University of Manchester
School of Computer Science
Project Report 2016

SpiDB
Databases on SpiNNaker

Author: Arthur Ceccotti

Supervisor: Dr. David Lester

Abstract

SpiDB
Databases on SpiNNaker

Author: Arthur Ceccotti

A Database Management System (DBMS) is a suite of programs which manage large structured sets of persistent data.[1] Thus it performs the organization, storage, security and integrity of the user's data.
This report contains details of an approach for implementing a DBMS, namely SpiDB, on the Spiking Neural Network Architecture (SpiNNaker), an emerging, highly distributed hardware design optimised for Neural Network simulations. The open-source project covers different implementations of a Key-value store and a Relational DBMS under the architecture constraints. As a research project, it has a strong focus on the evaluation of results, which allows exploration of how the SpiNNaker hardware performs under a non-neuromorphic environment. The conclusions gathered show limitations and strengths of the architecture under this general purpose application, with suggestions on possible modifications. This can serve as feedback for improvements on the ongoing SpiNNaker development.

Supervisor: Dr. David Lester

Acknowledgements

I would firstly like to thank my supervisor, David Lester and Ning Zhang for the useful feedback and motivation during the development of my project. Massive gratitude goes also to Alan Stokes, who became a second supervisor to me, always raising interesting insights and introducing me to the SpiNNaker lifestyle, alongside Andrew Rowley.

As always, friends and family will always be in my heart. I love you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes a high level view of the SpiNNaker reasearch and its main uses. It highlights also the motivation and aims of my project and how it may impact on the improvement of a large scale international research.

1.1 Overview

"SpiNNaker is a biologically inspired, massively parallel computing engine designed to facilitate the modelling and simulation of large-scale spiking neural networks of up to a billion neurons and trillion synapses (inter-neuron connections) in biological real time." [2] The SpiNNaker project, inspired by the fundamental structure and function of the human brain, began in 2005 and it is a collaboration between several universities and industrial partners: University of Manchester, University of Southampton, University of Cambridge, University of Sheffield, ARM Ltd, Silistix Ltd, Thales. [3] A single SpiNNaker board is composed of hundreds of processing cores, allowing it to efficiently compute the interaction between populations of neurons, partially simulating a human brain.

1.2 Project Aim

This research project involves making use of the SpiNNaker software stack and hardware infrastructure, both optimised for neural network simulations, in order to explore and evaluate its usability and performance as a general purpose platform. This has been achieved through the development of a distributed Key-Value store and a Relational Database Management System with limited scope. This project has grown to be the largest non-neuromorphic application now available as part of the SpiNNaker API.

SpiNNaker is particularly strong at parallel execution at low power consumption, which appealed as an extraordinary opportunity to store and retrieve data in a fast, distributed way, under a database management system. This allows exploration of a broad range of ideas outside of the initial scope of SpiNNaker, testing some of its capabilities and limitations against a non-neuromorphic environment.

In addition to usability testing, an important objective is to gather performance benchmarks for this application, allowing analysis which can provide insights for improvements to the current architecture, possibly influencing on changes to reflect on the next generation of the chip:

SpiNNaker 2. This data can also be used to further enhance my application in the future, as it is owned by the SpiNNaker team itself.

Chapter 2

Background

This chapter describes an architectural overview of the SpiNNaker research, hardware specifications and multicore communication used on my database project.

2.1 SpiNNaker Architecture

The basic building block of the SpiNNaker machine is the SpiNNaker Chip Multiprocessor (CMP), a custom designed globally asynchronous locally synchronous (GALS) system with 18 ARM968E-S processor nodes (figure 2.1). [2] The SpiNNaker chip contains two silicon dies: the SpiNNaker die itself and a 128-MByte SDRAM (Synchronous Dynamic Random Access Memory) die, which is physically mounted on top of the SpiNNaker die and stitch-bonded to it (figure 2.2). [4] The SDRAM serves as local shared memory for the 18 cores within the chip, also utilized for memory based communication. [5]

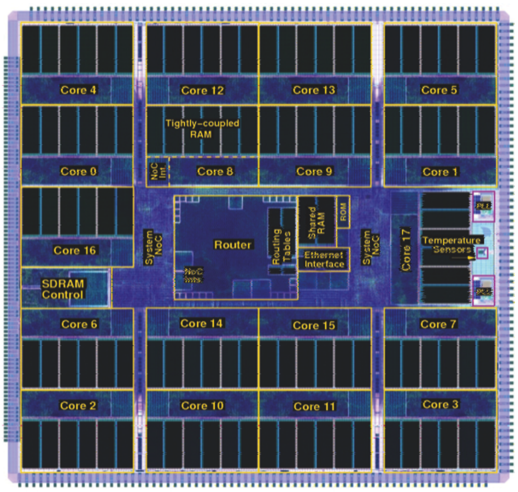

Fig. 2.1: SpiNNaker chip layout

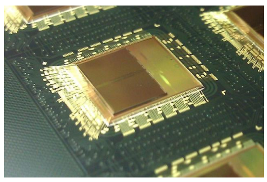

Fig. 2.2: SpiNNaker CMP and SDRAM

Each ARM core within the chip follows a 32-bit Harvard Architecture, holding a private

Fig. 2.3: SpiNN-3

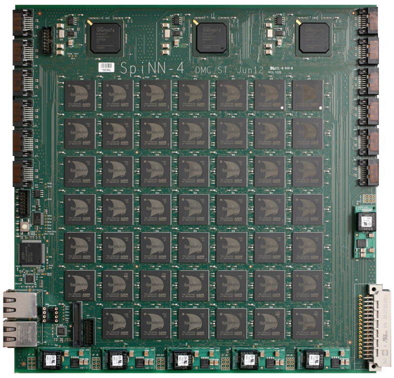

Fig. 2.4: SpiNN-4

32-KB instruction tightly coupled memory (ITCM) and 64-KB data tightly coupled memory (DTCM).[2] It has a small peak power consumption of 1-W at the nominal clock frequency of 180-MHz.[6]

There are currently two types of SpiNNaker boards: **SpiNN-3** (figure 2.3), composed of 4 chips, thus a total of 72 processing cores, and **SpiNN-4** (figure 2.4), composed of 48 chips for a total of 864 processing cores.

2.2 SpiNNaker Communication fabric

Cores on a SpiNNaker board exchange packets through wired connections over a large communication fabric. Each SpiNNaker chip is surrounded by a lightweight, packet-switched asynchronous communication infrastructure.[4] Packet exchange can be used to transmit information initially private to a core and it is managed under the API's even driven architecture, thus incomming packets issue an interrupt on the receiving core.

There are currently 4 different communication protocols in the system. It is worth noting that **none of these protocols guarantee successful delivery of data** and this effect is worsen if there is increased traffic in the communication fabric. Sending a large amount of packets symultaneously is likely to result on packet drops.

- **Multicast (MC)**: The *MC* protocol, originally designed to simulate neural spikes, is used when a sigle source issues information to multiple destinations (one-to-many). The packet contains a 32 bit routing key, used by an external router process to carry out the delivery, and an optional 32 bit payload, both provided at the source.

- **Point to Point (P2P)**: *P2P* packets have a single source and a single destination core (one-to-one). Each packet contains a 16-bit source ID, destinarion ID and an optional 32-bit payload.[5] On top of this layer, the SpiNNaker Datagram Protocol (*SDP*) was designed to allow transfers of up to 256-bytes of data between two cores, by sending a sequence of *P2P* packets with payloads.[7] *SDP* can be used to communicate to the *host* machine, wrapped around a larger *UDP* packet.

- **Nearest-neighbour (NN)**: Under the *NN* protocol, packets issued at a chip are delivered to the Monitor Processors of each of its 6 neighbours (one-to-many) according to the board's connections. *NN* packets contain a 32-bit payload and a 32-bit address/operation field.[5]

- **Fixed-route (FR)**: *FR* packets use the same mechanism as *MC*, without a key field. Packets being routed by the contents of a register instead of a routing key.

My database application makes extensive use of the *SDP* protocol, alongside *MC*.

Chapter 3

Development

This chapter describes the development of my database application, SpiDB, from September 2015 to March 2016, involving planning, testing and implemention. The project is open-source, which can be found at *https://github.com/SpiNNakerManchester/SpiNNakerGraphFrontEnd*.

By being part of the SpiNNaker team in Manchester, I was directly exposed to the ongoing research, frequently receiving feedback on my work. The collaboration was effectively bi-directional, as I was able to constantly find, evaluate and fix inconsistencies and bugs in the API not known to the team.

3.1 Planning

The first step of my project involved making a detailed plan of approach, taking into consideration the time resources and learning curve. Such plan was useful to set myself deadlines for deliverables in the cycle of short iterations and keep track of progress. A high level chart with the weekly delivery plan can be seen on figure 3.1.

The overall project was divided into two halves: developing a No-SQL Key-value store for insertion and retrival of non-structured data and an SQL based Relational Database for creation and manipulation of table structures.

3.1.1 Requirements Analysis

The project plan involved analysing the importance of different requirements based on their relative difficulty and scope within the project aim. Modern database management systems have a broad range of complex requirements. Given the limited resources, I have selected 4 important concepts which must have strict focus on the SpiDB system:

- **Reliability**: user's queries must complete in a reasonable way. This means any internal errors, inconsistencies or communication failures should be handled from within the database system, avoiding unpredicted failures to the user. This is a difficult task given the unreliability of the SpiNNaker communication fabric, earlier discussed on section 2.2.

- **Durability**: user's queries with the aim of modifying the database state should persist, being internally stored until removal. The SpiNNaker hardware does not contain permanent storage components, reducing this contraint to ”insert operations must persist until

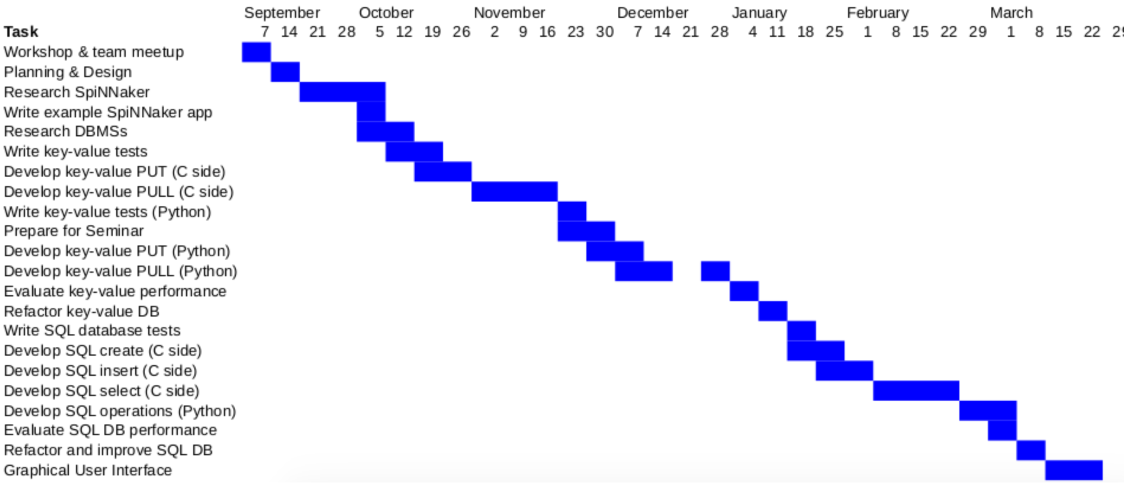

Fig. 3.1: Development Plan

shutdown". It would be possible to externally connect the board to a mass-storage device, but it is outside of the scope of this project.

- **Isolation**: executing transactions concurrently must have the same effect as doing so sequentially. Concurrency control is an important part of this parallel database system, as it aims to handle thousands of concurrent queries distributed among thousands of processing cores. An approach to ensure isolation in the system is further discussed in section 3.5.1.

- **Scalability**: the system must be able to handle a large number of parallel queries and have enhanced performance when given more hardware resources, in this case processor count. This is arguably the most important of all requirements, as the SpiNNaker team is currently focusing on building a large scale machine composed of 1,000,000 processing cores. If this application scales well, it will quickly be able to show the strengths and weaknesses of the machine.

These main requirements do not cover two of the four ACID (Atomicity, Consistency, Durability, Isolation) properties of a database: atomicity and consistency. Atomicity, althogh very important on a large commercial application, is extremely hard to achieve in a descentralized system, with the use of complex multi-core rollbacks, and falls out of the scope of this experimental project. Consistency is significant when ensuring data written to the database is valid according to constraints, triggers and cascades, which are originally non-existent in a reduced instruction data store and do not contribute to the project research.

Lastly another outstanding requirement not included in the plan was a strong security protocol. Data encryption and authorization have many advantages, but present themselves as unecessary complexity for a small, private, experimental project.

3.1.2 Technologies

Part of the project plan involves research on the SpiNNaker hardware and API, extensively used on my project. Given the steep learning curve of a low level distributed system, on the 7^{th} of September 2015, I attended the *5th SpiNNaker Workshop*, where tens of researchers from around the globe gathered for a one week course on the SpiNNaker Project in Manchester. I was officially introduced to the team, whom I would learn from and work with for the rest of the year.

The SpiNNaker API is split two: the Python toolchain, running on the *host* machine (user's computer), and C code, compiled to run on the board. The full API can be found at *https://github.com/SpiNNakerManchester*. A strong knowledge of ARM assembly was also needed, given the low level architecture.

These technologies were used to develop the following deliverables:

- **Python**: (2000 lines of code) uploading binaries to the board, status checking, query parsing, Graphical User Interface, data analytics and socket communication.

- **C**: (2500 lines of code) event driven communication between processors, low level memory management, distributed query processing.

3.2 Design

Internally, I designed SpiDB to have a hierarchical tree structure architecture, composed of *root*, *branch* and *leaf* nodes/cores. This structure allows a divide-and-conquer approach to the database query plan. When queries are issued by the *host* machine, they are received at the *root* core.

Each chip contains a single *root* node, which handles incomming packets from *host* by redirecting them to *branch* and *leaf* nodes, in an intelligent way, for parallel processing. The middle layer is composed of 4 *branch* cores in charge of aggregating data returned by *leaf* nodes, serving also as "capacitors", slowing down excessive queries which may overload a destination core. Finally the chip is composed of 12 *leaf* nodes, with the aim of actually storing and retrieving database entries (key-value pairs or table rows) on shared memory. These roles can be visualised on figures 3.2 and 3.3. The 2 remaining cores are used for internal use of the system (*sark* and *reinjector*), omitted from these figures.

There are two important advantages, in favour of scalability, which lead me to make this design choice. Firstly, SpiNNaker communication is unreliable, meaning that a lot of traffic and a central point of failure cause large packet drop rate. This hierarchical structure strongly reduces the problem, and it assures cores will never receive packets from more than 4 other cores, distributing queries in an efficient way and protecting cores from excessive incomming packets. Secondly this approach is inheritably beneficial for merges and aggregation (eg. sql keywords COUNT, MAX, MIN, AVG, SUM, etc.), as these can be done on different layers over iterations.

A disadvantage of this design is that less cores perform the actual storage and retrival of data. Out of 16 application cores, 4 are used only for distribution of the query plan (*root* and *branches*). This consequently impacts performance, as each *leaf* node will be assigned more memory to read and will be kept busier with query processing.

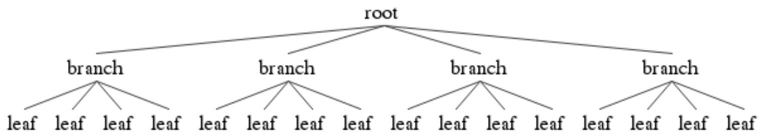

Fig. 3.2: SpiDB tree structure

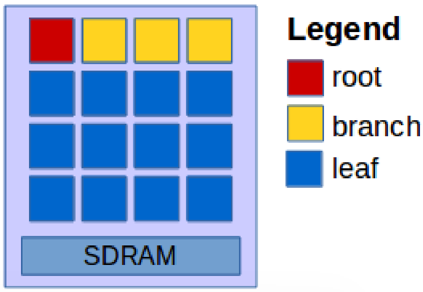

Fig. 3.3: Role assignments per chip

3.3 Implementation

3.3.1 Key-value Store

The first half of the project involved developing a distributed key-value store, in which users can insert and retrieve data entries in a dictionary form. This means the user must be able to set keys mapping to values, both of type *int* or *string*, and retrieve such values when given the same keys. This section describes in detail the internal processing of the insert (*put*) and retrieve (*pull*) queries.

3.3.1.1 PUT

The *put* operation is the main way of inserting data onto the SpiDB distributed database. Upon completion, such operation will store the specified key, mapping to its value, on the memory of an arbitrary chip in the Spinnaker board (as chosen by the *root* core). This operation expects an acknowledgement from the core which stored the entry.

Example usage:

```
put "hello" "world"
put 123 "foo"
put "life" 42
put 1 2
```

Internally the following steps occur:

1. User issues query of format *put "key" "value"* on host machine.

2. Query and metadata are converted into a byte array with the following format, transferred via UDP over Ethernet to the board (figure 3.4).

```
typedef struct putQuery{
  spiDBcommand     cmd;
  uint32_t         id;

  uint32_t         info;
  uchar            k_v[256];
} putQuery;
```

   In this case *SpiDBCommands* is set to the constant representing the put operation, *id* identifies every query uniquely, *info* contains bit encoding of the size and type of both key and value, *k_v* contains key and value appended.

3. Packet arrival triggers interrupt on *root* core, via the SDP protocol, and is decoded.

4. *root* selects a *leaf* core to store the data entries in one of the following ways, specified by the user:

   - **Naive**: as packets arrive, they are assigned to a different *leaf* node in a Round-Robin manner.
   - **Hashing**: the given key is used to produce a 32-bit hash value, which is decoded to assign the query to a specific *leaf* node.

5. *root* communicates to chosen *leaf* node with *put* contents via SDP (figure 3.5).

6. *leaf* node triggers an interrupt and stores key-value entry into its dedicated region of SDRAM.

7. *leaf* sends an acknowledgement message over UDP back to host (figure 3.6).

8. User receives timing information and query result.

Complexity: linear to the size of the input key-value, constant to database size.

3.3.1.2 PULL

The *pull* operation is the main way of retrieving data from the SpiDB distributed database. Upon completion, such operation will return the value mapped by a given key, from an arbitrary chip in the SpiNNaker board, or not respond if such key was not found. This operation expects a response only if the key is present on the database, thus it is an undecidable problem. The reason for this is further explained in section 3.5.2.

Example usage:

```
pull "hello"
pull 123
```

Internally the following steps occur:

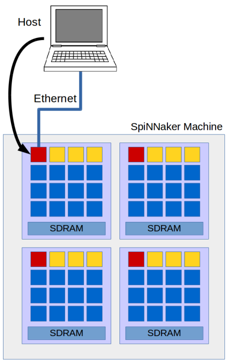

Fig. 3.4: Host to root packet

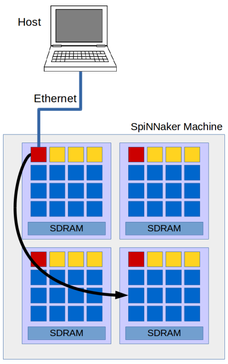

Fig. 3.5: Root to leaf packet

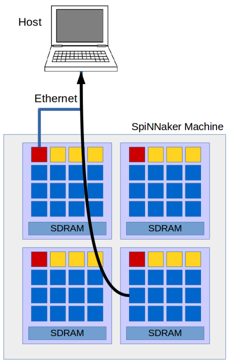

Fig. 3.6: Leaf to host packet

1. User issues query of format *pull "key"* on host machine.

2. Query and metadata are converted into a byte array with the following format, transferred via UDP over Ethernet to the board

```
typedef struct pullQuery{
  spiDBcommand      cmd;
  uint32_t          id;

  uint32_t          info;
  uchar             k[256];
} pullQuery;
```

   Where *SpiDBCommands* represents the pull operation constant, *id* is the query, *info* contains bit encoding of the size and type of the given key, *k* contains the key itself encoded as a byte-array.

3. Packet triggers interrupt on *root* core, via SDP, and is decoded.

4. *root* selects one of the following search strategies, based on database type:

   - **Naive**: there is no knowledge of which, if any, chip contains the given key. Therefore *root* issues a multicast packet to all *leaf* nodes on the board, requesting them to linearly scan their regions of shared memory, searching for the entry.
   - **Hashing**: the key is used to produce a 32-bit hash value, which, if existent on the database, must be present at the memory of a specific core, pointed by the decoding of such hash value. Therefore *root* node sends a single SDP packet to chosen *leaf*, requesting it to search for the given key.

5. Each *leaf* node that received a search request triggers an interrupt and searches for key-value mapping in SDRAM memory. If key is found, value is returned over UDP back to host.

6. User receives timing information and query result.

Complexity: linear to the size of the input, linear to database size.

Algorithms 1 and 2 show a high level simplification of code running in the *root* and *leaf* nodes, as described above, through the event driven application, implementing *put* and *pull* operations.

Algorithm 1 Root core

```
procedure onReceive(sdp)
    if sdp.command is PUT then
        if DB_TYPE is NAIVE then
            forward PUT query to next core (Round-robin)
        else if DB_TYPE is HASH then
            h ← hash(sdp.key)
            chipx ← h[0 : 7]
            chipy ← h[8 : 15]
            core ← h[16 : 24]
            forward sdp PUT query to (chipx, chipy, core)
    if sdp.command is PULL then
        if DB_TYPE is NAIVE then
            issue multicast PULL to all cores in the system
        else if DB_TYPE is HASH then
            h ← hash(sdp.key)
            chipx ← h[0 : 7]
            chipy ← h[8 : 15]
            core ← h[16 : 24]
            forward sdp PULL query to (chipx, chipy, core)
```

Algorithm 2 Leaf core

```
procedure onReceive(sdp)
    if sdp.command is PUT then
        store sdp.key and sdp.value in SDRAM
    if sdp.command is PULL then
        entry ← SDRAM[0]
        i ← 0
        while entry is not null do
            if entry.key is sdp.key then
                send response to host with sdp.value
                return
            entry ← SDRAM[i]
            i ← i + 1
```

3.3.2 Relational Database

The second half of the project involves developing an SQL-based distributed database on top of the key-value store layer. Entries are stored in a structured way, bounded by the definition of tables. In this RDMS users can create tables with different fields, insert values and retrieve them with given conditions. This section describes in detail the internal processing of the *create*, *insert* and *select* queries.

3.3.2.1 CREATE

In SpiDB, the *create* operation is used to generate a table definition in the SpiNNaker hardware. Data can only be inserted into the database if a corresponding table exists. This query has as parameters the table name, field names and their types (*int* or *string*). Multiple distinct tables can be issued symultaneously, although they are handled by a single core. This operation expects an acknowledgement from the *root* core, which sets the failure flag if the table definition already exists.

Example usage:

```
CREATE TABLE Dogs(name varchar(10), owner varchar(35), age integer);
CREATE TABLE People(name varchar(35), lastname varchar(20));
```

Internally the following steps occur:

1. User issues query of format *CREATE TABLE name(column1 type(size), ...)* on host machine.
2. Query and metadata are converted into a byte array and sent to the board (format can be found on appendix section A.2).
3. *root* core receives and decodes SDP packet.
4. If table does not exist on the database yet, *root* core stores table definition and metadata in its region of shared SDRAM, accessible by other cores for insert/retrieve operations.
5. *root* core sends acknowledgement back and information is displayed to the user.

Complexity: linear to the size of the input, constant to database size.

3.3.2.2 INSERT

New values can be added to the database management system through the *INSERT* query. A single entry is an *int* or *string* value assigned to a column on a given table at a new row. Multiple entries can be safely inserted concurrently, as they are distributed across different cores. This operation expects an acknowledgement from the *leaf* node in charge of storing the value and metadata, with the failure flag set if memory is full.

Example usage:

```
INSERT INTO Dogs(name,owner,age) VALUES ("Toddy","Arthur",8);
INSERT INTO Dogs(name,owner,age) VALUES ("Guto","Arthur",10);
INSERT INTO People(name,lastname) VALUES ("Arthur","Ceccotti");
INSERT INTO People(name,lastname) VALUES ("Canny","Dattlin");
```

Internally the following steps occur:

1. User issues query of format *INSERT INTO table(column1, ...) VALUES (value1, ...)* on host machine.

2. Query is broken down into column-value pairs, containing also value type, size and specified table (eg. [”name”:(”Arthur”, type: string, size: 6)]). This step is necessary because SDP packets have a limit size of 256-bytes, thus not being able to carry one single large packet containing all the assignments for a table row. Streaming smaller packets also decreases the need for a large storage buffer at the destination.[7]

3. Column-value pairs (entries) are converted into a byte array, sent to the board, being received and decoded by the *root* node.

4. The column-value pair query is forwarded to a *leaf* node in a Round-Robin way.

5. *leaf* node receives query, checks if specified table exists in shared memory SDRAM and reads its definition.

6. *leaf* node stores column-value pair into a new row in reserved region for given table. The SpiDB RDMS is row-based, thus entire rows are stored consecutively in a structured way at all cores.

7. *leaf* node acknowledges query, returned to the user.

Complexity: linear to the size of the input, constant to database size.

3.3.2.3 SELECT

In the SpiDB system, multiple entries from a table can be retrieved using the *SELECT* query, following the standard SQL syntax. Selection criteria can be specified by the user and matching results are streamed until a timeout is reached. If no value in the table matches such criteria, there will be no response from the board, which is a consequence of not making use of internal acknowledgements, as discussed in section 3.5.2. This is a similar behaviour to *pull* requests.

The select query has the capability of producing a very large amount of traffic in the system, as it is the only query with a variable number of packets returned. All other queries have at most one acknowledgement per incomming packet.

Example usage:

```sql
SELECT name FROM Dogs WHERE owner = "Arthur";
SELECT name, owner FROM Dogs;
SELECT * FROM People WHERE age > 5 and age < 20;
SELECT lastname FROM People WHERE name = lastname;
```

Internally the following steps occur:

1. User issues query of format *SELECT *|fiel1, ... FROM table WHERE condition*, which is converted and sent as a single packet to the SpiNNaker board.

2. Upon receival, *root* node issues a multicast packet to every *leaf* node on the board, requesting search for all rows which match the WHERE criteria.

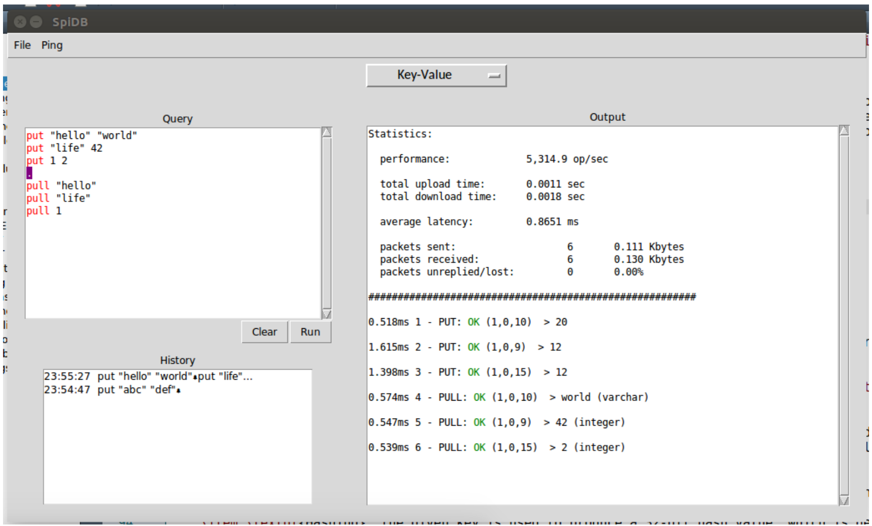

Fig. 3.7: Graphical User Interface

3. Every *leaf* node linearly checks appropriate condition and forwards matching column-value pairs on packets to their appropriately assigned *branch* nodes.

4. *branch* nodes aggregate fields if requested, controls speed of execution and sends packet to host with such entry definition.

5. User receives an arbitrary amount of entries, displayed as a table of results.

Complexity: linear to the size of the input, linear to database size.

3.3.3 User Interface

Although a graphical user interface was not an essential part of the project plan, a simple one was created for the purposes of demonstration, data plotting and ease of visualization. By making use of the SpiDB API, the UI includes features to import *sql* files, ping the board, issue concurrent queries, visualize previous queries, display result statistics and graphs. An example instance can be seen in figure 3.7.

3.4 Testing and Debugging

I started the project with a Test-Driven Development approach, writing Python tests with the *unittest* module and C assertions with part of the SpiNNaker API module *debug.h* and my own code. This allowed high reliability from the start. Running tests can be seen in figure 3.9. Testing and development was done from the bottom up, starting from internal memory management, core communication and finally the user interface and host communication. Testing

Operation	arg1	arg2	expected result	reason
PUT	”hello”	”world”	success	
PUT	”foo”	”bar”	success	
PUT	”A”	”B”	success	
PUT	42	”life”	success	
PUT	”abc”	123	success	
PUT	1	2	success	
PUT	1.5	2.5	failure	no floating point support
PUT	”hello”	”earth”	failure	key already exists
PUT	(very long string)	”value”	failure	256-byte limit for now
PUT	(very long int)	”value”	failure	256-byte limit for now
PUT	(empty)	”string”	failure	both key and value must be set
PUT	”你”	”好”	failure	ASCII support only
PULL	”hello”		”world”	
PULL	42		”life”	
PULL	”abc”		123	
PULL	”HELLO”		failure	case sensitive
PULL	(very long string)		failure	256-byte limit for now
PULL	”a new key”		failure	key does not exist

Table 3.1: Examples of Key-value store test cases

involved initially a number of insert queries with different types and sizes. Given the SpiNNaker architecture is word aligned, it was important to test boundaries along data sizes along multiples of 4 bytes. Examples of tests run can be seen on table 3.1. Both the key-value store and the SQL-RDMS were also tested for their performance and scalability, running over 100,000 symultaneous incomming queries with real data pulled from a modern English vocabulary. The results from these experiments can be read on chapter 4.

Realtime debugging on the SpiNNaker board is relatively hard, but luckly the API provides ways to log messages in each core’s private data memory, which can be read by an external process upon execution of the program. Debugging was performed with a tool named *ybug* (*https://github.com/SpiNNakerManchester/ybug*), also developed by the team, which allows core status checking, uploading binaries, reading logs and memory content, among other functionality.[8] On the *host* Python code, debugging was done using the Pycharm IDE runtime tools.

3.5 Challenges

This section outlines a list of different challenges and problems I had to face during the development of the application and how they influenced decision making.

3.5.1 Out-of-order execution

A multi-threaded system must account for the fact that queries may not be executed in the order they are issued, which can be a concern depending on the application. As SpiNNaker is a distributed architecture, it cause problems and inconsistencies. For example, using SpiDB, if

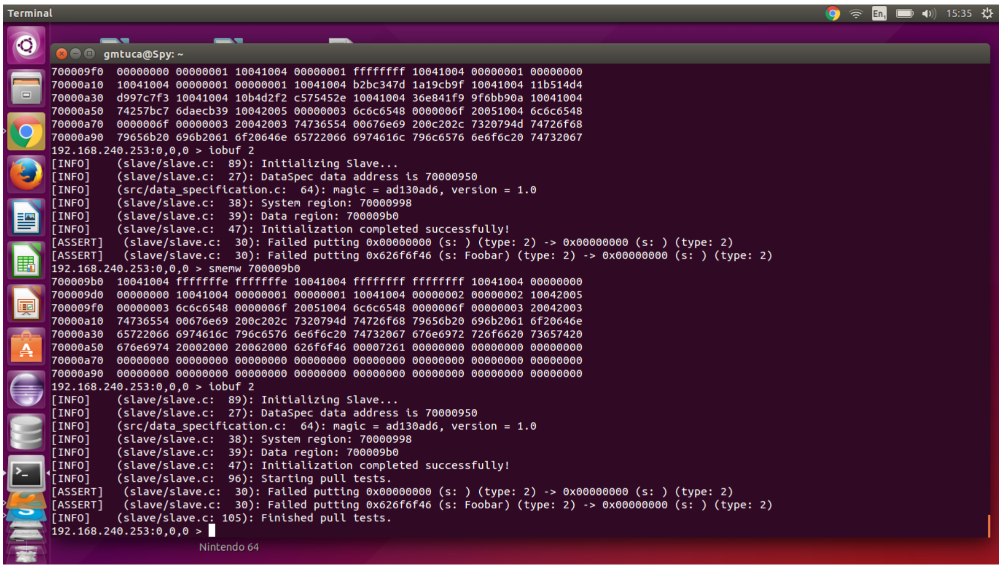

Fig. 3.8: Debugging

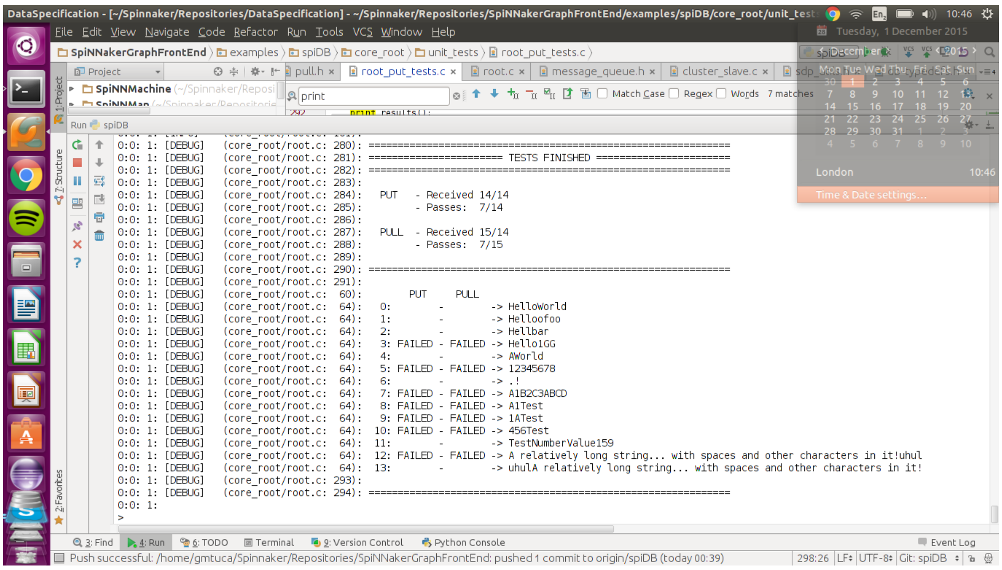

Fig. 3.9: Testing

we try to execute the following code sequence in order:

Listing 3.1: Non-blocking execution

```
put "hello" "world"
pull "hello"
```

We are not guaranteed that the *pull* query will retrieve the value ”world”. As both queries are executed symultaneously on the SpiNNaker board, there is a chance that the *pull* operation will terminate before the *put* does, thus failing to return a value.

As a solution to this dependency constraint, my SpiDB database API includes the syntax symbol ”.” (dot) which blocks execution until all preceeding operations terminate. This allows the programmer to control execution flow, choosing what is allowed to compute out-of-order (in parallel) and what should be sequentialised at a cost of performance. This is a similar concept to the Verilog blocking and non-blocking assignments.

Listing 3.2: Blocking execution

```
put "hello" "world"
.
pull "hello"
```

The above code will assure sequential code, meaning the *pull* operation will always return ”world”. It is usually a good idea to block execution when given a dependency constraint. This can also be done with larger code fragments.

Listing 3.3: Blocking execution

```
put "hello" "world"
put "life" 42
put 123 456
.
pull "hello"
pull "life"
pull 123
```

It is worth noting that although non-block operations can cause out-of-order execution, it does not occur very frequently. This is because, when transmitting data to the board, queries are serialised over ethernet in order of appearence and in addition there is a small forced transmission delay between these packets. This will be further discussed on chapter 4.

3.5.2 Unreliable communication

The SpiNNaker communication protocol can be unreliable, as packets are not guaranteed to arrive at destination, as discussed earlier in section 2.2. This effect is worsen when there is large traffic in the system, so the more packets are being sent, the more packets are lost. While developing the application I had to consider this cost, assuring reduced communication when possible.

Queries such as *put* issue only one packet at a time, allowing an end-to-end acknowledgement without significant packet drops or performance costs. For this reason the *put* query, alongside the SQL commands *create* and *insert* always expect a response, which contains timing information and a flag whether it was successful or not. The steps of running SpiDB queries can be revised in section 3.3.

On the other hand the *pull* and *select* commands issue multicast packets (one-to-many). This means for each query, a packet is sent to up to hundreds of codes, which SpiNNaker is particularly optimized for, but the opposite (many-to-one) is more difficult. Multiple cores sending *SDP* packets to a single core results on a very large packet drop. Only a total of 4 of these packets arrive successfully, as it is the limit of *SDP* channels per core. In the worst case, too many packets have also the capability of crashing the destination core. This means when a packet has multiple destinations, acknowledgements would worsen the situation, as most of these would be dropped and it would highly increase traffic in the system, which itself is a reason for packet loss.

This analysis has lead me to make the design decision of not using internal acknowledgements for every packet sent. In *pull* queries, *leaf* nodes only respond when they have found the specified key, which means there will be at most one acknowledegement. If the key is not found on the database, not a single *leaf* core will respond, which can only be known externally with the use of a timeout.

The advantage of this approach is that it increases speed of execution for successful queries and avoids excessive traffic in the system, increasing reliability. This comes at the cost of slow execution of instances of *pull* requests where the key does not exist on the database. Using this approach performance is poor when requested entries do not exist. Assuming most of the time users will try to retrieve entries previously inserted on the database, I evaluate this decision to be wise.

3.5.3 API Bugs

I have been one of the few users of a large amount of the SpiNNaker API, currently under contruction. This means some of it has not been fully tested, resulting in strange behaviour at times, making debugging of my own application difficult. Besides aiming to gather benchmarks for SpiNNaker, my project itself has been very useful when exposing unexpected errors or inconsistencies in the team's codebase.

Facing these issues was certainly a challenge, as they belonged to domains I had little knowledge of. I was a good opportunity for me to learn and improve code quality of a large, collaborated project, by testing its usability and evaluating its outputs. This means my project involved not only developing the database application, but also collaborating to improve SpiNNaker itself.

The main API bugs I exposed and helped resolve were:

- **Duplicate packets**: when multiple distinct SDP packets were sent from different sources to the same destination, strangely the receiving core would read the same packet duplicated a number of times. This behaviour was highly unexpected to the team, so upon extensive testing, I was able to point the issue to a running processor named *reinjector*, in charge of re-issuing lost packets, and resolve the problem.

- **Data Specification Error**: when uploading binaries to the board, sometimes cores would crash with an internal error state (SWERR), upon evaluating and providing the team with detailed feedback on the issue, I found this to be caused by the SpiNNaker Data Specification framework, which handles allocation of data in shared RAM.

Chapter 4

Evaluation

This research project has among its main aims the evaluation of results, described in this chapter. It is important to analyse how this general purpose application performs under the SpiNNaker environments and what architectural limitations it faces, which is useful feedback to the team. The chapter outlines also performance benchmarks of SpiDB in comparison with other database systems and how it can be improved.

4.1 Transmission delay

According to my experiments, sending SDP packets immediately between two cores results on a very large packet drop ration. Without explicit delay between the transmission of packets, the number of successful deliveries is only about 10% of those sent for a large number of packets. This effect is strongly reduced by the addition of a small delay of 5-10μs, which guarantees over 99% successful deliveries, as can be seen on table 4.1. The table also shows that having long delays, greater than 10μs is mostly redundant, as they do not decrease packet loss significantly, but tragically reduce throughput.

Upon experimenting, I found that sending a large amount of consecutive, immediate packets (at least 1,000,000) has a chance of consequently crashing the destination core or causing it to reach the erraneous watchdog state (WDOG), meaning it cannot respond to commands.

Note that the experiment, made on SpiNN-3, involved sending a single SDP packet multiple times to the same destination on the same chip, containing only the 8 byte SDP header. The destination did not store or read the packet contents, only incrementing a counter upon packet receival. This was used to show the maximum possible transmission rate allowed by the SDP protocol under the hardware communication fabric. Code snippets and more information can

	Successful deliveries (%)				
Packets sent	**no delay**	**2μs delay**	**5μs delay**	**10μs delay**	**100μs delay**
50,000	9.56%	57.73%	95.95%	98.36%	99.85%
100,000	12.15%	54.97%	97.99%	99.13%	99.92%
200,000	13.07%	50.55%	99.01%	99.33%	99.96%
500,000	12.97%	50.08%	99.49%	99.80%	99.99%
1,000,000	13.05%	45.06%	98.84%	99.88%	99.99%

Table 4.1: Successful SDP deliveries with delay between each packet

be found on the appendix under the appendix section A.3.

Ideally these packets would send useful information, to be read upon arrival, which would keep the destination busy. From my experience and input from the team, the best way to achieve this is by immediately storing the SDP packet contents into a buffer when it is received and then handling it at a different point in time (listing 4.1). The reason for this is that if another packet arrives as we are processing the current packet with same or higher priority, the incomming packet will drop.

High priority should be assigned to storing incomming packets into a buffer and the actual processing should have lower priority, as it can be handled at a later point in time. It is important to note that this can cause starvation and buffer overflow if there are too many immediate packets being received. For instance, if our SDP packet is of size 12-byte (8-byte header + 32-bit data) and stored into a buffer in private DTCM memory upon receival, we would only ever be able to hold up to about 5,000 messages at once (64-Kbytes DTCM size / 12-byte packet size). Realistically a large part of DTCM contains the stack, local and global variables, so that number will be drastically reduced. In my application, SpiDB, insert and retrieve database queries have a size of 256-bytes, which means a limit of 250 entries in the buffer if memory were empty.

This evaluation is important because it allows finding the optimal transmission delay for an application. A developer on SpiNNaker using the point-to-point protocol needs to find the balance between transmission rate and reliability, which are inversely proportional. This balance is application dependent.

Listing 4.1: Storing incomming packets into a queue

```
//buffer to store incomming packages
sdp_msg_t** msg_cpies = (sdp_msg_t**)sark_alloc(QUEUE_SIZE, sizeof(sdp_msg_t*));
uint i = 0;

void sdp_packet_callback(register uint mailbox, uint port) {
    //immediately store incomming packet contents into a queue

    i = (i+1)%QUEUE_SIZE;
    register sdp_msg_t* m = msg_cpies[i];
    sark_word_cpy(m, (sdp_msg_t*)mailbox, sizeof(sdp_hdr_t) + SDP_DATA_SIZE);
    spin1_msg_free((sdp_msg_t*)mailbox);

    // If there was space, add packet to the ring buffer
    if (circular_buffer_add(sdp_buffer, (uint32_t)m)) {
        if (!processing_events) {
            processing_events = true;

            //trigger lower priority request processing
            if(!spin1_trigger_user_event(0, 0)){
                log_error("Unable to trigger user event.");
            }
        }
    }
```

```
    else{
        log_error("Unable to add SDP packet to circular buffer.");
    }
}

void process_requests(uint arg0, uint arg1){

    uint32_t mailbox;
    do {
        if (circular_buffer_get_next(sdp_buffer, &mailbox)) {
            sdp_msg_t* msg = (sdp_msg_t*)mailbox;

            ...

        }
        else {
            processing_events = false;
        }
    }while (processing_events);

...

//priority assignment
spin1_callback_on(SDP_PACKET_RX, sdp_packet_callback, 0);
spin1_callback_on(USER_EVENT, process_requests,     2);
```

On SpiDB, I experimented with different time delays when transmitting packets from host over ethernet to a core on SpiNNaker, in order to find the best evaluation. The speed of one operation is calculated as the time of successful reply minus the time that operation was sent from host, thus being a round-trip-time plus internal processing time. The performance is calculated as the amount of operations successfully executed in one second.

As can be seen on figure 4.1, large packet transmission delays of 50-100μs are redundant, as they do not reduce packet drops, while being a high cost on performance. Naturally the more packets we can send in one second influences the speed of replies, thus improving performance. This hits a maxima at 40μs, with almost 10,000 op/sec, in which transmission is at a high rate with no loss of reliability. Transmission delays between 40-10μs result on a decrease of performance, because although packets are sent more frequently, a lot of them are dropped (up to about 35%), being also extremely unreliable. We reach the worst case at 10-5μs delay, when the destination core cannot cope with the speed of incomming packets, simply crashing and ceasing to respond. This SpiDB experiment was performed with 100,000 *put* operations with keys and values of size 4-bytes each. More information on the data gathered can be found on the appendix under section A.3.

4.2 Performance benchmark

(still needs writting)

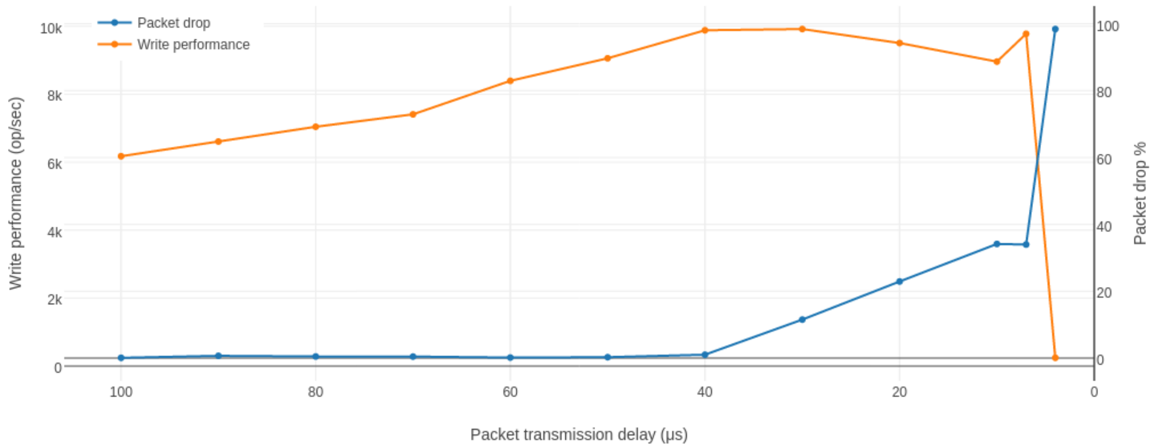

Fig. 4.1: Transmission delay plot

4.3 Limitations

(still needs writting)

4.4 Future work

This project has been the tip of the iceberg of what a fully-functional SpiNNaker database management system can be, involving mostly research and simple operations to evaluate its performance. All code written by me is now part of the official open-source SpiNNaker API, available to the team and any developers interested. This means SpiDB is likely to expand in the future or serve as an example application running on SpiNNaker, accessible to researchers around the globe.

These are some features which can be implemented or improved in the future:

- **Caching** and **indexing**: frequently accessed areas of shared SDRAM memory could be cached at the smaller but much faster private DTCM.
- **Security** and **multi-user access**: different sections of the database can have restricted access through credentials checking.
- **Scalability testing** on the large scale million core machine.

- An application **server** allowing queries to be requested over the internet on different locations.
- Improve **reliability** and increase **query sizes**, perhaps by implementing a protocol on top of the *SDP* and *MC* layer. As of now packets are limited to 256-bytes, with unreliability during busy times.
- **Self balancing** during idle times. While no queries are being executed, cores could distribute their contents in a balanced way for faster retrival. Indexing or other pre-processing could also be executed on the meantime.
- **Additional operations** supporting table merges, triggers and aggregations.

Bibliography

[1] "The free on-line dictionary of computing," March 2016.

[2] E. P. et al., "Spinnaker: A 1-w 18-core system-on-chip for massively-parallel neural network simulation," *IEEE Journal of solid-state circuits, vol. 48*, August 2013.

[3] "Spinnaker project," April 2016.

[4] "Spinnaker project - the spinnaker chip," April 2016.

[5] S. Furber, *SpiNNaker Datasheet - Version 2.02*. University of Manchester, January 2011.

[6] "Arm968 processor," April 2016.

[7] S. Temple, *AppNote 4 - SpiNNaker Datagram Protocol (SDP) Specification - Version 1.01*. SpiNNaker Group, School of Computer Science, University of Manchester, November 2011.

[8] S. Temple, *ybug - System Control Tool for SpiNNaker - Version 1.30*. SpiNNaker Group, School of Computer Science, University of Manchester, April 2014.

Appendix A

Example of operation

A.1 Experiments Specification

The **memcached** and **Redis** benchmarks were run on an ASUS X555L quadcore Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 8gb DDR3 RAM @ 1600 MHz. SpiNN-3

A.2 Queries

```
typedef enum var_type  { UINT32=0, STRING } var_type;

typedef struct Column {
    uchar        name[16];
    var_type     type;
    size_t       size;
} Column;

typedef struct Table {
    uchar        name[16];
    size_t       n_cols;
    size_t       row_size;
    size_t       current_n_rows;
    Column       cols[4];
} Table;
```

A.3 Communication Reliability

This section contains code snippets to send SDP packets from single source to single destination with delays between them. These were used in the experiments described on chapter 4, section 4.1.

Transmitting packets with variable delay:

Listing A.1: Source

```
uint rcv = 0;

void sdp_packet_callback(register uint mailbox, uint port) {
    rcv++;
    spin1_msg_free((sdp_msg_t*)mailbox);
    return;
}

...

spin1_callback_on(SDP_PACKET_RX,     sdp_packet_callback, 0);
```

Listing A.2: Destination

```
void send_sdp_packets(uint32_t number_of_packets, uint32_t delay,
    uint32_t chip, uint32_t core){
        sdp_msg_t msg;
    msg.flags        = 0x87;
    msg.tag          = 0;

    msg.srce_addr    = spin1_get_chip_id();
    msg.srce_port    = (SDP_PORT << PORT_SHIFT) | spin1_get_core_id()
        ;

    msg.dest_addr    = chip;
    msg.dest_port    = (SDP_PORT << PORT_SHIFT) | core;
    msg.length           = sizeof(sdp_hdr_t);

    for(uint i = 0; i < number_of_packets; i++){
        if(!spin1_send_sdp_msg(&msg, SDP_TIMEOUT)){
                log_error("Failed to send");
        }
        sark_delay_us(delay);
    }
}
```

SpiDB performance with variable delay:

interval (µs)	**packet drop (%)**	**performance (ops/sec)**
100	0.076	6175
90	0.670	6608
80	0.475	7042
70	0.445	7408
60	0.150	8394
50	0.265	9057
40	1.010	9882
30	11.510	9918
20	22.910	9506
10	34.140	8960
7	33.965	9778
4	98.432	243

Table A.1: SpiDB insertion performance with variable transmission delay