Advanced Architectures: Report document with a description of the proposed vector extensions

D4.1 v1.0

WP4 – Advanced Architectures: D4.1 Report document with a description of the proposed vector extensions

Dissemination Level: Confidential

Lead Editor: Adrián Cristal

Date: 01/11/2013

Status: Final

Description of Work:

In particular, the work in WP4 will consist of the following:

(1) Introducing novel vector instructions for scalable execution of database queries.

We will investigate how adding new vector instructions to the existing processors can help to exploit data-level parallelism, and how the corresponding processor hardware can be

designed. The majority of the effort focuses on determining how to divide the work between the hardware and the runtime system to dynamically exploit the new support from hardware, and how to fine-tune the system to provide a satisfactory power/performance ratio.

T4.1 Characterize the DBMS execution and describe the potential hardware extensions
Evaluate the performance of DBMSs and discover the parts of the engine and the types of workload can particularly benefit from having specific hardware extensions. We will extract the kernels which represent the computational hotspots in the DBMSs. Describe the requirements of proposed hardware extensions, and the interaction between the hardware extensions and the current and future DBMS use-cases.

T4.2 Acceleration with novel vector operations

We will investigate extending processors with novel vector hardware operations that aim in accelerating the execution of column-store databases in a power-efficient way. We will use the DBMS kernels identified in T4.1. We will then investigate how traditional vector instructions suit these selected kernels at exploiting this parallelism. Next, we will define new operations that capture the available parallelism concisely where previous vector instructions are unable to do so efficiently. From these new instructions, we will propose a corresponding hardware design which we will then implement using a detailed, cycle-accurate simulator. We will then modify the source code of the profiled software to vectorize it with our new instructions and then evaluate the performance impact. We will investigate the potential power and energy savings when using this new hardware.

Activity Description

Initial simulator development M4-M8 (2PM BSC)

Detailed hardware design M9-M12 (2PM BSC)

Contributors:

Adrián Cristal
Nehir Sonmez
Adrià Armejach
Timothy Hayes

Internal Reviewer(s):

2ndQ: Martín Marqués
PV: Olivier Marchesini

Version History

Version	Date	Authors	Sections Affected
0.1	21/10/2013	AC, AA, NS, TH	Initial version
1.0	01/11/2013		Final version (inc. feedback)

Table of Contents

List of Figures

1. Summary

This document describes the work being done for vector extensions for PostgreSQL. It details simulating the addition of new vector instructions and hardware to the x86-64 ISA on PTLsim. Section 2.1 gives an introduction to vector architectures, 2.2 briefly presents the TPC-H benchmark, and Section 2.3 gives detailed information about the PTLsim simulator. Section 3 describes the extensions made to the simulator in order to support a rich set of vector extensions and Section 4 includes the references.

2. Analysis/ Description of work done

2.1 Introduction

Modern microprocessors often include SIMD multimedia extensions that can be used to exploit data-level parallelism. There has been some prior work using these extensions to accelerate database software, which 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.

2.1.1 Vector Architectures

Vector architectures [ICS98] 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 and can be implemented on-chip using simple and efficient hardware. 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 [VLDB99]. Vector architectures have traditionally been used for scientific applications with floating-point code, and however a lot of the concepts could as well be applied to business-domain (integer) applications.

Vector architectures [ICS98] 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. Vector hardware is typically much simpler 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 for being more energy efficient 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.

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. Additionally, there is less requirement for aggressive speculation and memory prefetching since execution and memory access patterns are fully encoded into a single instruction.

This work takes a top-down methodology and profiles PostgreSQL looking for opportunities to use vector technology. From this, software bottlenecks caused by unscalable scalar hardware structures are identified and new vector ISA extensions for x86-64 are proposed as a solution. These ISA extensions are currently being implemented using a cycle-accurate microprocessor simulator.

2.2 Benchmarking: TPC-H

The TPC Benchmark H (TPC-H) that was selected by all partners of the AXLE consortium for evaluation is a decision support benchmark [TPCH]. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

The experiments were conducted on the following machine:

- Processors: 2 x Intel Xeon E5-2670, Sandy Bridge DP, 2.6Ghz 8 Core/16 Threads, 3.3Ghz Turbo, 8GT/s QPI, 20MB L3 cache
- RAM: 16 x Samsung 16GB DDR-3 at 1600Mhz
- Serverboard: 1 x Supermicro X9DRi-F, 2 x Intel Gbit LAN, IPMI KVM-over-IP, 8 x Intel SATA2 software RAID
- Network: 1 x Supermicro AOC-STG-I2 Network Card, 2 x 10Gbit/sec CX4
- Disk: 2 x Intel® DC S3700 Series, 400GB , 2.5in SATA 6Gb/s, MLC, Read 500MBs/75k iops, Write 460MBs/36k iops.

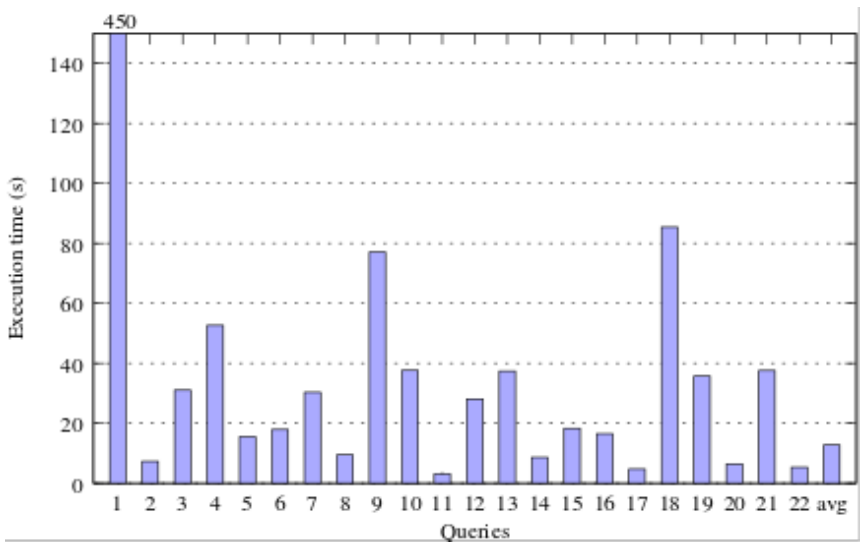

Figure 1 : TPC-H runtimes

We ran this benchmark with a 10 GB DB size. The tests were ran on a ramdisk in order to rule out the i/o overheads and to only look at the CPU-intensive operations.

Figure 1 shows the CPU time of 22 queries executed on a database of 10 GB on our Intel Xeon system with 256 GB of DDR3-1600 memory.

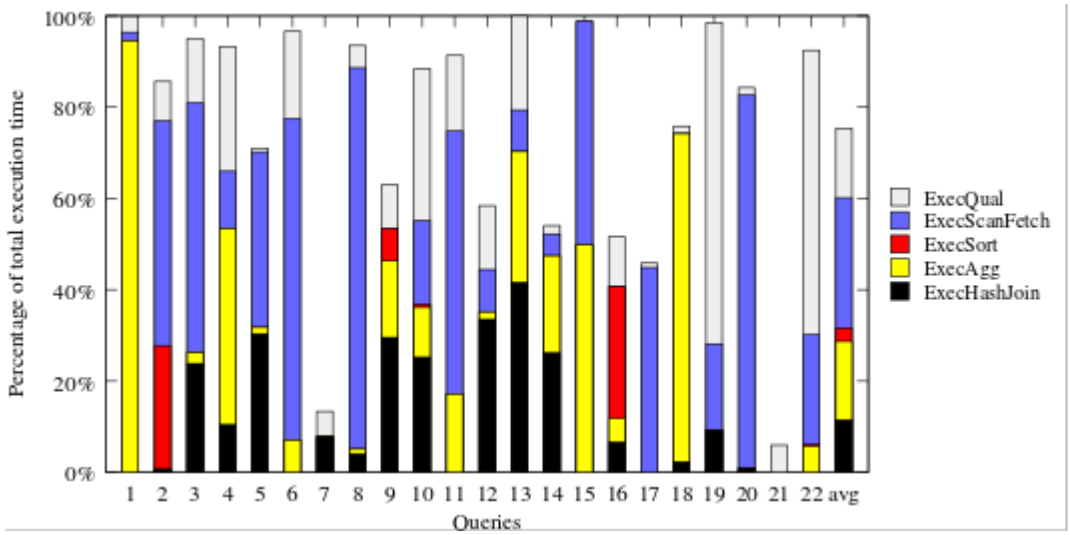

Figure 2: TPC-H breakdown

Figure 2 shows a breakdown of the most significant PostgreSQL execution kernels. ExecScanFetch, ExecAgg, ExecQual and Hash Join can be seen as some of the most time consuming kernels. For this work we focused on the hash join operation. Although this work focuses on a particular algorithm, we expect many of our findings to be applicable to other aspects of the DBMS. The vector extensions proposed here are as generic as possible to enable this.

2.2.1 An Example: Hash Joins

The hash join operator requires two input sets called the outer and inner tables. It does not require either of the two sets to be ordered: the inner table is always a hash table and the ordering of the outer table is not important.

First the inner table is created using the Hash operator, which creates a temporary hash index that covers the join column of the inner table.

Then, hash join reads each row in the outer table, hashes the join column from the outer table and searches the temporary hash index for a matching value.

```
EXPLAIN
SELECT * FROM customers, sales
        WHERE sales.customer_id = customers.customer_id;
Hash Join
        -> Seq Scan on customers
        -> Hash
                -> Seq Scan on sales
```

Figure 3: An example query plan that uses the Hash Join operator

2.3 The simulator

2.3.1.1 Base ISA

x86-64 was chosen as the base ISA to extend for several reasons. It is the leading ISA in the server market having roughly 60% of the market share. It is a very universal ISA with mature optimising compilers and toolchains. x86-64 is also a large improvement over the archaic 686 ISA and many improvements have been made - e.g. the number of general purpose registers is doubled, and some legacy features such as memory segmenting have been removed.

Additionally, using x86-64 also allows for any simulated experiments to be compared against modern commodity hardware. 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. Although different x86-64 incarnations contain varying support of different SSE version, x86-64 has SSE and SSE2 integrated into its core ISA.

2.3.1.2 Memory or Registers

A major design decision was whether to have a memory-memory based ISA like the CDC-STAR 100 [COMPCON72] or a vector-based ISA like the Cray 1 [ACM78]. 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 become 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. For all these reasons, a register-based ISA was chosen for this design.

2.3.1.3 Registers

The registers used for the ISA have been chosen based on the requirements of explored functions. 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 and the number of iterations needed can be decided dynamically.

To make the hardware simpler, there is no sub-word 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 sub-word parallelism is allowed, it means that an indexed memory instruction could operate on x qword elements, 2x dword 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 sub-word 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 [ISCA02], 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 relevant kernels 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.

2.3.1.4 Instructions

In general, the new ISA has a lot more flexibility than the SSE versions. Currently it only works with integer datatypes, 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
- optional vector mask (for the majority of instructions)

2.3.1.5 Out of Order Execution

One of the biggest design decisions made was to allow vector instructions to issue out of order. [MICRO97] 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 are very power-hungry. 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.

2.3.1.6 Cache integration

[ICS99] proposes a solution to integrate 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 is 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 [ISCA02] 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.

2.3.1.7 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 results, there is a dependency that must be honoured. Adding logic to detect these 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 purposes. 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 is already included in the x86-64 ISA.

2.3.1.8 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.

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

Figure 4: Cache Hierarchy Parameters

The cache hierarchy should be inclusive. The L1 data cache is expected to use a writethrough 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 practice storing a vector to a location already present in the L1 data cache seldom happens.

2.3.1.9 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, except for the memory instructions.

Vector loads pose the most problematic situations. Because of the use of 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 → 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 → vector register
- vector register → mask register
- mask register → vector register
- vector register → store

2.3.1.10 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.

2.3.2.1 PTLsim

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 86-64 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

2.3.2.2 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.

2.3.2.2.1 In order pipelined 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
- 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.

2.3.2.2.2 Out of Order Engine

The out of order engine is implemented using a clustering mechanism. 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. A single issue queue and some functional units are placed into exactly one cluster. Every cycle, the oldest µop (micro-operation) 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.

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 mis-speculation: replaying the instruction, redispatching the instruction and annulling the instruction.

parameter	value
fetch width	4
fetch queue	28
frontend width	4
frontend stages	7
dispatch width	4
writeback width	4
commit width	4
issue width per cluster/total	1/6
reorder buffer	128
issue queue	8 (per cluster)
load queue	48
store queue	32
outstanding l1d misses	10
outstanding l2 misses	16

Figure 5: Superscalar and out of order parameters

2.3.2.2.3 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.

2.3.2.2.4 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 ⊂ L2 ⊂ L3 ⊂ 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.

parameter	value
type	DDR3-1333
clock	1.5 ns
transaction queue	64
command queue	256
policy	open page
row accesses	8
data bus	64 bits (JEDEC standard)
queue	per rank per bank
scheme	row:rank:bank:chan:col:burst
scheduling policy	rank then bank
banks	8
ranks	4
rows	32768
columns	2048
device width	4
burst length	64 bytes

Figure 6: Memory System Parameters

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) 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 bring the requested data from the closest cache or from 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.

3.1 Proposed 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 [ISCA02] 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.

3.2 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 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.

parameter	value
maximum vector length (MVL)	64
physical vector registers	16
physical mask registers	8
vector load requests	12
vector store requests	8
L2 → vector register bus width	32 bytes
vector cache line requests	256
maximum datatype width	64 bits

Figure 7: Vector parameters

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 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.

class	instructions
memory	unit stride, indexed, prefetching
arithmetic	integer mul, add, sub
bitwise logical	and, or, not, xor, shift
comparison	not equal
initialisation	set all, clear all, iota
mask	set, clear, and, or, not, xor, popcount
permutative	compress
vector length	set, set MVL, get
memory fence	scalar-vector, vector-scalar, vector-vector

Figure 8: Vector instruction listing

3.3 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 useful 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:

• Vector 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.

3.4 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 9: Vector Load Request file

Figure 9 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.

3.5 Vector Store Request file

The VSRF is very similar to the VLRF. Figure 10 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.

F	register	vclrf_tosend	vclrf_sent

Figure 10: Vector Store Request file

3.6 Vector Cache Line Request File (VCLRF)

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.

F	L	R	idx	address[48:5]	bytes_needed[63:0]	reg_el

Figure 11: Vector Cache Line Request File

Figure 11 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 with the number of bits equal to the size in bytes of the 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.

3.7 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 the asynchronous wakeup mechanism, loads don't actually need this entry for normal execution however both loads and stores need to have this information when mis-speculation recovery occurs. In these cases, the entry in the VLRF or VSRF and all its corresponding VCLRF entries must be annulled and recycled.

3.8 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 buses 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.

3.9 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, there were few constraints needed on the ordering of memory operations and so these were allowed to be executed out of order as well. 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 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 committed. 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.

On our architecture, there are three types of fence instructions. Scalar store → vector memory; vector store → scalar memory; and vector store → 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 12 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 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.

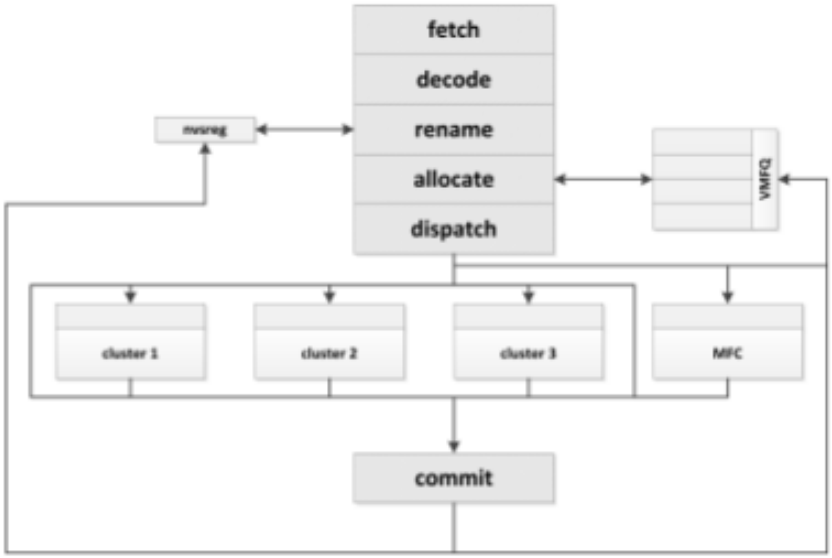

Figure 12: Vector memory fence implementation

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 in 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 mis-speculated 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.

3.10 Sorting Additions to the ISA

In addition to the instructions for vectorizing hash joins, we are introducing vector reverse, select and shuffle instructions. These allow us to implement and experiment with various sorting network algorithms. The shuffle instruction is of particular importance as its implementation may lead to an all-to-all multiplexor. This is a particular topic that we are interested in and will actively look for better implementations and/or alternatives (in year 2 of the project) to a generic shuffle instruction to have a fast sorting network without complex hardware. Also, strided memory access instructions which are important for implementing radix sort, are being implemented, as well.

4. References

[ICS98] 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.

[VLDB99] 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.

[TPCH] http://www.tpc.org/tpch/

[COMPCON72] R. G. Hintz and D. P. Tate. Control data star-100 processor design. In IEEE Computer Society Conf., COMPCON 72, pages 1–4, 1972.

[ACM78] Richard M. Russell. The cray-1 computer system. Commun. ACM, 21:63–72, January 1978.

[MICRO97] 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.

[ISCA02] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and Andre 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.

[ICS99] 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.