$\vec{mask}$	optional mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    vrd[i] = ~vra[i];
  }
}
```

A.7.5 comparisons

vector-vector compare: not equal

mnemonic	opcode
vvcne	0x19

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values to compare
mask	n	y	n	1	$\vec{mask}$	optional mask register
mask	y	y	y	1	$\vec{maskd}$	destination mask
vector	y	y	n	1	$\vec{vra}$	vector of first values
vector	y	y	n	1	$\vec{vrb}$	vector of second values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    maskd[i] = (vra[i] != vrb[i]) ? 1 : 0;
  }
}
```

vector compare: not equal to zero

mnemonic	opcode
vcnez	0x1A

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of values to compare
mask	n	y	n	1	$\vec{mask}$	optional mask register
mask	y	y	y	1	$\vec{maskd}$	destination mask
vector	y	y	n	1	$\vec{vra}$	vector of values

```
for (i=0; i<vlen; i++)
{
  if ( (!mask) || (mask && mask[i]) )
  {
    maskd[i] = (vra[i] != 0) ? 1 : 0;
  }
}
```

A.7.6 position manipulation

vector compress

mnemonic	opcode
vcprs	0x1B

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source values
mask	y	y	n	1	$\vec{mask}$	required mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of source values

```
k=0;
for (i=0; i<vlen; i++)
{
   if (mask[i])
   {
      vrd[k++] = vra[i];
   }
}
```

This interesting instruction can be found in the NEC SX, Fujitsu VP, Cyber and Convex machines.

vector expand

mnemonic	opcode
vexpd	0x1C

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
dtype	y	n	n	1	dtype	datatype of source values
mask	y	y	n	1	$\vec{mask}$	required mask register
vector	y	y	y	1	$\vec{vrd}$	destination vector
vector	y	y	n	1	$\vec{vra}$	vector of source values

```
k=0;
for (i=0; i<vlen; i++)
{
  if (mask[i])
  {
    vrd[i] = vra[k++];
  }
}
```

A.7.7 mask

mask-mask bitwise: or

mnemonic	opcode
mmbor	0x1D

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask
mask	y	y	n	1	$\vec{ma}$	first source mask
mask	y	y	n	1	$\vec{mb}$	second source mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = ma[i] | mb[i];
}
```

mask-mask bitwise: and

mnemonic	opcode
mmband	0x1E

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask
mask	y	y	n	1	$\vec{ma}$	first source mask
mask	y	y	n	1	$\vec{mb}$	second source mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = ma[i] & mb[i];
}
```

mask-mask bitwise: exclusive or

mnemonic	opcode
mmbxor	0x1F

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask
mask	y	y	n	1	$\vec{ma}$	first source mask
mask	y	y	n	1	$\vec{mb}$	second source mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = ma[i] ^ mb[i];
}
```

mask bitwise: not

mnemonic	opcode
mbn	0x20

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask
mask	y	y	n	1	$\vec{ma}$	source mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = ~ma[i];
}
```

mask set all

mnemonic	opcode
msa	0x21

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = 1;
}
```

mask clear all

mnemonic	opcode
mca	0x22

type	required	src	dst	nybles	symbol	comment
mask	y	n	y	1	$\vec{md}$	destination mask

```
for (i=0; i<MAXVL; i++)
{
  md[i] = 0;
}
```

mask population

mnemonic	opcode
mpop	0x23

type	required	src	dst	nybles	symbol	comment
vlen	y	y	n	0	vlen	vector length (implicit argument)
mask	y	y	n	1	$\vec{mask}$	source mask
scalar	y	n	y	1	srd	destination scalar

```
srd=0;
for (i=0; i<vlen; i++)
{
   if (mask[i]) srd++
}
```

Assume it zero extends the scalar value to 64 bits, that way we need not specify a datatype.

A.7.8 vector length

set vector length

mnemonic	opcode
svl	0x24

type	required	src	dst	nybles	symbol	comment
scalar	y	y	n	1	vra	source value
vl	y	n	y	0	vlen	implicit destination

```
vlen = sra;
```

get vector length

mnemonic	opcode
gvl	0x25

type	required	src	dst	nybles	symbol	comment
vl	y	y	n	0	vlen	implicit source
scalar	y	n	y	1	srd	destination scalar

```
srd = vlen;
```

set maximum vector length

mnemonic	opcode
smvl	0x26

type	required	src	dst	nybles	symbol	comment
vl	y	n	y	0	vlen	implicit destination

```
vlen = MAXVL;
```

A.7.9 memory fences

These instructions are relevant to out of order execution and have no semantic meaning by themselves.

fence: scalar store to vector memory operation

mnemonic	opcode
fssvm	0x28

Commits all prior scalar stores before any subsequent vector memory operation can issue.

fence: vector store to scalar memory operation

mnemonic	opcode
fvssm	0x29

Commits all prior vector stores before any subsequent scalar memory operation can issue.

fence: vector store to vector memory operation

mnemonic	opcode
fvsvm	0x2A

Commits all prior vector stores before any subsequent vector memory operation can issue.