Rapid Evaluation of Permutation Entropy for Financial Volatility Analysis – *A Novel Hash Function using Feature-Bias Divergence*

Jun Ren Lim
Department of Computer Science
Imperial College London

19th June 2014

Abstract

In the wake of the flash crash on 6 May 2010, there is a surge of interest in volatility analysis of financial time series, with the purpose of pre-empting future flash crashes. If this can be achieved to some degree, more informed decisions could be made by regulators to mitigate the resultant market turbulence and keep hold of investor confidence. The existing techniques for time series analysis range from machine learning to genetic algorithms. However, they fall short in capturing the pattern trends of time series. The use of Permutation Entropy (PE), a measure for arbitrary time series based on analysis of permutation patterns, is proposed. Typically used in the biomedical field, PE's simplicity and potential for fast calculation are vital advantages in analysing today's high-frequency data.

The optimization of the speed of permutation hashing is crucial to the fast calculation of PE. In the context of hashing a large sequence of permutations, the number of hash collisions has a major influence on hashing speed. Therefore, this paper devises an original hashing algorithm specialized in minimizing collisions when hashing permutations. This novel hash function dissects the subtle relationships between permutation patterns and contrives to hash different permutations into distinct slots. Preliminary testing demonstrates its promise to be a good permutation hash function. Furthermore, GPU acceleration of hashing permutation patterns is performed through parallel programming with CUDA.

Acknowledgements

I would like to thank the following people for their support in this project:

- Professor Wayne Luk for his patient supervision and advice throughout,
- Dr. Stephen Weston for his original insights on Permutation Entropy and his words of encouragement,
- Guo Ce for being always there to answer my queries and provide precious opinions, and lastly,
- Jun Wei and Mike for teaching me the basics of CUDA.

Contents

List of Figures

List of Tables

List of Definitions

List of Theorems

Chapter 1

Introduction

The analysis of financial time series is growing in importance to the economic world. Financial time series involve data of a sequence of prices of certain financial assets over a particular period of time. Most financial time series processes are non-stationary and time-dependent. Price sequences are in high frequency and especially volatile in the stock market, giving much room for technical analysis, which is the investigation of past price movements to predict future prices. This paper is based on the proposal of the use of Permutation Entropy in financial time series prediction. Permutation Entropy is founded upon the theoretical basis of technical analysis that (1) price is an explicit manifestation of all concealed mechanisms, (2) price movements are not random, and (3) history repeats itself.

In particular, rapid evaluation of Permutation Entropy is inextricably linked to optimization of the hashing of permutations. This paper primarily focuses on the **proposal, implementation and performance evaluation of a novel hash function** to optimize the hashing of a large sequence of permutations based on a given financial data series.

1.1 Motivation

In the wake of the flash crash on 6 May 2010, there is a surge of interest in attempting to grasp the relationships between the sudden extreme volatility and the price movements just before the onset of the crisis. If similar future flash crashes can be pre-empted to some degree of accuracy, more informed decisions could be made by regulators to mitigate the resultant turbulence in the financial market and keep hold of investor confidence.

Financial volatility analysis is certainly not new. There are several well-documented volatility-based linear and non-linear forecasting models in financial theory, such as AutoRegressive Moving Average (ARMA) (Box and

Jenkins, 1970) and ARMA-GARCH (Engle, 1982) models. However, there is a fundamental flaw in volatility analysis applied to price movement prediction, which is the failure to take into account the exact patterns of price sequences – a rapidly increasing sequence of upward price movement yields no difference in interpretation to a rapidly decreasing price sequence! Pattern recognition techniques are definitely better poised to address this flaw.

The existing techniques for time series pattern analysis include machine learning implementations[1] and genetic algorithms[2]. Data mining methodologies, which aim to unveil concealed patterns and consequently examine relationships among these patterns, have been used in time serial databases[3]. Financial time series exhibit sophisticated patterns such as trends and volatility clustering that vanish and resurface almost regularly. However, most of the time series analysis techniques discussed above possess limitations in capturing such pattern trends of non-stationary time series[4]. Also, most of them do not perform well in the presence of noisy data in real-world time series.

A key consideration for choosing a good technique for time series pattern analysis is the issue of *speed*. The high-frequency torrent of stock market data dictates that computational speed is of utmost importance, so as to ensure a fast response time between receiving data and execution of pattern analysis.

1.2 The Idea: Using Permutation Entropy

In a recent paper, Bandt and Pompe[5] proposed Permutation Entropy (PE) as a natural complexity measure for arbitrary time series which may be stationary or non-stationary, deterministic or stochastic, regular or noisy. Their method is based on a comparison of neighbouring values, and fundamentally involves calculation of a PE index given a pattern of order (i.e. size) N. The advantages of this method are its simplicity, robustness, invariance with respect to non-linear monotonous transformations, and most importantly potential for overwhelmingly fast calculation, which is vital in analysing today's high-frequency stock market data.

1.3 Objectives

Today's existing approaches to volatility analysis of financial time series bear the fundamental weakness of being unable to capture and distinguish observable price patterns, while most of the current pattern-recognition machine-learning methods including Hidden Markov Model and Artificial Neural Net-

work are smart guesswork that are not convincingly founded upon a clear fundamental mathematical principle in pattern analysis.

Founded on the concept of Permutation Entropy (PE), this paper's objectives include the **proposal** and **implementation**, and **performance evaluation** of a **novel hash function** specific to dealing with an input of a large sequence of permutations resulting from any given time series. This hash function aims to even out hash collisions as much as possible, which, if achieved, will contribute vastly to speed of hashing and consequently increase responsiveness to real-world high-frequency price movements.

1.4 Challenges

As we seek to calculate PE indices for increasingly large N (where N is the order of a permutation pattern π), we face the following challenges in speed and memory:

1. **Hashing**
   Since we need to keep count of the number of occurrences of every permutation for a given large data sequence, an implementation of a hash table, with permutations as keys, is necessary. However, as N increases linearly, the number of possible permutations increases factorially to $N!$. This means that the range of possible keys can easily reach an extreme value when N is sufficiently large. If this happens, a one-to-one mapping between key and hash table index is infeasible in terms of memory usage. A smart, customized hash function that strikes a good balance between speed and memory is needed.

2. **Big Number Manipulation**
   When N is large (say $N > 12$), $N!$ enters the "big number" zone. Therefore the abovementioned customized hash function could involve big number manipulation, which is another speed-memory optimization conundrum.

This paper seeks a **detailed solution to the first challenge of hashing**, while any discussion addressing the second challenge of big number manipulation will be peripheral at most.

1.5 Contributions

Based on our objectives to propose, implement and accelerate a novel permutation hash function, our contributions are as follow:

1. We propose an original hashing algorithm specialized in minimizing collisions when hashing permutations. We detail the entire rationalization process in coming up with this algorithm, along the way providing theoretical justification of the algorithm's capability to avoid collisions. We name this novel hashing algorithm the FBD hash. The construction of the FBD hash is expounded in Chapter 3.

2. We detail the implementation of the FBD hashing algorithm. As there is a loop in the algorithm's final step, we provide a vigorous, non-trivial proof of termination of that loop. Due to the loop being the algorithm's bottleneck, we perform termination time analysis on the loop by constructing and proving a probability bound on the loop's survival through i iterations. These can be found in the first main section of Chapter 4.

3. We evaluate the performance of the FBD hash against traditional integer hash functions. For the hash performance evaluation, we use 3 hash performance metrics; among these, 1 is a well-established metric and is hence used as the primary metric, while the other 2 are original hash performance metrics. We analyse the test results and articulate what the results suggest with regard to (1) the promise shown by the FBD hash and (2) the credibilities of the 2 self-proposed hash performance metrics. Finally, we discuss the limitations of the metrics used. These can be found in the second main section of Chapter 4.

4. We explore the options of hardware acceleration of the FBD hash. We explain the infeasibility of the option of hashing in FPGA, and discuss the pros and cons of performing hashing in CPU and GPU. We outline the key features of our CUDA C implementation of (1) our FBD hashing algorithm, (2) multithreaded hash insertions, and finally (3) the computation of the PE. We then evaluate the speed-up that our GPU implementation offers as compared to a CPU implementation. These can be found in Chapter 5.

Chapter 2

Background

2.1 Permutation Entropy

A system whose evolution is observable throughout a certain period begs a burning question – how much information can be deduced about the internal workings of the underlying system? The information content of a system is generally quantified using a probability distribution function (PDF) P that is defined by a time series $M(t)$ [7].

2.1.1 Limitations of classical entropy measures

The Shannon entropy is a classical measure of information content. Given any arbitrary discrete probability distribution $P = \{p_i : i = 1, , F\}$, with F degrees of freedom, Shannon's logarithmic information measure is expressed as[6]:

$$S[P] = -\sum_{i=1}^{M} p_i \log p_i$$

This is a quantification of the uncertainty borne by the physical process characterized by P. For example, if $S[P] = S_{min} = 0$, we can be 100% sure which of the possible outcomes i with probability p_i will occur.

However, this traditional entropy calculation is not without significant limitations:

1. **Failure to capture chronological relationships**
   Shannon's entropy fails to capture chronological relationships between values of the time series[8]. For example, if two time series are defined as $M_1 = \{0, 0, 1, 1\}$ and $M_2 = \{0, 1, 0, 1\}$, we automatically have $S[P(M_1)] = S[P(M_2)]$. This shows that order relations are not preserved.

2. **The prerequisite of fundamental knowledge**
   Fundamental knowledge about the system is a prerequisite for the employment of classical entropy measures. For instance, in using quantifiers based on Information Theory, a probability distribution defined by the time series under analysis has to be known[7]. Even though there exists many established methods to construct a suitable PDF, such as frequency counting, Fourier analysis or wavelet transform, they are not derived from the dynamical properties of the system. This is contrary to the principle of Technical Analysis that maintains that explicit observations are capable of manifesting all underlying information. Hence a more general and system-independent PDF is desired.

3. Classical entropy measures do not perform well with non-linear chaotic regimes.

2.1.2 Permutation Entropy by Bandt and Pompe

The abovementioned drawbacks are absent in the method of **Permutation Entropy** introduced by Bandt and Pompe[5], which captures time causality by comparing adjacent values relatively in a time series. Firstly, no prior knowledge is required. Secondly, there is no notion of magnitudes – *order relations* are sought after instead. Through the comparison of neighbouring values, a permutation entropy that quantifies order relations is calculated.

2.1.3 Permutation Entropy Formulation

The method of Permutation Entropy is as follows: at each time s of a given time series $M = \{m_t : t = 1, ..., T\}$, a sequence comprising the N subsequent values is constructed:

$$s \longmapsto (m_s, m_{s+1}, ..., m_{s+(N-2)}, m_{s+(N-1)})$$

N is called the *order*, and determines how much information is contained in each sequence. To this vector, an ordinal pattern is associated, defined as the permutation $\pi = (r_0 r_1 ... r_{N-1})$ of $(01...N-1)$ which fulfils

$$m_{s+r_0} \leq m_{s+r_1} \leq ... \leq m_{s+r_{N-2}} \leq m_{s+r_{N-1}}$$

This means that the values of each sequence are sorted in an ascending order, and a permutation pattern π is created with the offset of the permuted values. As an example, $M = 4, 2, 5, 1, 6, 7$. When $N = 3$, the sequence of values corresponding to $s = 1$ is $(4, 2, 5)$. Then the sequence is sorted in ascending order, giving $(2, 4, 5)$ and hence yielding the corresponding permutation pattern $\pi = (102)$. For $s = 2$, the sequence of values is $(2, 5, 1)$, yielding the permutation (120).

The basis of employing permutation entropy as a tool for pattern analysis is the expectation that there is a skewed probability distribution of pattern occurrences, and discovery of this distribution will open up doors to learning how the underlying system works. This leads to the extreme but very real idea of *forbidden patterns*, which refer to the patterns that do not appear at all in the time series.

If there are forbidden patterns, why? Assuming that the time series is long enough (to render invalid the trivial justification that the time series is too short to reasonably expect all patterns to emerge), the inescapable proposition is that the time series is not random. With this in mind, take a look at the logistic map, popularized in a 1976 paper by biologist Robert May:

$$x_{t+1} = \alpha x_t(1 - x_t) \qquad \forall x \in [0, 1]$$

The plot below illustrates the results of the logistic map with the parameter $\alpha = 4$[7].

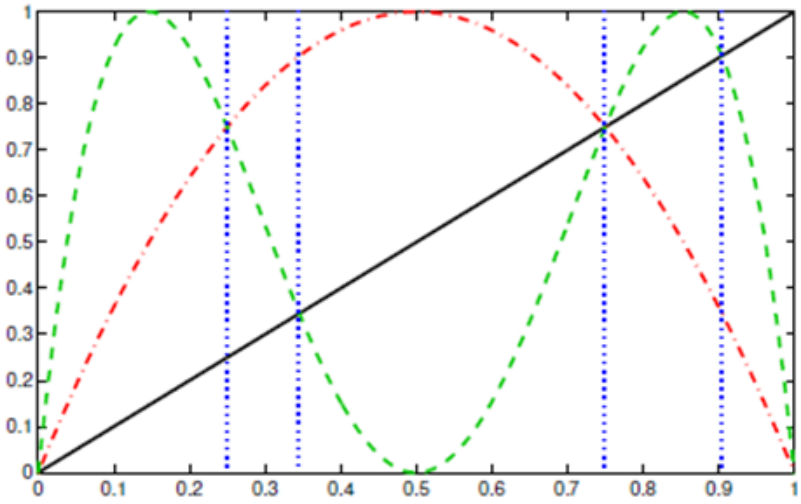

Figure 2.1: Plot of x_1 (red) and x_2 (green) for all possible x_0 (black)

The order of these curves graphically represents the corresponding permutation pattern. It can be observed that 5 different permutations are generated by this map, identified by the 5 regions enclosed by vertical dashed lines, while the number of possible permutations are $3! = 6$. It is striking to note that the permutation $\pi = (210)$ is *forbidden* by the own dynamics of the system.

Henceforth the purpose of this example surfaces: we learn that investigating information regarding the presence or absence of certain permutation patterns of a time series can give us an insight of the dynamics of the underlying system. Or to be more general, the essence of permutation entropy lies in investigating the frequencies of occurrences of each permutation pattern.

Before we formally define Permutation Entropy (PE), we first need to define the concept of a **permutation value** v associated with a permutation pattern D:

Definition 1 (Permutation Value)**.**

Let $\pi \in \Pi$ be a permutation pattern of some sequence of order N data $(d_1, d_2, ..., d_N)$, with Π = the set of all $N!$ possible permutation patterns. The permutation value v of π w.r.t. $\boldsymbol{\nu}$ is given by

$$v = \nu(\pi)$$

where ν can be any[a] **bijective** function that maps Π to $\{1, 2, ..., N!\}$.

A **relaxed definition** of ν, where ν takes in an actual sequence of data values D instead of a permutation π, **is allowed and often used in this paper**. For e.g., consider the data sequence $D = (2, 4, 5)$ that exhibits permutation pattern of (102):

$$\underbrace{\nu(2,4,5)}_{\text{relaxed}} = \underbrace{\nu(102)}_{\text{formal}}$$

[a] The ν function that this paper will use in subsequent chapters is defined later in Definition 5 in Section 3.1

The **Permutation Entropy**, **PE**, is then defined as the Shannon entropy associated to such distribution:

Definition 2 (Permutation Entropy (PE))**.**

$$PE = -\sum_{v=1}^{N!} \pi_v \log \pi_v$$

where π_v is the frequency associated with the permutation value v w.r.t. some[a] ν.

[a]Note that as long as $\nu : \Pi \mapsto \{1, 2, ..., N!\}$ is bijective, the actual definition of ν is inconsequential to the calculation of PE.

2.1.4 Current Biomedical Applications

The use of Permutation Entropy has been growing steadily since its inception, as indicated by the chart[7] below.

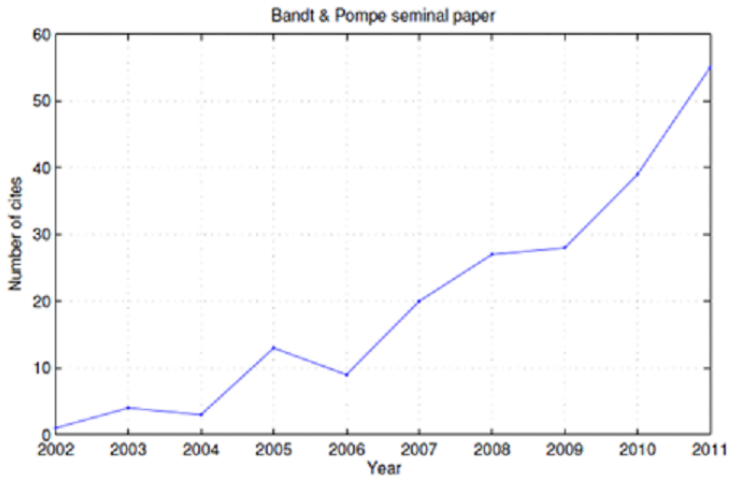

Figure 2.2: No. of cites of Bandt and Pompe's PE in the last decade

2.1.4.1 Reasons for Success

In particular, the use of PE has seen thriving success in the biomedical field. The unique advantages of PE that make it so relevant to the biomedical scene are described as follows:

- Both healthy and pathological states (epileptic seizures being a prominent successful example) are associated with stereotyped patterns of activity – which explains why the usefulness of permutation entropy can be exploited to a high degree.

- Being robust to noise, permutation entropy delivers wonderfully in analysis of biological systems, as biological time series typically contain a certain degree of randomness.
- The simplicity and extreme computational efficiency of permutation entropy contribute to its effectiveness in real-time applications for clinical purposes, where little time is allowed for preprocessing and fine-tuning of parameters.

2.1.4.2 Description of Applications

The use of PE has been markedly successful in epilepsy analysis, among others[7]. A well-known neurological disorder, epilepsy afflicts one out of a hundred people worldwide. An epileptic seizure is associated with unusual and hyper-synchronous brain activity. Moreover, the time of occurrence of seizures are often unpredictable. PE can be used to predict occurrence times with satisfactory certainty.

For epilepsy sufferers who do not respond well to anti-epileptic drugs, neurosurgical resection of epileptogenic brain tissue is an option. To identify this tissue, doctors implant intracranial electrodes in the patients' brain to attempt to identify the location of epileptic activity. In this respect, the use of PE has shown good promise in accurately recognizing patterns of epileptic activity. In addition, with the use of PE, spatio-temporal patterns of various parts of the brain can be characterized, assisting doctors' in their endeavours to spot the epileptogenic brain tissue.

2.1.5 Novel Use in Financial Time Series

The previous discussion on the biomedical applications of PE prompts one's imagination of its promise in the analysis of financial time series. In financial time series analysis, price movements are inextricably linked to an aggregation of hidden economic and psychological factors, there are certainly some degree of random noise littered throughout, and data are real-time and in high frequency. These characteristics are precisely those kind of characteristics that enabled biological time series to fully harness the merits of permutation entropy! Hence in this paper, we explore the exciting idea of applying permutation entropy to a large, high-frequency financial time series.

2.2 Hashing

The purpose of building hash tables is to enable quick location of a record given its search key. This is typically done in O(1) time. A hash function

is used to map a search key to its index in the hash table. A hash function should be **deterministic** – that is, when used to evaluate on two identical data, the function should produce the exact same value. This characteristic is imperative to the correctness of the hashing algorithm. However, hash functions are usually not invertible, meaning it is common for several search keys to hash to the same value, a condition called **hash collision**.

The Pigeonhole Principle: if M items are placed in N buckets, and M is greater than N, at least one bucket contains at least two items. This proves that no hashing algorithm can hash every key to a unique index if the possible keys exceeds the array size. Since hashing often involves using a limited size hash table to contain a broader possibility of search keys, collisions will occur.

Due to the possibility of collisions, average/worst case search complexity, which depends on the hash function, number of input keys and table size, can escalate to undesirable levels. Hence a good hash function to spread collisions evenly, so as to optimize worst case search complexity, is of utmost importance, especially when building a non-trivial hash table to store large amount of keys generated from a non-random distribution.

2.2.1 Well-established Integer Hash Functions

The main aim of this paper is to expound a novel integer hash function to specifically cater to hashing *permutation values* of a data stream (in particular financial data). Traditional integer hash functions would provide a useful benchmark. Here are some of them:

1. **The Remainder Operator**
   The most straightforward hash function for an integer key is *key* mod N, where N is array size.

2. **Additive Hash**
   Sum up all the digits of an integer key and then use the remainder operator.

3. **Bernstein's Hash**
   Bernstein's Hash is illustrated by the code[17] below:

```
unsigned bern_hash (void *key, int length) {
        unsigned char *k = key;
        unsigned h = 0;
        int i;
        for (i = 0; i < length; i++)
                h = 33 * h + k[i];
        return h;
```

```
}
```

Bernstein's hash exemplifies the fact that sometimes a hash function might have little theoretical grounding but is capable of achieving good practical performance: the number 33 does better than other logical constants for no apparent reason!

4. **Robert Jenkins' Integer Hash**
   Developed by Robert Jenkins, this integer hash function is based on a sequence of bit shifts, exclusive-ors and additions to hash the input key. Given just one bit change in the key, this strategic sequence of transformations can impact bits that are spaced significantly apart in the hash output. The Robert Jenkins' integer hash is illustrated by the code[19] below:

```
unsigned int hash(unsigned int a)
{
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
}
```

2.2.2 General Hash Strategies

As this paper focuses primarily on introducing a novel hash function, analysing effectiveness of an overall hash strategy in detail is beyond scope. That said, hash strategies are useful in balancing space-time constraints and could be used to complement a good hash function, hence the two most popular hash strategies are briefly touched on below as possible future extensions to this paper.

1. **Cuckoo Hashing**
   Cuckoo hashing[11] yields constant time complexity for search and constant amortized time complexity for insertions. Its stand-out characteristic is that it uses more than 1 hash function, hence a key-value pair can be in more than 1 location. Let F be an ordered list of these hash functions. When a key-value pair K is inserted, it goes through F, and the first hash function in F that hashes K without collision will be used. If no such hash function exists, then the last hash function $f \in F$ will be used to hash K, and this insertion procedure will be repeated to re-hash the key that collided with K under f. Termination

is when no collision happens or when all hash slots are traversed; the latter event will cause the table to be dynamically resized. Due to the contrived avoidance of collisions, cuckoo hashing can achieve excellent space utilisation.

2. **2-choice hashing**
   2-choice hashing[11] can be considered a simpler version of cuckoo hashing. It uses exactly two hash functions, which map any key-value pair K into two different hash slots. If K is to be inserted, both hash functions will be evaluated and K will be placed in the slot where a smaller number of keys reside. Hence fewer collisions and greater space utilization can be achieved. However, searching may require the traversal of all the key-value pairs contained in two buckets.

2.2.3 Number Theory

The hash function introduced in Chapter 3 uses principles of number theory extensively. The 3 concepts in computational number theory employed in the formulation of Chapter 3's hash function are:

1. Basic linear congruences
2. **Calculating inverses** modulo a prime via the **extended Euclidean Algorithm**
3. **Primality testing** using the **Miller Rabin's test**

2.2.3.1 Basic Linear Congruences

Definition 3 (Linear Congruence)**.**

Let $m \neq 0$ be an integer. Two integers a and b are **congruent modulo m** if there is an integer k such that $a - b = km$, or $a \equiv b \mod m$.

Basic Properties (Linear Congruence)**.**

For any integers a, b, c, and $m \neq 0$, the properties of reflexivity, symmetry and transitivity hold. Furthermore under modulo m,

$$a + b \equiv (a \mod m) + (b \mod m)$$

$$a \times b \equiv (a \mod m) \times (b \mod m)$$

2.2.3.2 Primes

The following two basic properties of primes are fundamental to the construction of the hash function introduced in Chapter 3.

Basic Properties (Primes)**.**

Let p be a prime.

1. Let a, b, c be any integer that is not a multiple of p. Then $ab \not\equiv ac \mod m$.
2. Let $S = \{1, 2, 3, ..., p-1\}$ be the set of all congruences modulo m excluding 0. Let S' be the set generated when p is multiplied by every element in S under modulo m. Then $S' = S$.

2.2.3.3 Computing Modular Inverses

Efficient computation of modular inverses of a prime is quintessential to the implementation of Chapter 3's hash function. This section outlines a leading algorithm for computing inverses modulo a prime p.

Definition 4 (Modular Inverse)**.**

Let $a, p \in \mathbb{Z}$ with p prime. We say that $z \in \mathbb{Z}$ is a **multiplicative inverse of** a modulo p if $az \equiv 1 \mod p$.

Theorem 1 (Uniqueness of Modular Inverse)**.**

There exists a unique multiplicative inverse of a modulo p iff. $a \not\equiv 0 \mod p$.

Given $a, b \in \mathbb{N}_0$, their greatest common divisor $d := gcd(a, b)$ can be computed by the well-known *Euclidean algorithm*, which is founded on the invariance of the gcd when a smaller integer is subtracted from a greater integer. Continuously iterating this procedure will cause the greater of the two numbers to decrease progressively, hence there will be eventual termination[12].

The **extended Euclidean algorithm** goes a step further and allows us to efficiently compute integers s and t such that $as + bt = d$. Supposed we are given integers a, b, n, where $0 \leq a, b < n$, and we want to compute a solution z to the congruence $az \equiv b \mod m$. Here is the extended Euclidean algorithm in the form of pseudo-code[15] below:

```
d = gcd(a, n)
if !(b = 0 mod d) then
        output "no solution"
else
        A = a/d, B = b/d, N = n/d
        t = inverse(A) mod N
        z = tB mod N
output z
```

Based on the extended Euclidean algorithm, computing modular inverses of a prime p can be achieved, by setting $b = 1$ and $N = p$. The following theorem gives a bound on the time complexity of the algorithm.

Theorem 2 (Time Complexity)**.**

The extended Euclidean algorithm for calculating a modular inverse of a prime p runs in $O(p^2)$ time[a].

[a]The proof can be found in Victor Shoup's *A Computational Introduction to Number Theory and Algebra*[15]

2.2.3.4 The Miller-Rabin Test for Primality

The construction of the hash function (fully expounded in Chapter 3) involves locating a suitable prime p. This is done by using the **Miller-Rabin test**, which is a well-known fast test for primality executed in polynomial-time. Though a number passing the Miller-Rabin test (with a reasonable number of iterations, say around 20) is not guaranteed 100% to be prime, it has an extremely high probability of being prime. A bound for the accuracy of the Miller-Rabin test is given below:

Theorem 3 (Accuracy of the Miller-Rabin test)**.**

If N multiple independent tests are performed on a composite number, then the probability that it passes each test is $\leq \frac{1}{4}^N$.

Hence the Miller-Rabin test gives a very good trade-off between speed and chance – and hence its use is expedient to the hashing algorithm outlined in

Chapter 3 which could involve the search for very large primes.

Based on the properties of strong pseudoprimes,the algorithm[16] proceeds as follows: Given an odd integer n, let $n = 2^r s+1$ with s odd. Then choose a random integer a with $1 \leq a \leq n-1$. If $a^s \equiv 1 \mod n$ or $a^{2^j s} \equiv -1 \mod n$ for some $0 \leq j \leq r-1$, then n passes the test. A prime will invariably pass the test. The test is extremely quick and involves at most $(1 + O(1))^2 \log n$ multiplications modulo n.

If we want to find the smallest prime that is > an arbitrary positive integer x, we can make use of Bertrand's Postulate to set a definitive range over which to perform the Miller-Rabin tests.

Theorem 4 (Bertrand's Postulate)**.**

For every $n > 1$, there is always at least one prime p such that $n < p < 2n$.

2.3 Hardware Acceleration of Hashing

This section delineates the main methods that can be possibly used for accelerating the computation of the hash values of the permutations of a given time series. Hashing acceleration is imperative if high-frequency tick data is used, as the demand will then be to compute the hash values of a large series of permutations in a very short amount of time. The race against time for hashing a large series of permutations can potentially be boosted by two cutting-edge technologies – (1) **CUDA** and/or (2) **FPGAs**.

2.3.1 Parallelization using CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA and implemented by the GPUs (graphics processing units) that they produce[14].

CUDA enables GPUs to be used for general-purpose processing. In contrast to traditional CPUs that specialize in high-speed execution of a single-threaded application, GPUs have a throughput architecture that is capable of executing a large number of threads in *parallel*, though each individual thread runs at a low speed. CUDA is available in the form of extensions to many programming languages including C.

The CUDA processing flow is best illustrated with a diagram[13]:

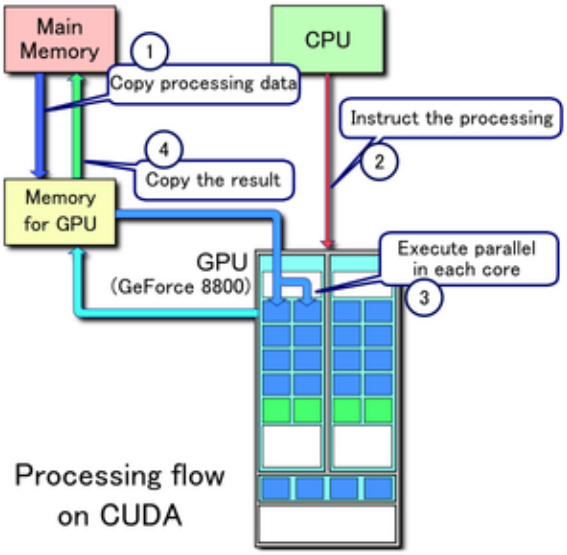

Figure 2.3: Illustration of the CUDA processing flow

2.3.2 Dataflow Computing

Sequential execution is an inescapable characteristic of traditional computer architecture, where data are stored in a central memory. The dataflow approach was first suggested by Karp and Miller. In a dataflow computer, a program is not represented by a linear instruction, but by a dataflow graph. Data flows to instructions, causing evaluation to occur as soon as all operands are available. Data is sent along the arcs of the dataflow graph in the form of tokens, which are created by computational nodes and placed on output arcs. They are removed from arcs when they are accessed as input by other computational nodes. Concurrent execution is a natural result of the fact that many tokens can be on the dataflow graph at any time[9].

As an example, the computation of the following statements

$$A := B \times C + D/F$$

$$G := H \times 2 + A$$

is represented by the dataflow graph in the diagram below[9].

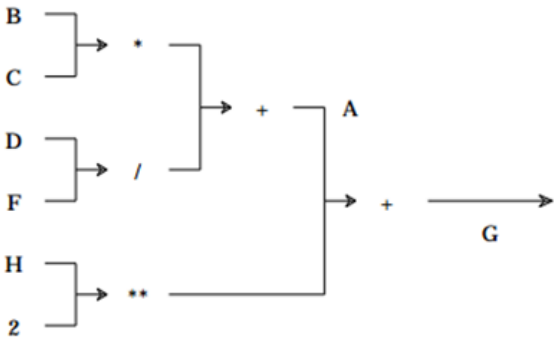

Figure 2.4: Dataflow Graph

The dataflow graph manifests the concept of a huge computational pipeline, which streams in input data in a sequential order and streams out output data. Consequently, a throughput rate of one value per cycle can be achieved. This means that a design running at few hundred megahertz can outperform a CPU implementation running at a few gigahertz!

2.3.2.1 FPGAs and the Maxeler Platform

Field Programmable Gate Arrays (**FPGAs**) are hardware chips that may be configured and programmed to execute combinational and sequential logic. They comprise an array of programmable logic blocks, each connected by sets of reconfigurable wires so as to allow signals to be transmitted while adhering to the user-defined circuit.

Users define custom circuits through hardware descriptive languages (HDL). The process of writing HDL to implementation in hardware is as follows: the HDL is first compiled and then converted into a configuration that programs the logic blocks and routes the signals within the FPGA according to the HDL's circuit definition.

The leading platform offering a comprehensive software and hardware acceleration solution based on dataflow computing is the **Maxeler Platform** created by Maxeler Technologies. Housing the latest-generation, largest FPGAs available, Maxeler's hardware platforms are specifically designed for high-performance computing through emphasis on low-latency and high-throughput. On the other hand, Maxeler's software solution boasts of the high-level MaxCompiler programming tool suite, which offers a Java-based environment for stipulating hardware designs before manufacturing a hardware implementation.

2.4 Related Work

An FPGA implementation of the concept of permutation entropy has been done by Ce, Guo. In the process, the calculation of permutation entropy has been optimized for speed through elimination of the need to sort data values and some additional slight reformulation of the traditional algorithm.

2.4.1 Ce's FPGA Implementation of Permutation Entropy

The flowcharts below illustrate the dataflow graph representing Ce's implementation of a permutation value calculation[1] using FPGAs.

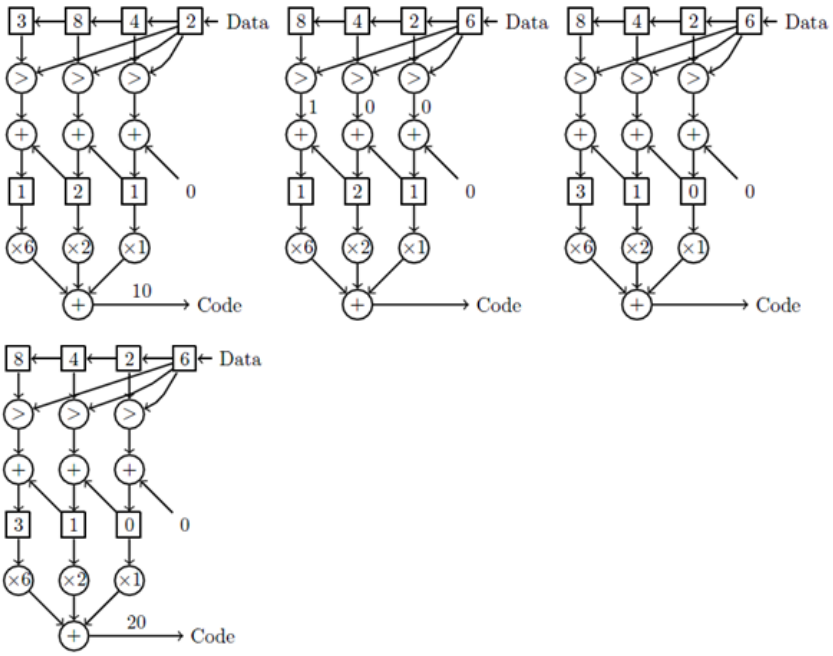

Figure 2.5: Dataflow Graph to calculate permutation value (order $N = 4$)

Starting from the top left corner of Figure 2.5, the current 4 values ($N = 4$) that are under consideration from the sample input data streaming in from the right is $(3, 8, 4, 2)$. Instead of directly sorting these values to derive their permutation pattern, the number of values before (i.e. to the right side of) each digit that are smaller that digit are tabulated. This effectively exploits the parallelism of dataflow technology, and renders the calculation of permutation entropy much faster. So take for example the leftmost number 3 (top left of Figure 2.5): only the number 2 before it is smaller than it, so the number 1 is produced as a result. Whereas for 8,

[1]Formal definition for the permutation value function ν used is given in Definition 5 in Section 3.1

both 4 and 2 before it are smaller, hence the number 2 is produced in the dataflow graph corresponding to the input 8.

After this count is produced, it is multiplied by a factorial base, based on its position. Summing up all 4 results produced from this multiplication step gives us the permutation pattern. Input values that are more towards the left have a greater possible maximum count of values smaller and to the right of themselves, hence a factorial base is needed to generate a unique number representing the permutation of the 4 input data. This unique number can be reverse-engineered to find out the order relations among the 4 input data.

Following from this, our next step, which is what this paper primarily explores, is to come up with a suitably good hashing function to hash all the calculated permutation pattern values into hash table indices. The hash table will contain the count of number of occurrences of each permutation, and increment the corresponding count when a permutation is hashed.

Chapter 3

A Proposed Hash Function for Permutation Entropy

The main aim of this chapter is to give a detailed **proposal** of a original hash function used for hashing permutation patterns. Before plunging into its exposition, the first section gives an overview of the hashing situation.

3.1 Hashing Permutations

The permutation value function ν[1] used is formally defined[2] as follows:

[1] Introduced in Definition 1 in Section 2.1.3

[2] This permutation value function is implicitly used in Ce's FPGA implementation outlined in Section 2.4.1

Definition 5 (Permutation Value Function ν)**.**

Let $\pi \in \Pi$ be a permutation pattern of some sequence of data $(d_1, d_2, ..., d_N)$ of order N, and Π is the set of all possible permutation patterns. $\nu : \Pi \mapsto \{1, 2, ..., N!\}$ is defined as

$$\nu(\pi) = \prod_{i=0}^{N} k(N-i)!$$

where

$$k = \#\{d_j | j > i, d_j < d_i\}$$

Then the permutation value v of π is given by $v = \nu(\pi)$.

As mentioned earlier in Definition 1 in Section 2.1.3, a **relaxed** definition of ν, where ν takes in an actual sequence of data values D instead of a permutation π, is allowed and often used in this paper. For e.g., consider the data sequence $D = (2, 4, 5)$ that exhibits permutation pattern of (102):

$$\underbrace{\nu(2,4,5)}_{\text{relaxed}} = \underbrace{\nu(102)}_{\text{formal}}$$

Following from this, the range of $P(d_1, ..., d_N)$ is $\{0, 1, 2, ..., N! - 1\}$, which can be easily proved by induction. Hence the permutation value of a sequence of data of order N takes on $N!$ possible values. Now we are ready to illustrate the hashing situation, which is best done with an example.

3.1.1 An Example

Let order $N = 4$, and ν as defined in Definition 5 in Section 3.1. Assume we are given the following data stream of values:

$$...4, 8, 7, 6, 9, 1, 10, 15, 2, 17... \tag{3.1}$$

Imagine a moving window of size 4 that starts from the left and repeatedly shifts to the right by 1 data value. With every shift, this window captures a consecutive sequence of 4 data values, and this sequence's permutation value v is calculated using the ν function. The formula for PE in 2 necessitates the tabulation of the frequencies of all permutation values v.

Henceforth the objective is to keep track of the frequency of every v observed – i.e. once a certain v is observed, its count is incremented in the

hash table immediately. There are a total of 4! possible permutation values in this example. In practice, the order N tends to be large, causing the hash table size to be a fraction of N. So in this example, the table size is set to $8 < 4!$.

Using the data stream example in Example 3.1, Figure 3.1 shows a table of permutation values based on the moving window and their corresponding arbitrary hash values. The subsequent Figure 3.2 shows how the hash table looks like after hashing is performed based on Figure 3.1.

Data sequence (**brown**)	Permutation value v	Hash value $h(v)$
... **4, 8, 7, 6**, 9, 1, 10, 15, 2, 17 ...	$0 \times 0! + 1 \times 1! + 2 \times 2! + 0 \times 3!$ **= 5**	$h(5) = 1$
... 4, **8, 7, 6, 9**, 1, 10, 15, 2, 17 ...	$0 \times 0! + 0 \times 1! + 1 \times 2! + 2 \times 3!$ **= 14**	$h(14) = 3$
... 4, 8, **7, 6, 9, 1**, 10, 15, 2, 17 ...	$0 \times 0! + 1 \times 1! + 1 \times 2! + 2 \times 3!$ **= 15**	$h(15) = 6$
... 4, 8, 7, **6, 9, 1, 10**, 15, 2, 17 ...	$0 \times 0! + 0 \times 1! + 1 \times 2! + 1 \times 3!$ **= 8**	$h(8) = 4$
... 4, 8, 7, 6, **9, 1, 10, 15**, 2, 17 ...	$0 \times 0! + 0 \times 1! + 0 \times 2! + 1 \times 3!$ **= 6**	$h(6) = 2$
... 4, 8, 7, 6, 9, **1, 10, 15, 2**, 17 ...	$0 \times 0! + 1 \times 1! + 1 \times 2! + 0 \times 3!$ **= 3**	$h(3) = 2$
... 4, 8, 7, 6, 9, 1, **10, 15, 2, 17** ...	$0 \times 0! + 0 \times 1! + 1 \times 2! + 1 \times 3!$ **= 8**	$h(8) = 4$

Figure 3.1: An example of v and $h(v)$ based on moving window

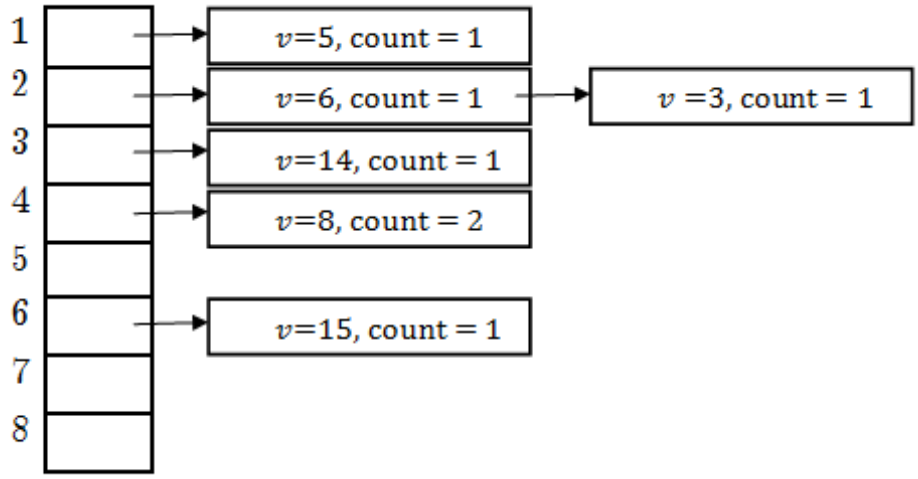

Figure 3.2: Hash Table storing counts of every v using linked lists

3.1.1.1 The Insertion Procedure

This section aims to provide the reader more insight on the procedure of inserting a permutation value in the hash table to either set its count to zero if it is not already stored in the table, or otherwise increase its count. Hence insertion of a permutation value v can entail either one of the following two procedures:

1. **Search & Insertion**
   This happens if v is not already stored in the table. Developing from Figure 3.2, let's say $v = 7, h(v) = 2$ is to be introduced into the hash table. Then as illustrated by Fig 3.3 below, insertion of $v = 7$ will involve the traversal through the list of length 2 in slot 2 to search for an existing key with $v = 7$. Since no existing key with $v = 7$ is currently stored in slot 2, then a new key-value pair with with $v = 7$, count $= 0$ will be *inserted* to the list.

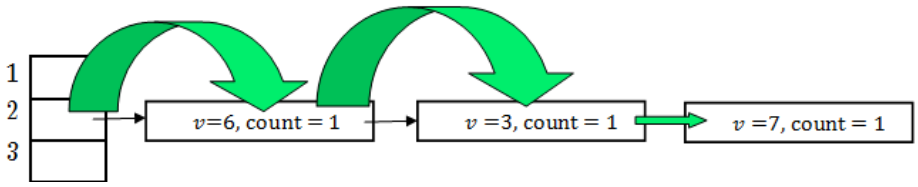

Figure 3.3: Slot 2: Traversal of list (big green arrows) followed by insertion of new key. Objects of other slots are not depicted.

2. **Search & Modification**
   This happens if v is already stored in the table. Developing from the previous Figure 3.3, let's say $v = 7, h(v) = 2$ is to be introduced *again* into the hash table. Then as illustrated by Fig 3.4 below, insertion of $v = 7$ will involve the traversal through the list of length 3 in slot 2 to search for an existing key with $v = 7$. Once the key-value pair with $v = 7$ is found, *modification* of the object will be done through incrementing its count.

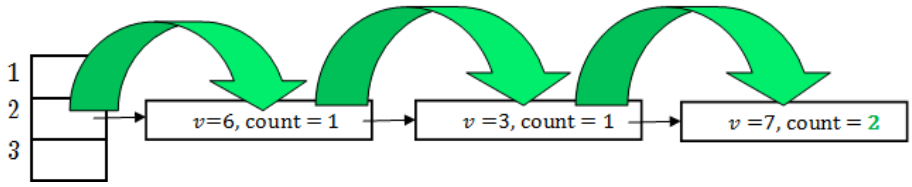

Figure 3.4: Traversal of list followed by modification of existing key.

Of course, as this is just a simple linked-list implementation, there are other ways for search and insertion, such as using a sorted linked-list for eg. Nevertheless, the main purpose of this section's discussion is to highlight the obvious disadvantage of collisions for both of the aforementioned two types of insertion procedures. In the context of hashing permutations of real-world time series, the disadvantage due to collisions is very much emphasized by the "Search and Modification" insertion procedure, and this will be discussed in greater detail in the next section.

3.2 The Need for A Good Hash Function to Minimize Collisions

If order N is large (say 100), the number of possible keys (i.e. permutation values) $= N!$ will be excessively big. Consequently due to memory constraints, the hash table size T will be some order of magnitude smaller than $N!$, and collisions will occur in various table slots.

Firstly, just like any other general hashing situation, high number of collisions implies the occurrence of a great amount of list traversal due to the "Search & Insertion" procedure outlined above. Therefore hashing speed is adversely affected if there are many collisions.

Secondly, when hashing permutations of a given large **real-world** time series, the same permutation values are likely to be **repeated again and again** throughout the time series[4]. Due to the large size of the time series

and repeated occurrences of the same permutation values, the "Search & Modification" procedure is expected to happen many times, probably much more than the "Search & Insertion" procedure. Each execution of the "Search & Modification" procedure is "unseen" because it does not cause a new collision, but it's negative impact on hashing speed is in fact commensurate with the the number of collisions.

As a result, in the context of hashing permutations of a real-world time series, the number of collisions has a manifold adverse impact on hashing speed due to the large number of occurrences of "Search & Modification" procedure. Optimal hashing speed is absolutely essential to the evaluation of the PE of a massive high-frequency time series. Therefore, there is an emphatic need for a good hash function that minimizes collisions when hashing permutations.

3.3 Motivation for the Design of a Specialized Hash Function

In general, time series are not readily predictable and the values from the data stream display apparent "randomness". Therefore, given the more or less random nature of data as input keys, any generic hash function to hash permutation values such as the Remainder Operator[3] should already perform respectably. Nevertheless, there is motivation for developing a better hash function that is specific to the purpose of hashing permutations[4].

3.3.1 Hash Function: Generic vs Dependency-specific

If input keys are seemingly random, i.e. their probability density function (PDF) follow a uniform or unknown distribution, then any well-established generic hash function(such as those outlined in Section 2.2.1) could well be already theoretically ideal. However, if there exists *dependencies* among input keys, then a generic hash function may not be ideal. For example, if we know input keys are integers ending with 7, then a function that hashes based on the last digit of an integer key would not be very smart!

Another example: for every integer key A to be hashed, there would be another integer key A' that is equal to A with digits *reversed*. Consider the hash function that multiplies all digits of a key before taking modulo table size – it would not fare very well because A and A' would hash to the same index. A good **dependency-specific** hash function would take advantage of the information about the "reverse" relationship between keys and tackle

[3]Described in Section 2.2.1

[4]The process of hashing permutations is described in Section 3.1

this dependency – i.e. by ensuring as much as possible that A and A' map to *different* table indices! The challenge is to invent such a function.

3.3.2 Motivation for a Dependency-specific Hash Function for PE

Following from the previous section, are there discernible dependencies among permutations of a data stream? If there are, it would be wise to exploit this information and incorporate it into a new dependency-specific hash function, with the objective to even out collisions as much as possible. If this information is exploited effectively, the dependency-specific hash function theoretically should outperform generic ones.

3.4 Exploiting the Dependencies

3.4.1 The Dependencies Among Neighbouring Permutation Patterns

It turns out that there exists subtle dependencies among neighbouring permutation patterns by virtue of the **repeated overlapping due to the moving window**[5]. The definition of the permutation value (Definition 5 in Section 3.1) is central to the appreciation of the these dependencies. To illustrate, we use an example, where order $N = 4$.

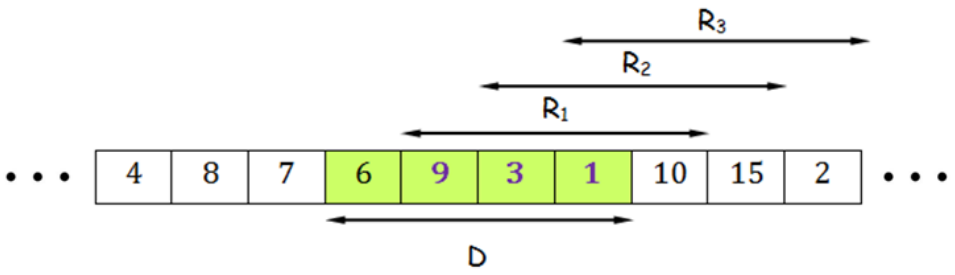

Figure 3.5: Moving 4-window of permutations in a data stream

Consider the permutation of data sequence D derived from the window corresponding to the central 4 consecutive data values in green. Does D share some information with R_1? R_2? Or R_3?

Since permutation values represent information about the relative ordering of the individual data points, D and R_1 do share some information due to the overlap of their windows. The windows of D and R_1 overlap in a

[5]The notion of the moving window is explained in Section 3.1.1

size-3 window containing the digits $(9, 3, 1)$ that are highlighted in purple in Figure 3.5. Similarly, the windows of D and R_2 overlap in $(3, 1)$.

3.4.2 Materializing these Dependencies

This overlapping information has to be materialized into actual figures to pass into our dependency-specific hash function. This can be done by considering the relative ordering of the numbers in overlapping window and deriving their permutation value. For example, the permutation value of $(9, 3, 1)$ which D and R_1 overlaps on can be derived by using Definition 5 in Section 3.1:

$$\begin{aligned}\nu(9,3,1) &= (\#\text{data to the right of and} < 9) \times 2! \\ &\quad + (\#\text{data to the right of and} < 3) \times 1! \\ &= 2 \times 2! + 1 \times 1! \\ &= 5\end{aligned}$$

Now considering the overlapping data $(3, 1)$ between D and R_2, we have $\nu(3, 1) = 1$. Then considering the left-aligned data values of D's window (D's window consists of the data 6, 9, 3, 1 in green), we have $\nu(6, 9) = 0$ and $\nu(6, 9, 3) = 3$. Of course, there is a need to calculate the actual permutation value of D as well, and we have $\nu(6, 9, 3, 1) = 17$. Then D can be represented as a sequence of permutation values $Pseq(D)$[6]:

$$\begin{aligned}Pseq(D) &= (\nu(6,9), \nu(6,9,3), \nu(6,9,3,1), \nu(9,3,1), \nu(3,1)) \\ &= (0, 3, 17, 5, 1)\end{aligned}$$

where the central number in $Pseq(D)$ represents D's own permutation value, while numbers towards the left of $Pseq(D)$ represents the permutation values of the left sub-windows of D, and numbers towards the right of $Pseq(D)$ represents the permutation values of the right sub-windows of D.

This representation of D contains vital information about permutation values of its left and right sub-windows. Our dependency-specific hash function would require **all** these information for hashing the permutation pattern of D; this is in contrast to a generic hash function using just $\nu(6, 9, 3, 1)$ to hash D's permutation pattern. How our dependency-specific hash function uses these information would be made clear as this chapter goes on.

[6]Defined formally in Definition 7 in Section 3.4.4.1

Using the same method, R_1 is represented as $(x, \nu(9,3,1), ...) = (x, 5, ...)$ and R_2 is represented as $(\nu(3,1), ...) = (1, ...)$, where x is some value inconsequential to current discussion. Notice that the right values of $Pseq(D)$, i.e. 5 and 1, occur as one of the left values of R_1 and R_2.

3.4.3 Assigning Weights To Features

3.4.3.1 The Idea

Consider permutation values as **features** in current context. For any D', the idea is to devise a dependency-specific hash function that, as much as possible, places $\nu(D')$ in an array index that has "**greater right-value features**" and "**less left-value features**".

So for example, the hash function should place $\nu(D)$ in a hash table slot that has "great 5-feature", "great 1-feature", but "little 0-feature" and "little 3-feature". Similarly, the hash function should place $\nu(R_1)$ in a slot with "little 5-feature" and $\nu(R_2)$ in a slot with "little 1-feature". In this way, R_1 and R_2 are **forced** to avoid hashing to the same slot as D with a high probability!

The key to ensuring that this idea works well lies in the method of assignment of **weights** to every possible feature w.r.t. hash table index. But what is the best way of assigning the weights?

3.4.3.2 Ideal Assignment of Weights

The situation now can be understood as: given two arbitrary sets of features A, B, the objective is to choose a hash table index $\hat{I}$ that, among all table indices, favours the features in A over the features in B the *most*. In other words, if we define the score of a table index I w.r.t. sets A, B to be

$$S(I) = \sum_{f \in A} w(I, f) - \sum_{f \in B} w(I, f),$$

where f denotes feature and $w(I, f)$ refers to the weight assigned to f w.r.t. index I, then the objective is to find an $\hat{I}$ such that $S(\hat{I})$ is the maximum over all $S(I)$.

The intuition is that for this idea to work well, $S(I)$ should yield different values for all I, so that the $\hat{I}$ would be clearly better than the rest. Furthermore, since both A and B can indeed be any set of features, it follows that for this idea to work as well as possible, the requirement is that $S(I)$ should yield different values for all I w.r.t. *any* two sets of features.

This ideal requirement thoughtfully translates to: each possible permutation value/feature should ideally be assigned different weights for different hash table indices, and each hash table index should have an equal distribution of weights w.r.t. all possible permutation values/features. An example of an ideal assignment of weights is given below for order $N = 3$ and hash table size $= 3$:

		All Permutation Values / Features					
		0	1	2	3	4	5
Table Indices	1	**1**	**1**	**2**	**2**	**3**	**3**
	2	**2**	**3**	**1**	**3**	**1**	**2**
	3	**3**	**2**	**3**	**1**	**2**	**1**

Table 3.1: Ideal assignment of weights for $N =$ table size $= 3$

In Table 3.1 above, the number of possible permutation values $= 3! = 6$. The numbers in bold represent the weight assigned to a permutation value at a particular hash table index. Table 3.1 is ideal because each column is a different permutation of weights.

3.4.3.3 Introducing the Feature-Bias Divergence

Let $w(I, v), w\colon (\mathbb{N}, \mathbb{N}_0) \mapsto \mathbb{N}_0$ be the **weight function** assigned to permutation value v at hash table index I. Using the earlier example illustrated by Figure 3.5, the requirement for $\nu(D)$ to be hashed to an index that has "great 5-feature", "great 1-feature", but "little 0-feature" and "little 3-feature" should cause D to be hashed to I such that

$$FBD(\hat{I}, D) = \sup_{I \in \{1,2,3\}} FBD(I, D) \tag{3.2}$$

where $FBD(I, D)$ is the **Feature-Bias Divergence** of I w.r.t. D defined in the context of this example as

$$FBD(I, D) = w(I, 5) + w(I, 1) - w(I, 0) - w(I, 3) \tag{3.3}$$

For any permutation value v, $FBD(I, D)$ provides information about

1. how biased the hash table index I is *towards* v if the term $w(I, v)$ has a positive coefficient in Equation 3.3, and

2. how biased the array index I is *against* P if the coefficient of the term w(I,P) is negative.

If coefficient of $w(I, v)$ equals 0 then no bias is reflected.

3.4.3.4 A proposed $w(I, v)$ function

There exists one glaring challenge in the solving the maximisation problem i.e. Equation 3.2 – it seems that there does not exist any simple formulation of $w(I, v)$! Even if $w(I, v)$ is well-defined, the calculation of $w(I, v)$ and then the subsequent solving of Equation 3.2 might well be too complex and slow.

The idea is do a *trade-off* between ideality and simplicity. This paper **proposes** a simple definition for $w(I, v)$, which can

1. achieve an assignment of weights that is near ideality, and
2. allow Equation 3.2 to be solved easily and quickly.

The proposed definition is as follows:

Definition 6 $(w(I, v))$**.**

Let the hash table size be $p-1$, where p is a prime. Let $1 \leq I \leq p-1$ represent a hash table index and v represent a permutation value. The $w(I, v)$ function is defined as

$$w(I, v) \equiv I \times (v + 1) \mod p$$

where

$$0 \leq w(I, v) \leq p - 1$$

How the above-defined $w(I, v)$ function works is best illustrated with an example. Let order $N = 4$, hash table size $= 4$ (then $4 + 1 = 5$ is prime). Then we have $w(I, v) \equiv I \times (v + 1) \mod 5$. This yields the table of weights below:

		All Permutation Values / Features								
		0	1	2	3	4	5	6	7	...
Table Indices	1	**1**	**2**	**3**	**4**	0	1	2	3	...
	2	**2**	**4**	**1**	**3**	0	2	4	1	...
	3	**3**	**1**	**4**	**2**	0	3	1	4	...
	4	**4**	**3**	**2**	**1**	0	4	3	2	...

Table 3.2: Table of weights for N = table size = 4, $w(I, v)$ as defined above

The columns corresponding to the first 4 columns all have different permutations of 1 to 4. In general, when

1. hash table size = $p - 1$ where p is prime,
2. table indices range from 1 to $p - 1$, and
3. $w(I, v)$ is as defined in Definition 6,

then it can be mathematically proven, using *Properties 1 and 2* from Section 2.2.3.2, that

1. the first $p - 1$ *columns* in the corresponding table of weights all bear *different* permutations from 1 to $p - 1$,
2. the *rows* corresponding to the first $p - 1$ columns in the corresponding table of weights all bear *different* permutations from 1 to $p - 1$, and
3. the permutations that the first $p - 1$ columns bear are repeated continually in subsequent columns, i.e. column X's permutation is the same as column Y's permutation iff $X \equiv Y \mod p$. This ensures an equal distribution of repeated permutations belonging to columns, which is desirable because balance is attained.

These 3 features of the table of weights ensure that the assignment of weights is close to ideal, with the only limitations being that there exists a few columns with all weights equal (namely those columns with $v + 1$ being a multiple of p) and that a column's permutation is repeated.

Any subsequent reference to $w(I, v)$ in this paper would follow the definition prescribed in Definition 6.

3.4.4 The Final Hashing Algorithm – An Outline

This section proposes an outline of an algorithmic formulation of a final hash function that incorporates all the ideas expounded so far in this chapter.

Let N denote the order, and D be a sequence of N data values $d_1, ..., d_N$ whose permutation is to be hashed. Consider the input key to be D. The output of the hashing algorithm is a hash table index $\hat{I}$, where count of $\nu(D)$ is to be incremented. Before outlining the steps for choosing this $\hat{I}$, a formal definition of Feature-Bias Divergence, a concept central to the algorithm, is given below.

3.4.4.1 Formal Definition of Feature-Bias Divergence

Firstly, a definition of the $Pseq$ function is needed. The $Pseq$ function has been defined informally in an example in Section 3.4.4.1. Its formal definition is as follows:

Definition 7 (Representation of D: $Pseq(D)$)**.**

D's representation as a sequence of permutation values $Pseq(D)$ is defined formally as

$$Pseq(D) = (l_1, ..., l_{N-2}, \nu(D), r_{N-2}, ..., r_1)$$

where

$$\begin{aligned} l_i &= \nu(d_1, ..., d_i), \\ r_j &= \nu(d_j, ..., d_1), \\ 1 &\leq i, j \leq N-2 \end{aligned}$$

The Feature-Bias Divergence function (FBD) has been introduced briefly in Section 3.4.3.3. The following gives a formal definition:

Definition 8 (Feature-Bias Divergence)**.**

The **Feature-Bias Divergence** FBD of a hash-table index I w.r.t. D is defined as

$$FBD(I, D) = \sum_{j=1}^{j=D-2} w(I, r_j) - \sum_{i=1}^{i=D-2} w(I, l_i)$$

where l_i and r_j are as defined above in Definition 7.

3.4.4.2 The algorithm to choose $\hat{I}$

1 Determine a suitable prime p. The exact process of selecting such a p is delineated in Chapter 4 Section 4.1.1.

2 Set hash table size as $p - 1$.

3 Determine $Pseq(D)$.

4 Find $\hat{I}$ such that $FBD(\hat{I}, D)$ is a good approximation of $\sup_{I \in \{1,2,3...,p-1\}} FBD(I, D)$.

How? We know

$$\begin{aligned} FBD(\hat{I}, D) &\sim \sup_{I \in \{1,2,3...,p-1\}} FBD(I, D) \\ &= \sup_{I \in \{1,2,3...,p-1\}} \left(\sum_{j=1}^{j=D-2} w(I, r_j) - \sum_{i=1}^{i=D-2} w(I, l_i) \right) \\ &\equiv \sup_{I \in \{1,2,3...,p-1\}} \left(I \times \underbrace{\sum_{j=1}^{j=D-2} r_j - l_j}_{z} \right) \mod p \\ &\equiv p - 1 \mod p \end{aligned}$$

Hence,

4.1 If $z = 0$, there is no way to determine an $\hat{I}$ such that $FBD(\hat{I}, D) \sim \sup_{I \in \{1,2,3...,p-1\}} FBD(I, D)$. In this case, just arbitrarily set $\hat{I}$ using the Remainder Operator hash, i.e., $\hat{I} \equiv \nu(D) \mod p - 1$.

Otherwise, find $\hat{I}_1$ such that $\hat{I}_1 \equiv (p-1)/z \mod p$. [7]

4.2 Evaluate $FBD(\hat{I}_1, D)$.

4.3 If $FBD(\hat{I}_1, D) \geq 0$, set $\hat{I} = \hat{I}_1$ and terminate. Otherwise, continue to find $\hat{I}_2$ such that $\hat{I}_2 \equiv (p-2)/z \mod p$.

4.4 If $FBD(\hat{I}_2, D) \geq 0$, set $\hat{I} = \hat{I}_2$ and terminate. Otherwise, continue the process until $\hat{I}$ is set.

In practice, the number of iterations required to determine $\hat{I}$ is small and contributes negligibly to hashing time. Proof of this is found in Section 4.1.4 on termination analysis.

3.4.5 Determinism

As stipulated in Section 2.2, a hash function has to be deterministic, i.e. two equal keys should invariably hash to the same value. The proof that the proposed hash function is deterministic is a pretty trivial one and demonstrated below.

Proof of Determinism: Since the hash function depends on all the information in $Pseq(D)$ (and not just $\nu(D)$) to produce a hash value, the onus is to prove that $Pseq(D)$ is uniquely determined by its permutation pattern. Consider two data sequences A and B of order N that exhibit the same permutation pattern π. Then any subsequence A' of A and its corresponding subsequence B' of B will exhibit the same order relations and therefore the same permutation pattern π'. It follows that $Pseq(A) = Pseq(B)$.

3.4.6 The FBD Hash

The proposed hash function shall be named ***the FBD hash***.

[7] Here "/" would require the use of the modular inverse function which has been defined in Definition 4 in Section 2.2.3.3. By Theorem 1 in the same section, we know a unique $\hat{I}_1 \equiv (p-1)/z \mod p$ exists if $z \neq 0$. The exact method to compute modular inverses w.r.t. p is given in Chapter 4 Section 4.1.2.

Chapter 4

Implementation & Performance Evaluation of the FBD Hash

The first part of this chapter gives a broad overview of the implementation of the hashing algorithm proposed in Chapter 3 (in particular Section 3.4.4.2). The second part of this chapter covers the performance evaluation of the FBD hash, which includes discussions on the test data and benchmarks, proposal of performance metrics, analysis of results, as well as limitations of the test.

4.1 Implementation

Here is a summary of the algorithm behind the FBD hash that is outlined in Section 3.4.4.2:

The FBD Hashing Algorithm.

Let N denote the order, and D be a sequence of N data values $d_1, ..., d_N$ whose permutation is to be hashed. Then the FBD hashing algorithm is as follows:

1 Determine a suitable prime p.

2 Set hash table size as $p - 1$.

3 Determine $Pseq(D)$.[a]

4 The objective is to find $\hat{I}$ such that $FBD(\hat{I}, D) = \sup_{I \in \{1,2,3...,p-1\}} FBD(I, D)$. The following procedure determines $\hat{I}$:

4.1 Set $z = \sum_{j=1}^{j=D-2} r_j - l_j$ and $i = 1$. If $z = 0$, set $\hat{I}$ such that $\hat{I} \equiv \nu(D) \mod p-1$ and terminate. Otherwise, continue to Step 4.2.

4.2 Determine $\hat{I}_i$ such that $\hat{I}_i \equiv (p - i)/z \mod p$, where $z = \sum_{j=1}^{j=D-2} r_j - l_j$.

4.3 Evaluate $FBD(\hat{I}_i, D)$.

4.4 If $FBD(\hat{I}_i, D) \geq 0$, set $\hat{I} = \hat{I}_i$ and terminate. Otherwise, increment i and start from Step 4.2 again. This process will terminate, and usually very quickly.

[a] As defined in Definition 7 in Section 3.4.4.1.

Details of the steps highlighted in red – including their implementation and termination – will be discussed in the next sections.

4.1.1 Step 1: Finding the Prime p

The total number of possible permutation values is $N!$. Simply put, if N is large, $N!$ is *very* large. Ignoring space constraints, a hash table of size $N!$ is ideal because every permutation value can be mapped to a hash slot, resulting in zero collisions and hence optimal insertion speed. However, if N is large, space-time tradeoff is an important consideration, not to mention the real possibility of insufficient allocation space for a table of size $N!$.

Hence for evaluative purposes, the aim is to set the size of the hash table to a figure in the region of $\lfloor kN \rfloor !$, where k is a fraction between 0 and 1. In this paper's implementation, k is set as 0.5. Since hash table size = $p - 1$, finding a suitable prime p is done by starting from $p = \lfloor 0.5N \rfloor !$ and incrementing p until p is a prime.

Here the Miller-Rabin test expounded in Section 2.2.3.4 is employed every time the primality of p is to be determined. 20 iterations of the Miller-Rabin test are performed in a primality test. By Theorem 3, the chance of a composite number passing the Miller-Rabin test is $\leq 2^{-40}$, which is sufficiently small. Moreover, the Bertrand's Postulate given in Theorem 4 proves the termination of the algorithm to find p before $p = 2 \times \lfloor 0.5N \rfloor !$ is reached.

4.1.2 Step 2: Determining $Pseq$

At first sight it might appear that determining $Pseq(D)$ is a computational bottleneck in terms of hashing speed. This is because determining $Pseq(D)$ involves repeatedly finding the permutation values of an increasing sequence up to a length of N, hence – using the fact that complexity of $\nu(d)$, where d has order n, is $\mathrm{O}(n^2)$ – it seems that the time complexity is $\mathrm{O}(N^3)$. A noteworthy point is that these complexity analyses are with regard to CPU implementation, Ce's FPGA implementation[1] would reduce these complexities notably, namely by a factor of N^2. Nevertheless, though a CPU implementation is used for discussion here, choice of hardware implementation is irrelevant to the main point that this section aims to bring across.

However, this is a misconception. To hash D, $\nu(D)$ must be known, so hashing time is **at least** of $\mathrm{O}(N^2)$ complexity, because #(data to the right of d_i and ¡ d_i) must be tabulated for every $i \in \{1, 2, ..., N\}$. These tabulated figures are then conveniently used in the evaluation of $\nu(d)$ where d is some right subsequence of D (in other words, d is a subsequence of D that ends with d_N). Hence no additional damage in terms of computational time is done when evaluating the permutation values of the right subsequences of D.

What about the left subsequences of D? Any left subsequence of D is a *right* subsequence of D_l where D_l is some data sequence of order N that overlaps with D on D's left. For example, the left subsequence (d_1, d_2, d_3) of D is a right subsequence of D_l where $D_l = (d_{4-N}, d_{3-N}, ..., d_{-1}, d_0, d_1, d_2, d_3)$. Since the moving window shifts from left to right, the permutation values of the left subsequences of D would already have been evaluated.

[1] Described in Section 2.4.1

Therefore determining $Pseq(D)$ takes $O(N^2)$ time, achieving the minimum bound of time complexity for hashing D.

4.1.3 Step 4.2: Determining $\hat{I}_i$

Step 4.2 states: determine $\hat{I}_i$ such that $\hat{I}_i \equiv (p-i)/z \mod p$. Of course, the unspoken, but obvious, condition here is that $1 \leq \hat{I}_i \leq p-1$ so that $\hat{I}_i$ is within the range of possible hash table indices. Consider

$$\begin{aligned} & \hat{I}_i \equiv (p-i)/z, \qquad 1 \leq \hat{I}_i \leq p-1 \\ \Leftrightarrow \quad & \hat{I}_i \equiv (p-i)z^{-1} \qquad \mod p \end{aligned}$$

where z^{-1} is the unique multiplicative inverse of z under modulo p, as defined in Definition 4, with uniqueness proclaimed in Theorem 1. Hence to determine $\hat{I}_i$, the most critical step is to find the z^{-1} under modulo p. The fastest method for finding z^{-1} is the extended Euclidean algorithm described in Section 2.2.3.3. According to Theorem 2, finding z^{-1} using takes $O(p^2)$ time. Since p has the potential to be *very* large, this time complexity certainly cannot be tolerated.

The solution is to do some preprocessing work to construct a complete table of inverses modulo p, i.e. to construct an array $InvTable$ with indices ranging from 1 to $p-1$ such that $InvTable[x]$ stores the inverse of x modulo p. In addition, a neat trick to halve the computations of the inverses is to use the extended Euclidean algorithm to evaluate the first half of $InvTable$, and then use the identity

$$(p-y)^{-1} = p - y^{-1}, \quad 1 \leq y \leq \frac{p-1}{2}$$

to complete the rest of the table.

4.1.4 Step 4.4: Termination Analysis

This section handles two aspects of termination analysis of the loop comprising Steps 4.2 to 4.4, namely proof of termination and a discussion of termination time.

4.1.4.1 Proof of Termination

The objective is to prove that the loop consisting of Steps 4.2 to 4.4 will eventually terminate.

First, define $g(x) = (p-x)z^{-1} \mod p$ such that $1 \leq x \leq p-1$. The following lemma is a crucial part of the termination proof.

Lemma.

Take any x, v s.t. $1 \leq x \leq p-1, 1 \leq v \leq N!$. Then

$$w(g(x), v) = p - w(g(p-x), v)$$

where w is the weight function as defined in Definition 6.

Proof of Lemma: Under modulo p,

$$\begin{aligned} & w\left(g(x), v\right) + w\left(g(p-x), v\right) \\ \equiv \quad & g(x) \times (v+1) + g(p-x) \times (v+1) \\ & \quad \text{using Definition 6} \\ \equiv \quad & (p-x)z^{-1}(v+1) + \left(p-(p-x)\right)z^{-1}(v+1) \\ \equiv \quad & -xz^{-1}(v+1) + xz^{-1}(v+1) \\ \equiv \quad & 0 \end{aligned} \tag{1}$$

Since $p-x, z^{-1} \not\equiv 0$, we have $g(x) \equiv (p-x)z^{-1} \not\equiv 0$. This implies $1 \leq w\left(g(x), v\right) \leq N$. Applying the same reasoning, we also conclude $1 \leq w\left(g(x), v\right) \leq N$. Collectively, these give

$$1 < w\left(g(x), v\right) + w\left(g(p-x), v\right) < 2p \tag{2}$$

Combining both (1) and (2) yields $w\left(g(x), v\right) + w\left(g(p-x), v\right) = p$, proving the lemma. Termination can now be proven using this lemma.

Theorem 5 (Termination)**.**

The loop consisting of Steps 4.2 to 4.4 will eventually terminate.

Proof of Termination: Proving that the loop consisting of Steps 4.2 to 4.4 will eventually terminate is equivalent to proving that for *any* D,

$$\exists \;\; i \text{ s.t. } 1 \leq i \leq p-1, \quad FBD\left(g(i), D\right) \geq 0 \tag{3}$$

Take any k s.t. $1 \leq k \leq p-1$. Then

$$\begin{aligned} FBD\left(g(k), D\right) &= \sum_{j=1}^{j=D-2} w\left(g(k), r_j\right) - \sum_{j=1}^{j=D-2} w\left(g(k), l_j\right) \qquad \text{using Definition 8} \\ &= \sum_{j=1}^{j=D-2} p - w\left(g(p-k), r_j\right) - \sum_{j=1}^{j=D-2} p - w\left(g(p-k), l_j\right) \\ &= -\left(\sum_{j=1}^{j=D-2} w\left(g(k), r_j\right) - \sum_{j=1}^{j=D-2} w\left(g(k), l_j\right) \right) \\ &= -FBD\left(g(p-k), D\right) \end{aligned} \tag{4}$$

(4) tells us that $FBD\left(g(k), D\right)$ and $-FBD\left(g(p-k), D\right)$ are of differing signs, thus at least one of them ≥ 0. This effectively proves (3), and we are done.

4.1.4.2 Termination Time Analysis

Theorem 6 (Probability Bound on Survival of Loop)**.**

The probability that the loop consisting of Steps 4.2 to 4.4 is still alive (i.e. has not yet terminated) after i iterations is at most $\frac{1}{2}^i$.

Proof of Probability Bound: For any D, Equation (4) implies that there exists some subset S' with exactly half the elements in $S = \{1, 2, ..., p-1\}$ such that $FBD(s, D) \geq 0$ for all $s \in S'$. Hence the probability that the loop terminates after the first iteration is exactly $\frac{1}{2}$.

Assume the loop has survived until the j^{th} iteration, $j \geq 1$. It follows that none of the elements in S' is tested in the first j iterations. Let $S'' \subset S$ be the set of untested elements such that $S'' \cap S' = \emptyset$. Then $\#S'' < \#S'$, and the probability of the loop surviving the $(j+1)^{\text{th}}$ iteration $= \frac{\#S''}{\#(S' \cup S'')} < \frac{\#S''}{\#(S'' \cup S'')} = \frac{1}{2}$, proving the theorem.

This probability bound implies that it is likely for the loop to terminate very quickly – therefore Step 4 will not have any noticeable adverse impact in terms of time on the overall hashing algorithm.

4.2 Performance Evaluation

The objectives of this section include presenting a variety of relevant data types for testing, proposing performance metrics, and providing a general

idea of how the FBD hash fares against several benchmarks. This section's performance evaluation of the FBD hash is by no means – and has no intention to be – conclusive. That said, this section does provide a blueprint for how future evaluations of the FBD hash, or in fact any hash function specific to PE, can be carried out.

4.2.1 Data Types Used

A total of 6 types of data that are suitable for the application of PE are chosen as inputs for hashing:

1. **EEG data of an epilepsy patient from the Bern-Barcelona Database**[20]
   These data are used to identify the parts of the patient's brain that should be surgically removed in order to treat his seizures. As described in Section 2.1.4.2, the use of PE is extremely relevant to this purpose.

2. **High-frequency EEG signals from a rat**[21]
   Data represent EEG recordings over 5 sec at the frontal cortex of a male adult rat. Not only is PE adept in characterizing brain patterns derived from EEG data, but its superb computational efficiency also contributes to its effectiveness in analysing high-frequency clinical data, as little time is allowed for preprocessing and fine-tuning of parameters. These two advantages account for the selection of this data set.

3. **High-frequency SPDR S&P 500 Tick Data from QuantQuote**[22]
   Closing prices are used. In the first two chapters of this paper, the use of PE has been suggested to be very suitable to high-frequency financial data analysis.

4. **High-frequency tick data for forex pair EUR/USD**[23]
   Closing prices are used.

5. **Daily closing prices for Dow Jones**[24]
   The use of PE also stretches to non high-frequency data, therefore it would also be interesting to get a glimpse of how hash functions perform for such data.

6. **Randomly generated data**
   The rationale behind the FBD hash is not restricted to any data type. Hence a random data set should also be used.

4.2.2 Proposed Performance Metrics

There are two general themes that govern performance metrics of hash functions – (1) some measure of number of collisions and (2) direct measure of hashing speed. In this performance evaluation, metrics measuring collisions, and *not* hashing speed, are used. The next section explains why.

4.2.2.1 Metrics: Collisions vs Empirical Speed

Though the final implementation of the entire PE calculation procedure is projected to be based on Ce's FPGA implementation[2], the development of a good hash function – the primary focus of this paper – is done on the CPU. The FPGA implementation of calculating sequential permutation values of a large data stream is much faster than a CPU implementation. This is because calculating just one permutation value on CPU takes at least O($NlogN$), making the calculation all permutation values of a very long data series infeasible, especially when we want to test for large N. Therefore, in the current situation where CPU is used, there is no ready access to extremely large sets of permutation values of a broad range of real-world time series.

A real-world time series is very likely to exhibit the **pattern-repeating phenomenon**. This means that the same permutation values are likely to be repeated again and again over time[4]. As explained in Section 3.2, the number of collisions will have a manifold and drastic negative impact on hashing speed due to the presence of perpetually repeating patterns of a given real-world time series. This pattern-repeating phenomena can only be adequately captured when hashing a very large sequence of calculated permutation values, which a CPU implementation is incapable of offering, especially when N is large. Consequently, in the absence of massive sequences of permutation values over a wide range of N to work with, hashing speed as a metric is a limited measure of hash function quality.

On the other hand, number of collisions is asserted to have a particularly dominant impact on hashing speed in the context of hashing permutations of real-world time series (see Section 3.2 for further elaboration). It follows that suitable performance metrics that measure collisions instead of hashing speed should be employed.

4.2.2.2 Proposed Metrics

In this performance evaluation, 3 such carefully chosen/improvised metrics are used. These metrics quantify collisions; the first metric will be the

[2]Described in Section 2.5

primary metric used for evaluation, while the second and third metrics will only be used as supplementary metrics.

1. **Metric from the Red Dragon Book**
   The following formula, based on the Red Dragon Book[18], will be used as the primary metric to evaluate hash function quality in this test:

$$\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)} - 1$$

   where m is the number of buckets, n is the total number of key-value pairs, and b_j is the number of key-value pairs in the j^{th} bucket. Under a hash function h, the number of buckets that should be searched through before locating the correct key is given by the sum of $\frac{b_j(b_j+1)}{2}$. $\frac{n}{2m}(n+2m-1)$ gives the expected number of visited key-value pairs for an ideal function that hashes totally randomly. It follows that if h is ideal, the metric will yield 0. The higher the value of the metric, the poorer h is.

2. **"Applied" Kullback-Leibler Divergence**

Definition 9 (Application of the Kullback-Leibler Divergence)**.**

Also known as relative entropy, or **K-L divergence** for short, it is a measure of the distance between two probability distributions on a random variable. Formally, given two probability distributions $p(x)$ and $q(x)$ over a discrete random variable X, the K-L divergence given by $D(p||q)$ is defined as

$$D(p||q) = \sum_{x \in X} p(x) \log \frac{p(x)}{q(x)}$$

An ideal hash function assumes Simple Uniform Hashing, i.e. it will evenly distribute items into the slots of a hash table. Moreover, each item to be hashed has an *equal probability* of being placed into a slot, regardless of the other elements being placed. Hence under ideality, the hash table index X a key is hashed to follows a *uniform* distribution.

Just like the Red Dragon metric, a value of 0 for this metric implies ideality, and the greater the metric, the less ideal it is.

3. **Modified Variance**
   The following formula, using the concept of variance, is proposed by this paper as a measure of hash function quality based on collisions:

$$\sum_{j=0}^{m-1} \frac{b_j^2}{(n/m)^2 m} - 1$$

   where m is the number of buckets, n is the total number of key-value pairs, and b_j is the number of key-value pairs in the j^{th} bucket. Under a hash function h, the number of buckets that should be searched through before locating the correct key is proportional to the sum of b_j^2. The denominator $(n/m)^2 m$ gives the number of visited key-value pairs for an ideal hash function that distributes keys uniformly. Note that the definition of ideality here is slightly different from the definition used in the Red Dragon metric!

4.2.2.3 Novelty of Metrics

Why are the last two proposed metrics only intended to be supplemental? The Red Dragon metric is a well-established, tried and tested metric for hash function quality. However, the use of the last two metrics for hash function quality is proposed by this paper. The concept of K-L divergence is a well-known one, but its application to measuring hash function quality is not yet documented. The third metric is also an original idea. As a result, the use of the "Applied" K-L divergence and Modified Variance as hash performance metrics has a dual purpose: (1) to reinforce the credibility of the Red Dragon metric, and (2) to test their own suitability as hash performance measures. If they correlate highly to the established Red Dragon metric, it will lend support to their future use as hash metrics!

4.2.3 Benchmarks

A total of 5 hash functions will be used for testing. First of all of course is *the FBD hash*; the remaining four that are functioning as benchmarks are namely "The Remainder Operator", "Additive Hash", "Bernstein's Hash", and "Robert Jenkins' Hash Function", which are all described in Section 2.2.1.

4.2.4 Hypothesis

As the FBD hash is set up such that it specifically arrests the subtle relationships among adjacent permutation patterns and attempts to a certain extent to hash them (if distinct) to different slots[3], its performance should be consistently higher than the average benchmark performance.

[3]The discussion in Section 3.3.1 provides greater elaboration on this.

4.2.5 Results & Discussion

Due to integer limits, values from 6 to 12 are tested for N = order. All relevant results are illustrated and discussed in this section.

4.2.5.1 Performance Gauge of the FBD Hash

The charts below present an overview of the performance of the tested hash functions. Only the primary metric, the Red Dragon metric, is considered here. "FBD" stands for Feature-Bias Divergence and is used to represent the FBD hash, while "Remainder", "Additive", "Bernstein", and "Jenkins" represent each of the benchmark hash functions. Also, the metric scores in this section refer to the *average* metric score over the values 6 to 12 for N.

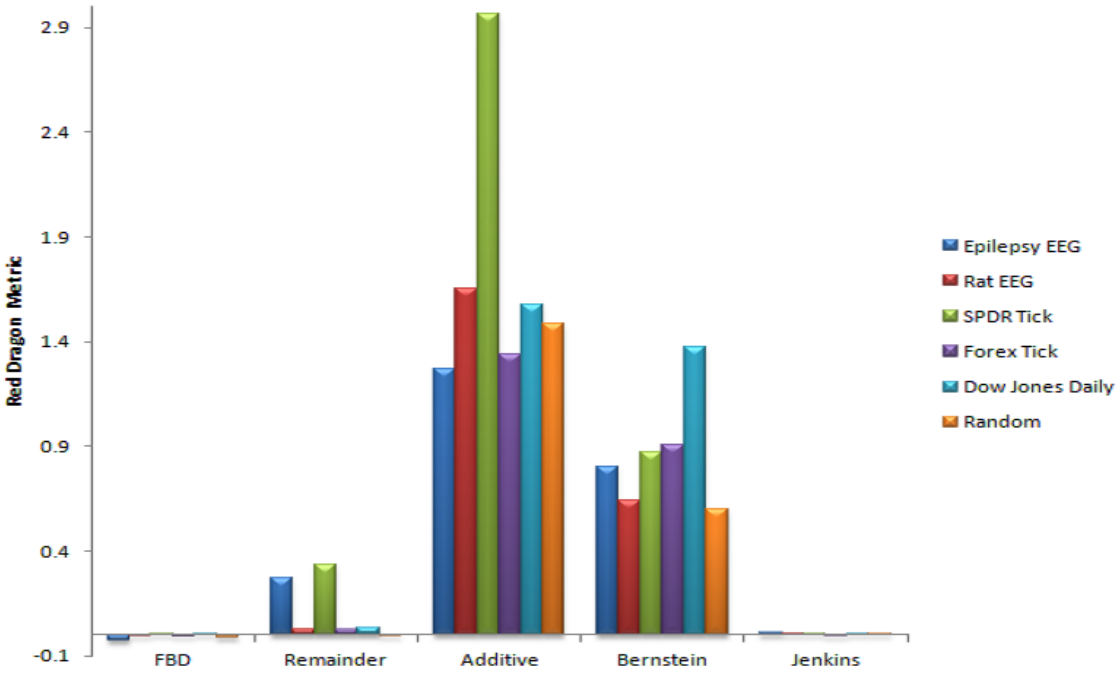

Figure 4.1: **Graph of the performance of the hash functions w.r.t. various tested data**

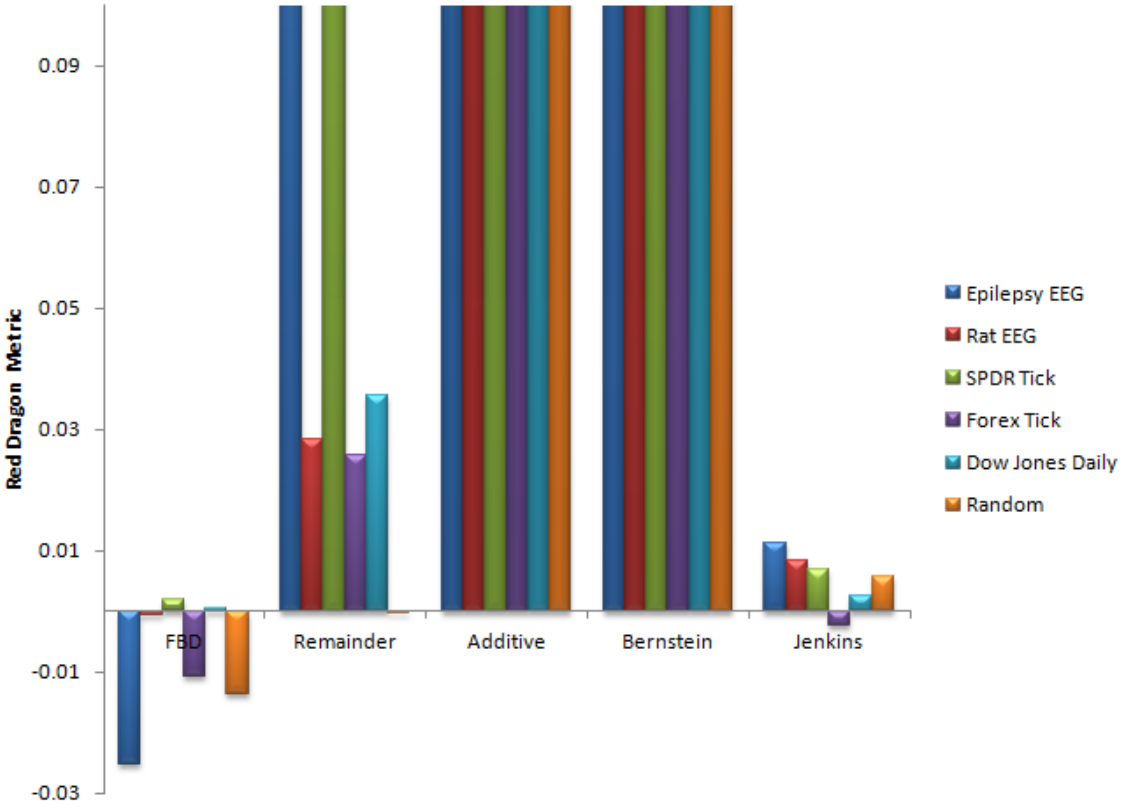

Figure 4.2: **Graph of the performance of the hash functions w.r.t. various tested data.** This graph is essentially the same as Figure 4.1, except that the metric range is reconfigured to enhance visibility of the metric scores of FBD and Jenkins.

How should the metric scores be interpreted? The only conclusion that can be made when looking at the scores is – the lower the metric score, the better the hash function quality (implying less collisions and better hashing speed). According to the Red Dragon Book[18], a score of < 0.5 indicate good performance. It is key to note that there is no clear principle that tells us how to interpret the magnitude of the disparity between two scores. For example, it cannot be said that a hash function with a score of 2 hashes a data series twice as fast as another hash function with score 1.

Looking at Fig 4.1, it is clear that performances of the hash functions are quite consistent over data types. This suggests that the performance of a hash function for permutations will not be very much affected by data type, be it random, high-frequency, financial, or biomedical. We can certainly deduce from Fig 4.1 that the Additive hash and Bernstein's hash are wide off the mark in the context of hashing permutations.

In Fig 4.2, it can be observed that FBD's metric score is dominantly negative. An "ideal" function that hashes randomly will give a score of 0, hence a negative value suggests that FBD hashes *even better* than randomly. Indeed, the objective of FBD is to forcefully attempt to hash given permutations into distinct slots, and the negative values could have reflected this endeavour. Hence this figure certainly suggests good performance of FBD.

The next charts show how the FBD hash fares against the average benchmark hash function:

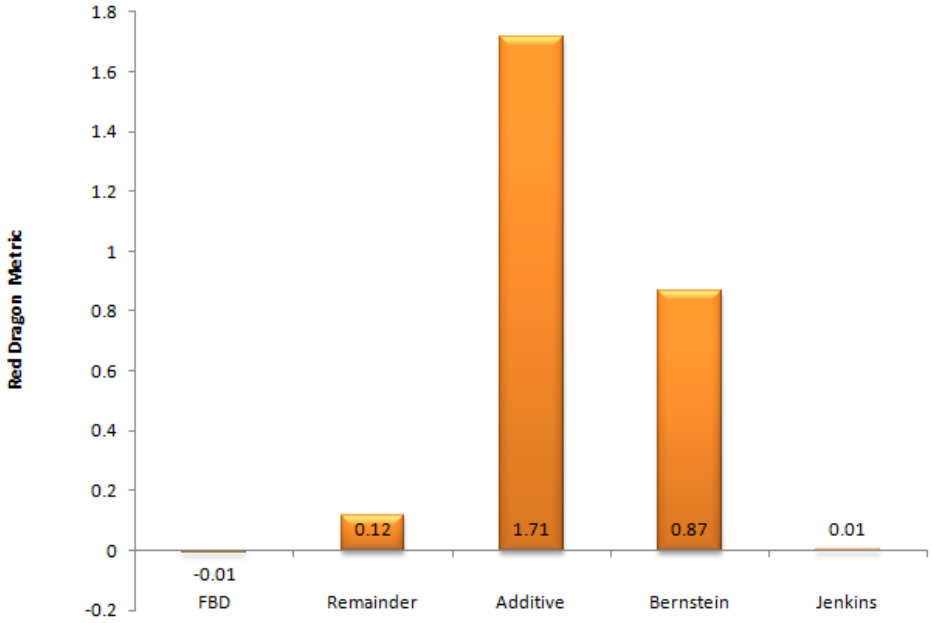

Figure 4.3: **Graph of the performance of the hash functions across all tested data.**

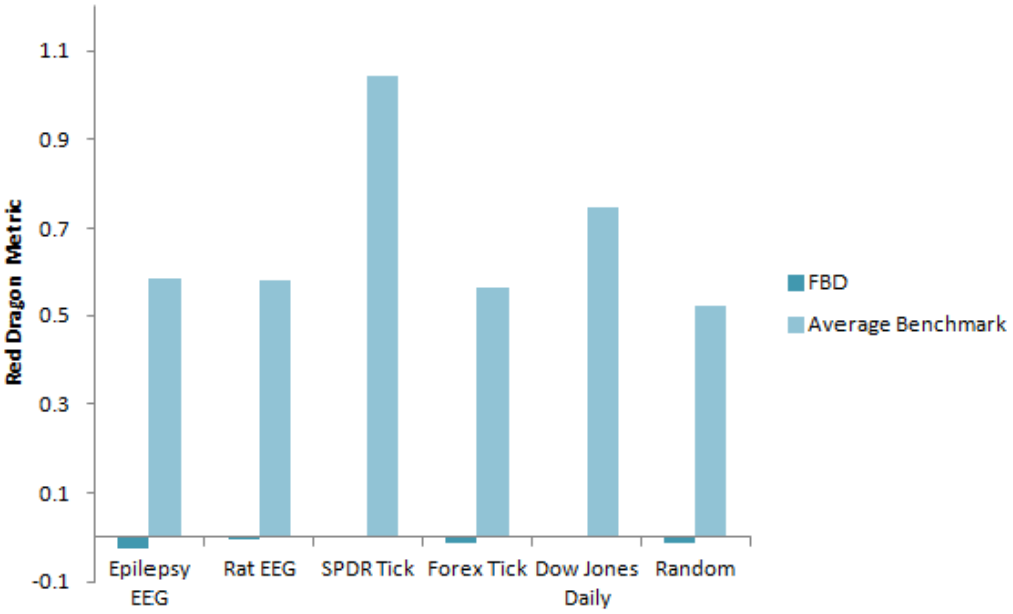

Figure 4.4: **Comparison between FBD's performance and the average benchmark performance w.r.t. each of the tested data.** The average benchmark performance is calculated by taking the average of the performances of the benchmark hash functions.

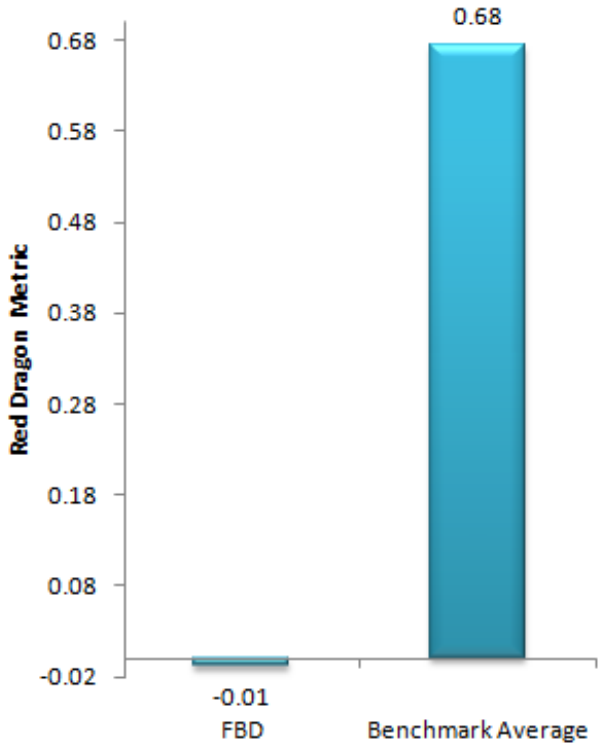

Figure 4.5: **Comparison between FBD's performance and the average benchmark performance across all tested data.**

Fig 4.3 shows a clear illustration of the overall performances of the tested hash functions. FBD gives the best performance of -0.01, with Jenkins also performing very well at 0.01. The Remainder hash function with the score of $0.12 < 0.5$ is deemed to have performed decently too. Fig 4.4 supports the hypothesis that FBD should perform consistently better than the average benchmark hash function. The performances across all tested data and order N are summarized by Fig 4.5.

The following table shows how the various hash functions fare under the "Applied" K-L divergence metric and the Modified Variance metric compared to the Red Dragon metric:

	Modified Variance	Applied K-L	Red Dragon
FBD	1.07	0.06	-0.01
Remainder	1.10	0.07	0.00
Jenkins	1.10	0.07	0.01
Bernstein	4.74	0.16	0.60
Additive	8.65	0.33	1.49

Table 4.1: **Table of hash function performances under various metrics.** The benchmark metric is the Red Dragon metric. For all 3 metrics, the lower the score, the better the hash function performance.

Notice that each of the 3 metric columns of Table 4.1 contains an increasing sequence of scores. This shows that the performance ranking of the hash functions is consistent under all 3 metrics! All 3 metrics agree that FBD ranks first, followed by the Remainder hash, Jenkins, Bernstein, and lastly Additive. The consistency among all 3 metrics reinforces the credibility of the Red Dragon metric and the analyses based on it. Also, the validities of the uses of the "Applied" K-L divergence and Modified Variance as hash performance metrics are supported based on this evidence. The following chart provides a graphic illustration of hash performances under the 3 metrics:

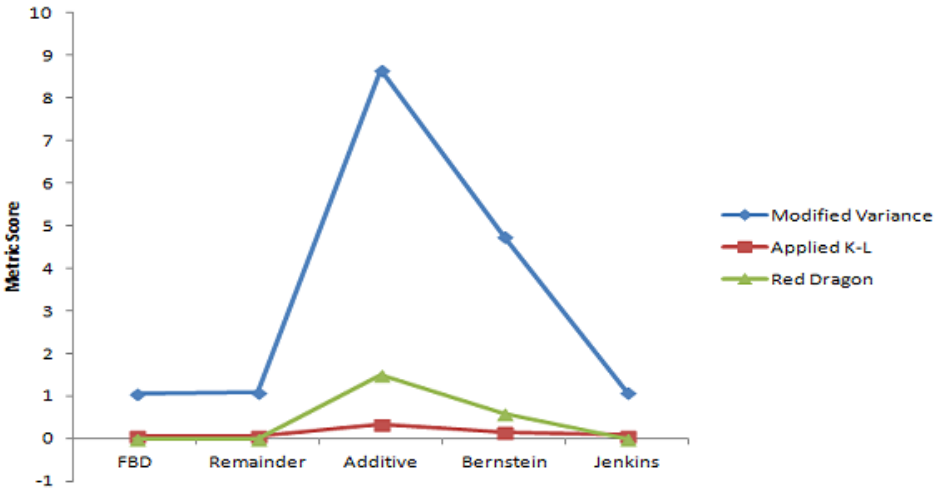

Figure 4.6: **Chart of hash function performances under various metrics.** The benchmark metric is the Red Dragon metric. For all 3 metrics, the lower the score, the better the hash function performance.

4.2.5.2 Variation of Hash Performance with N

As N increases, the hash table size increases factorially[4]. When table size is large, given the same total number of inserted keys, the distribution of number of collisions per hash slot is more likely to diverge from the uniform distribution (as compared to a small hash table). Therefore, it is quite likely for a hash function that is not well-suited for hashing permutations to dip drastically in performance as N increases.

We do not want a large value of N to seriously compromise hash performance. As a result, it is important to get a feel of how FBD, as compared to other benchmark hashes, performs as N increases from 6 to 12. The charts below provide an insight of how the performance of the tested hash functions vary with increasing N. Again, only the primary metric, the Red Dragon metric, is considered here. Also, in this section, the metric score for each $6 \leq N \leq 12$ w.r.t. hash function H indicates the *average* metric score under H over all tested data.

[4]Recall that the hashing algorithm of FBD sets hash table size = p - 1, where p is smallest prime ¿ $\lfloor 0.5N \rfloor$!

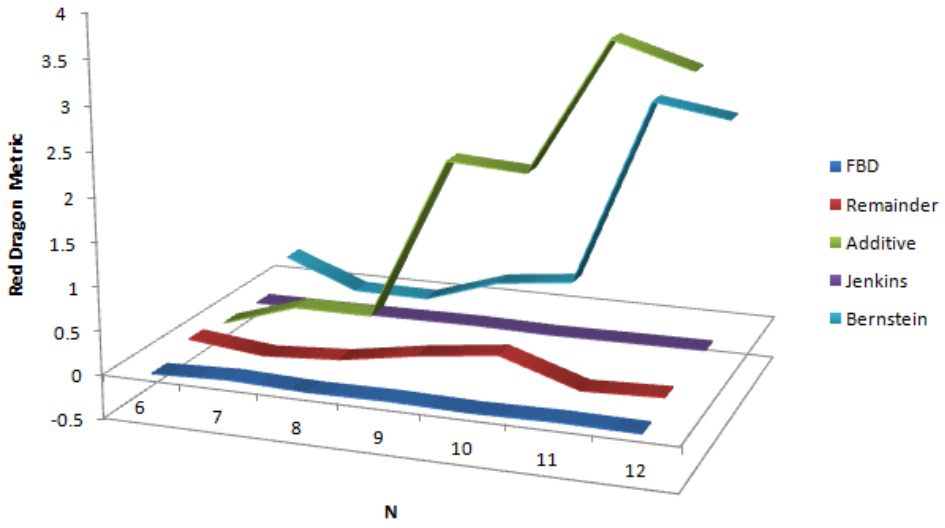

Figure 4.7: **Graph of the performances of the hash functions against increasing** N

.

Looking at Fig 4.7, it is clear that the additive and Bernstein's hashes exhibit, to some extent, performance dips due to increased N (the higher the metric score, the worse the performance). On the other hand, the Remainder, Jenkins and FBD hashes exhibit relatively stable performances over the range of N. This suggests that these three hashes are resilient to increases in N, with Jenkins and FBD the top two performers among the trio.

The close-up chart below shows that the performances of Jenkins and FBD remain rather consistent over $6 \leq N \leq 12$, with FBD slightly outperforming Jenkins:

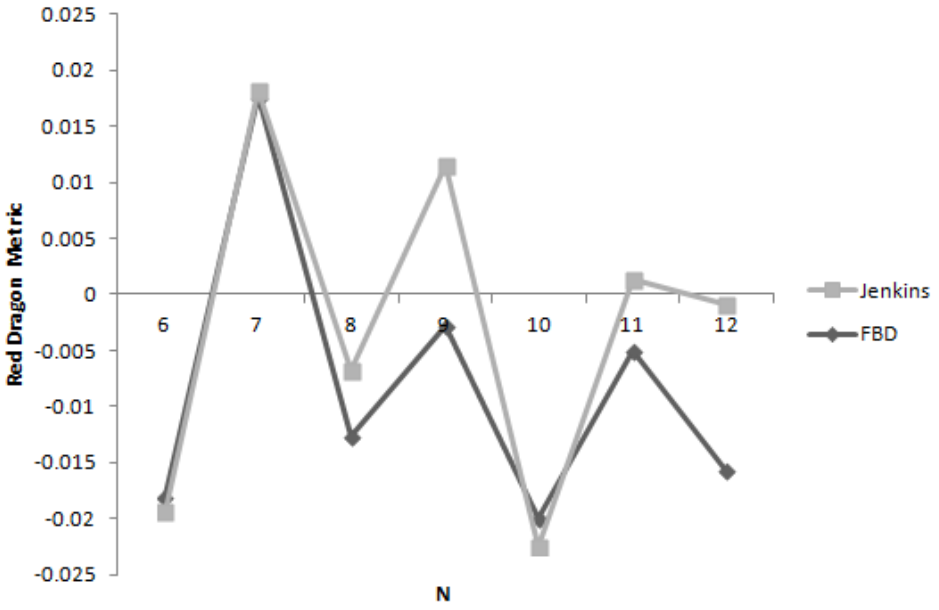

Figure 4.8: **Graph of the performances of FBD and Jenkins against increasing N.** This offers a close-up view of the performances of FBD and Jenkins against N.

4.2.5.3 Summary

Here is a summary of points that are suggested by the test results:

- The performance of a hash function for permutations is not significantly affected by data type, be it random, high-frequency, financial, or biomedical.
- The Additive hash and Bernstein's hash are not suitable for hashing permutations.
- Overall, FBD outperforms all the benchmark hashes under all 3 performance metrics, with Jenkins a close second. The results also agree with the hypothesis that FBD should perform consistently better than the average benchmark hash function.
- According to the Red Dragon metric, FBD hashes even better than randomly. The objective of FBD is to forcefully attempt to hash given permutations into distinct slots, and the test results reflected this endeavour.
- The Additive hash and Bernstein's hash experience drastic performance dips as N increases. On the other hand, the Remainder, Jenkins

and FBD hashes are resilient to increases in N, with FBD giving the most consistent performance over $6 \leq N \leq 12$ followed by Jenkins.

- While the Red Dragon metric is well-established, the "Applied" K-L divergence and Modified Variance metrics are originally conceived. Test results demonstrate that the performance rankings of the hash functions are consistent under all these 3 metrics. This consistency
  - reinforces the credibility of the Red Dragon metric and the analyses based on it, and
  - lends support to the validities of the uses of the proposed "Applied" K-L divergence and Modified Variance as hash performance metrics.

To round off, as mentioned earlier in this chapter, it is important to note that this performance evaluation of the FBD hash is by no means, and has no intention to be, conclusive. That said, this section does provide a blueprint for how future evaluations of hash function quality specific to PE can be carried out.

4.2.5.4 Limitations of the Metrics

As already pointed out, though the metrics used provide conclusive comparisons of hash performances (a lower metric score implies less collisions and faster hashing speed), there is no clear principle that tells us how to interpret the size of the disparity between two scores. For example, it cannot be said that a hash function with a score of 2 hashes a data series twice as fast as another hash function with score 1.

And there is good reason for this too. As described in Section 3.2, in the context of hashing permutations of a real-world time series, the pattern-repeating phenomenon will result in many hash insertions that go through the "Search & Modification" procedure. These "insertions" do not add new permutations to the hash table but increment the count of existing permutations. They collectively take significant time if there are many collisions in the hash table, yet do not add to the number of collisions. This means that due to the pattern-repeating phenomenon of a real-world time series, it is hard to quantify how collisions affect speed.

That said, it is also precisely due to this pattern-repeating phenomenon that we know for a fact that the number of collisions has a drastic impact on hashing speed. Therefore, the test results which deem FBD as the best in terms of collisions certainly suggest that FBD outperforms all other tested hash functions in terms of speed as well.

Nevertheless, it will be useful to also benchmark the performance of FBD in terms of actual speed measures, so that the improvements in speed due to FBD can be gauged more precisely. As mentioned in Section 4.2.2.1, this is not done in this chapter due to lack of ready access to extremely large sets of permutation values of a broad range of real-world time series. Ce currently has a very fast FPGA implementation to calculate permutation values of large data series. Collaborating with him can be a possible future extension to FBD's performance evaluation.

Chapter 5

Hardware Acceleration of the FBD Hash using CUDA

After the entire sequence of permutation values of a given data stream is computed on FPGA, what's next? There are basically 3 options to be considered: (1) continue to compute their hash values in FPGA, (2) transfer the permutation values to CPU and perform hashing in CPU, or (3) transfer the permutation values to GPU and perform hashing in GPU. The 3rd option is inspired from the idea that the parallelization offered by GPU can be exploited by the large sequence of permutation values to be hashed.

5.1 Infeasibility of Hashing in FPGA

Before we consider the feasibility of performing hash value computations in FPGA, here are some important background information regarding Ce's FPGA-based permutation encoder:

- **Hardware platform**: Maxeler MAX3 card with a Xilinx Vertix-6 SX475T FPGA
- **Performance**: Able to produce 6×10^8 permutation numbers per second
- **Resource consumption**: LUT – $\sim$ 20%, BRAM – $<$ 10%, DSP – $<$ 10%
- **Bandwidth consumption**: Memory to FPGA – $<$ 30%, FPGA to CPU – 95%
- **Performance bottleneck**: Bandwidth from FPGA to CPU
- **Performance requirement for hashing**: 6×10^8 counting operations per second

As can be seen from the information above, the bandwidth consumption due to transfer of permutation values from FPGA to CPU is extremely high (95%) and is effectively the performance bottleneck. If computation of hash values is to be done in FPGA, assuming we do not build the hash table in FPGA, we have to transfer the hash values along with the encoded permutation values from FPGA to CPU. This will further increase the FPGA-CPU bandwidth consumption which is already the performance bottleneck. If we build the hash table in FPGA, random memory over a large memory space is needed – for example, when order $N = 10,500$ accelerator cards are needed. Latency of memory makes it impossible to design a fully-pipelined architecture. All in all, it is **not** a good idea to perform hash value computations on the permutation values in FPGA.

5.2 Hashing – CPU vs GPU

Since it is decided that hashing should not be done in FPGA, the CPU will receive only the permutation values from the FPGA. Now the CPU has a large sequence of permutation values, and has to convert this sequence into a final PE value. Computation of the FBD hash values of these numerous permutation values can likely be accelerated by GPU through parallelization. However, once the computation of the hash values is performed in GPU, the dilemma arises: should we perform hash table insertions in GPU or transfer these hash values back to CPU and create the hash table in CPU instead?

This dilemma stems from the fact that multithreaded accesses to a hash table in GPU require some sort of synchronization mechanism, which might cause hash table insertions to be slower in GPU than in CPU. Still, transferring the whole sequence of hash values from GPU back to CPU would take considerable time too, so perhaps just continuing to perform hash table insertions followed by computation of the final PE in GPU would be more desirable.

Bearing these considerations in mind, these are the 2 plausible options to calculate the final PE from a given large sequence of permutation values:

1. Transfer the permutation values from CPU to GPU, and subsequently compute the FBD hash values of the permutation values in parallel in GPU **(GPU Stage 1)**. Create a hash table in GPU, and insert these permutation values into this hash table based on their computed hash values **(GPU Stage 2)**. Lastly, unravel the completed hash table and calculate the PE based on the frequencies of the stored permutation values **(GPU Stage 3)**.

2. Perform GPU Stage 1 as in the second option. Then transfer the hash values from GPU back to CPU and insert the permutation values, based on these hash values, into a hash table in CPU **(CPU Stage 2)**. Finally compute the PE in CPU based on the completed hash table **(CPU Stage 3)**.

To decide between the 2 options, it is key to compare the combined execution of GPU Stages 2 and 3 against the combined execution of CPU Stages 2 and 3.

5.3 GPU Stage 1: Parallelizing Hash Value Computations

5.3.1 Implementation

Given a large sequence of permutation values, we have to compute the FBD hash value of every permutation value in the sequence. The core idea behind GPU Stage 1's implementation is simple: transfer the permutation values from CPU to GPU, and then have each thread in GPU perform the FBD hashing algorithm on a permutation value for all permutation values in the sequence.

However, the FBD hashing algorithm cannot be carried out exactly as outlined in Section 4.1. This is because the evaluation of $Pseq(D)$ depends on the knowledge of D; however, in this case, only $\nu(D)$ is known. Therefore a *Modified* FBD hashing algorithm, which involves some sort of reverse-engineering to derive $Pseq(D)$, is performed instead. See diagram below:

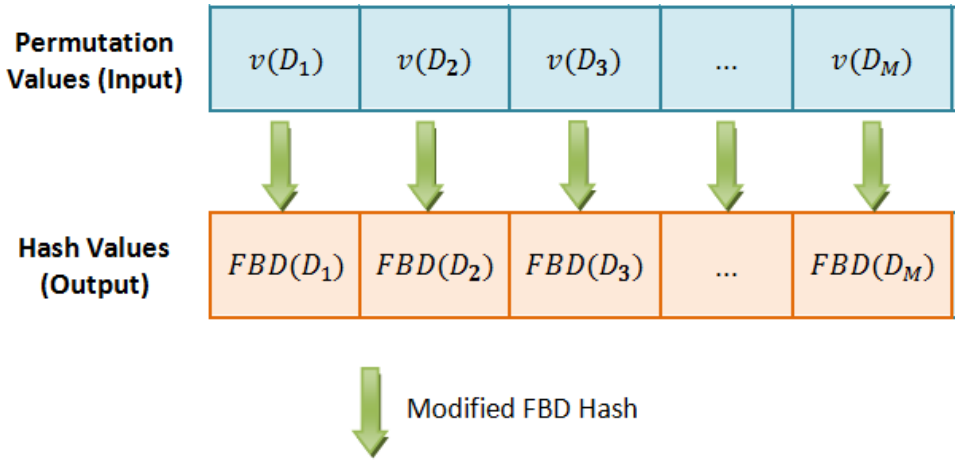

Figure 5.1: Each concurrent thread performs the Modified FBD hash on a permutation value to produce the resultant FBD hash value.

5.3.1.1 The Modified FBD Hash

$Pseq(D)$ is made up of $\nu(D)$, as well as the permutation values of the left and right subsequences of D. To illustrate, let D be $(d_1, d_2, ..., d_N)$. Using Definition 7 in Section 3.4.4.1, we need to evaluate

$$Pseq(D) = (l_1, ..., l_{N-2}, \nu(D), r_{N-2}, ..., r_1),$$

where $l_i = \nu(d_1, ..., d_i), r_j = \nu(d_j, ..., d_1), 1 \leq i, j \leq N-2$. The rest of this section describes how to evaluate $\nu(r_j)$ and $\nu(l_i)$.

Evaluating $\nu(r_j)$: By Definition 5 in Section 3.1, we have

$$\nu(D) = \prod_{i=0}^{N} k(N-i)!,$$

where $k = \#\{d_j | j > i, d_j < d_i\}$. Simply put, $\nu(D)$ is equal to the number of smaller (or equal) data values on the right of each data value multiplied by a factorial base. Calculating $\nu(r_j)$ for all $1 \leq j \leq N-2$ will simply require converting $\nu(D)$ into factorial representation. This can be done by dividing $\nu(D)$ repeatedly by a decreasing sequence of factorials, taking the remainder as digits and continuing with the integer quotient until this quotient becomes 0. With each division, the remainder gives the permutation value of a right subsequence.

To illustrate, take for example $\nu(D) = 500$. 500 can be converted into factorial representation by these successive divisions, giving $\nu(r_j)$ for all $1 \leq j \leq N-2$ along the way:

1. $500/5! = 4$ with remainder $20 \Rightarrow \nu(r_4) = 20$
2. $20/4! = 0$ with remainder $20 \Rightarrow \nu(r_3) = 20$
3. $20/3! = 3$ with remainder $2 \Rightarrow \nu(r_2) = 2$
4. $2/2! = 1$ with remainder $0 \Rightarrow \nu(r_1) = 0$

We have derived $500 = 4 \times 5! + 0 \times 4! + 3 \times 3! + 1 \times 2! + 0 \times 1! + 0 \times 0!$. This final factorial representation is unnecessary though, since we already have obtained the permutation values of all r_j in the process.

Evaluating $\nu(l_i)$: Besides evaluating $\nu(r_j)$, we also have to evaluate $\nu(l_i)$ for all $1 \leq i \leq N-2$ to determine $Pseq(D)$. However, there is no straightforward method to directly evaluate $\nu(l_i)$ based on $Pseq(D)$. This is because unlike right subsequences, permutation values of the left subsequences of D are not related to the remainders of factorial divisions of $\nu(D)$.

Fortunately, there is a simple trick to find $\nu(l_i)$. Notice that any left subsequence of D is a *right* subsequence of D_l where D_l is some data sequence of order N that overlaps with D on D's left. For example, the left subsequence (d_1, d_2, d_3) of D is a right subsequence of D_l where $D_l = (d_{4-N}, d_{3-N}, ..., d_{-1}, d_0, d_1, d_2, d_3)$.

To illustrate, let's say we are given a sequence of permutation values $(P_1, P_2, ..., P_M)$ produced by a moving window from left to right. By evaluating the permutation values of the right subsequences corresponding to every of $P_1, P_2, ..., P_M$, we would have *already* evaluated the permutation values of all left subsequences corresponding to $P_N, P_{N+1}, ..., P_M$! Therefore, the trick is "not" to find $\nu(l_i)$!

How about $P_1, P_2, ...P_{N-1}$ then? Since M is very large compared to N, not hashing $P_1, ..., P_N$ has a negligible effect on the final PE value; so it is okay to neglect the first $N-1$ given permutation values.

5.3.2 Speed Evaluation

We want to know how much faster it is to perform GPU Stage 1 (which exploits GPU's parallelism) than to simply compute the hash values sequentially in CPU. To test for the magnitude of acceleration when GPU Stage 1 is performed instead of the sequential computation of hash values in CPU, randomly generated input data of sizes 10000, 100000, 1000000, 10000000 are used. These input data are permutation values that are based on order = 6, which means that the generated permutation values range from 0 to $6! - 1$. Using an Intel Core i7 with an NVIDIA's GeForce GTX TITAN Black, these are the results:

Size of input	CPU time (ms)	GPU time (ms)	GPU speedup
10000	10	0.4	**25**
100000	40	1	**40**
1000000	320	4.7	**68**
10000000	3190	18.1	**177**

Table 5.1: **GPU speedup for different input sizes.** The CPU and GPU take in an array of permutation values, and produce an array of corresponding FBD hash values.

As can be seen from the above table, when size of input is 10 thousand, the GPU computes the FBD hash values 25x more quickly than the CPU. When the size of input reaches 10 million, the GPU computes the FBD hash values 177x more quickly! Table 5.1 suggests that within the range of 10 thousand to 10 million, the larger the input size, the more fully the GPU can exploit the parallelization of computation of FBD hash values. Therefore, the execution of GPU Stage 1 is much faster than the sequential computation of the FBD hash values in CPU.

5.4 GPU Stage 2: Inserting into a Multithreaded Hash Table

5.4.1 Implementation

Now we have an entire sequence of permutation values $\{P_i\}$ and another sequence of their corresponding hash values $\{H_i\}$. Given these two sequences, we have to insert these permutation values into a hash table built in GPU. Each thread will take care of a (P_i, H_i) pair, meaning that it will insert P_i into the hash table bucket H_i. The global hash table and the permutation objects that it stores are defined as follows:

```
struct Permutation {
       unsigned int   v;
       unsigned int   count;
       Permutation    *next;
};

struct HashTable {
       unsigned int size;
       Permutation  **buckets;
       Permutation  *permutationObjects;
};
```

Each `Permutation` object is recognized uniquely by its permutation value `v`, and its `count` is the number of times its `v` has been hashed. In a bucket, each `Permutation` is connected to another `Permutation` through `next` in the form of a linked list, which terminates at `NULL`.

The `buckets` of the `HashTable` is an array of linked lists of `Permutations`. As it is very inefficient to allocate memory when the code is already running in GPU, sufficient memory for all possible `Permutation` initializations are allocated when the code first runs in CPU. The total number of possible `Permutation` initializations is equal to the size of the input array of permutation values; this maximum occurs when all given permutation values are distinct. The relevant code is as follows:

```
const int noOfPossiblePermutations = 1024; //can be anything
    depending on number of given permutation values

void initializeHashTableinGPU(void) {
        HashTable d_hashTable;
        cudaMalloc((void**)&d_hashTable.permutationObjects,
            noOfPossiblePermutations * sizeof(Permutation)) ;
...
}
```

5.4.1.1 A Locking Mechanism for Hash Table Insertions

If two threads insert into different hash buckets, all is fine. However, the problem lies if two concurrent threads insert into the same hash bucket. As explained in Section 3.1.1.1, depending on the state of the hash table, there are two possible insertion procedures – "Search & Insertion" or "Search & Modification". We highlight one example where a race condition could occur due to writing to the same hash bucket. Let's say this is the current state of the first two buckets of the hash table:

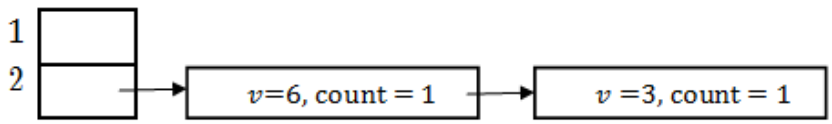

Figure 5.2: Example – Current State of Hash Table

Then assume now there are 2 concurrent threads each attempting to add the permutation value 7 into bucket 2. The expected subsequent state should be:

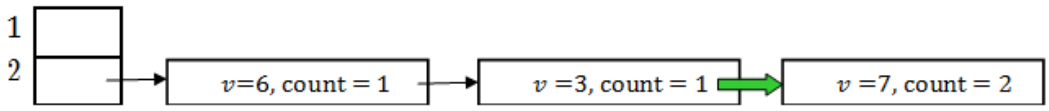

Figure 5.3: Example – Expected State after $v = 7$ is inserted twice

However, due to lack of synchronization mechanisms between the 2 threads, simultaneous memory reads and writes may very occur which could cause the subsequent table state to only capture a *single* insertion:

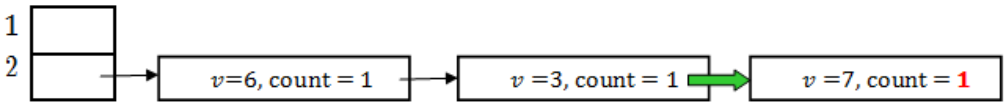

Figure 5.4: Example – Race Condition where only 1 insertion is captured

The count of permutation value 7 is erroneously recorded as 1 instead of 2!

It is clear that a lock is required for each hash bucket. In particular, the atomic functions `atomicCAS()` and `atomicExch()` built in CUDA C are used in our lock implementation. `atomicCAS()` is an atomic compare-and-swap function. It accepts three parameters – the first is a memory pointer `ptr`, the second and third are values `v1` and `v2`. If the value at `ptr` is equal to `v1`, then `v2` will be stored at `ptr`. `atomicCAS()` returns the value originally stored in `ptr`. With helpful guidance from Kandrot and Sanders' *CUDA by Example*[25], a GPU Lock structure is implemented as follows:

```
#define LOCKED 1
#define UNLOCKED 0

struct Lock {
        int *condition;
        Lock(void) {
                cudaMalloc((void**)&condition, sizeof(int));
                cudaMemset(condition, UNLOCKED, sizeof(int));
        }

        ~Lock(void) {
                cudaFree(condition);
        }

        __device__ void lock(void) {
                while(atomicCAS(condition, UNLOCKED, LOCKED) ==
                    LOCKED);
        }
```

```
        __device__ void unlock(void) {
                atomicExch(condition, UNLOCKED);
        }
};
```

In the function `lock()`, any thread attempting to acquire that `Lock` will wait until its `condition` is `UNLOCKED`, after which the thread will atomically change the `Lock's` condition to `LOCKED`. The function `unlock()` essentially resets the `Lock's` to `UNLOCKED`.

An array of `Locks` of the length of the hash table is needed, with each `Lock` responsible for a hash bucket. If there are more than one thread accessing a bucket at any one time, then these threads will take turns to insert their permutation values into the bucket through either the "Search & Insertion" or "Search & Modification" procedure.

5.4.1.2 A Warp Competing for a Lock

When too many threads compete for the same lock concurrently, the CUDA C program stops. To avoid this situation, we have to first understand a little about the concept of a **warp** in the CUDA Architecture. A warp is a group of 32 threads that execute in lockstep. Such lockstep execution is dangerous because it is likely to cause the situation where many threads fight for the same lock at the same time. Kandrot and Sanders' *CUDA by Example*[25] recommends alleviating this situation with a simple software tweak that steps through each thread in the warp and provides each with the opportunity, one after another, to acquire the lock. The relevant code demonstrating this step-through is as follows:

```
__global__ void insertPermutation(unsigned int *v, unsigned int
    *hashValues, HashTable hashTable, Lock *lockList) {
       int index = threadIdx.x + blockIdx.x * blockDim.x;
       if (index < noOfPossiblePermutations) {
              unsigned int pValue = pValues[index];
              unsigned int hashValue = hashValues[index];
              int modified = 0;

              /* Steps through each thread in all warps */
              for (int i = 0; i < 32; i++) {
                     if ((index % 32) == i) {
                            lockList[hashValue].lock();
                            ... // insert the permutation value
                                into the hash table
                            lockList[hashValue].unlock();
                     }
              }
       }
```

```
}
```

5.5 GPU Stage 3: Calculating the PE

5.5.1 Implementation

Recall Definition 2 in Section 2.1.3:

$$PE = -\sum_{v=1}^{N!} \pi_v \log \pi_v$$

where π_v is the frequency associated with the permutation value v. The main idea behind the CUDA implementation is simple enough:

1. Have each thread take charge of a hash bucket by stepping through every `Permutation` node in the bucket's linked list and summing up $-\pi_v \log \pi_v$ in the process. Then each thread will have calculated a "subPE".

2. Add up all the threads' subPE to get final PE value.

The first step is straightforward. For the second step, assuming large hash table size, there will be numerous threads with subPE values, and adding them up one by one in GPU is not the wisest choice, especially if global memory accesses are involved. The following section outlines the method of using reductions within each block to optimize the speed of summing up the subPEs.

5.5.1.1 Reductions within Blocks

CUDA C provides an area of shared memory for all threads in the same block. This area of shared memory is located on the GPU instead of off-chip DRAM. Therefore, accessing shared memory involves considerably lower latency. This fact enables all threads in the same block to be able to efficiently communicate when computations across threads are involved. As a result, it makes good sense to first sum up the subPEs of all threads within a block using the method of reductions, then add these sums up to derive our final PE.

For every block, a shared memory buffer is declared. After the threads of the same block calculate their respective subPEs (each thread calculating one subPE), they will store their subPEs into the shared memory buffer. Then a CUDA C function `__syncthreads()` is called. This synchronizes all threads in the block and ensures that every thread in the block has inserted its calculated subPE into the shared buffer before continuing to execute the next instruction.

Now we can perform reductions on this shared buffer of subPEs to derive their sum. A *reduction* refers to the procedure of deriving a smaller set of values from a larger input set after a series of operations. The most straightforward way of performing a reduction in this context is to use one thread to step through the shared memory array of subPEs and iteratively sum them up. This is not the wisest way because we did not exploit the GPU's immense capacity for parallelism. A faster way is to implement concurrent threads to continuously convert two subPE values into just one subPE value by adding both up.

For simplicity's sake, assume the original number of subPEs is a power of 2. In the first iteration, each concurrent thread transforms a pair of subPEs into a single subPE using addition. `__syncthreads()` is used here to ensure all threads in the same block have completed the first iteration. Consequently, half the number of subPEs are left. After the second iteration, only a quarter of the original number of subPEs remain. This continues until all the original subPEs are added up. The diagram below illustrates one such iteration.

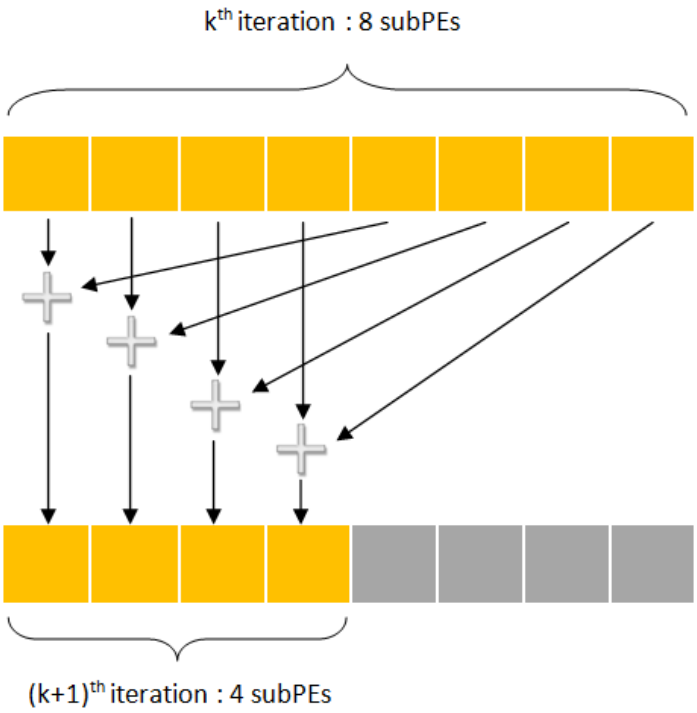

Figure 5.5: One iteration of a reduction

Now each block of threads carries one subPE value; we have to sum up these to obtain the final PE. The number of blocks is very small compared

to the original total number of threads. The small number of subPE values that remain cannot exploit the immense parallelism of the GPU well enough to compensate for the relatively slower speed of the device (as compared to CPU). Therefore, the last step is to transfer the last set of subPE values to the CPU and let the CPU sequentially sum them up to calculate the final PE.

Notice that the exact number of blocks to work with has not been specified. Since the number of blocks is the number of subPEs that the CPU have to sum, it cannot be too large. On the other hand, the number of blocks must be large enough to fully exploit the parallelism of the GPU. We adopt Kandrot and Sanders' *CUDA by Example*[25] proposal to use 32 blocks in such a situation.

5.6 Final Plan Of Action

As put forth in Section 5.2, there are 2 options for the final plan of action. These 2 options are:

1. Transfer the permutation values from CPU to GPU, and subsequently compute the FBD hash values of the permutation values in parallel in GPU (GPU Stage 1). Create a hash table in GPU, and insert these permutation values into this hash table based on their computed hash values (GPU Stage 2). Lastly, unravel the completed hash table and calculate the PE based on the frequencies of the stored permutation values (GPU Stage 3).

2. Perform GPU Stage 1 as in the second option. Then transfer the hash values from GPU back to CPU and insert the permutation values, based on these hash values, into a hash table in CPU (CPU Stage 2). Finally compute the PE in CPU based on the completed hash table (CPU Stage 3).

To decide between these 2 options, we have to compare the combined execution of GPU Stages 2 and 3 against the combined execution of CPU Stages 2 and 3. For this comparison, hash table sizes of 10^3, 10^4, 10^5, and 10^6 are used. Randomly generated sequences of permutation values and corresponding hash values of length $4 \times$ table size are used for insertions into the hash tables in GPU and CPU. These are the results:

No. of Buckets of Hash Table	Time taken for CPU Stages 2 & 3 (ms)	Time taken for GPU Stages 2 & 3 (ms)
1000	1	72
10000	6	1085
100000	60	12642
1000000	599	248145

Table 5.2: **Stages 2 and 3: CPU vs GPU.**

As can be seen from the table above, our hypothesis that using locks in GPU could slow down performance substantially proved to be resoundingly true. This set of results clearly favour the execution of CPU Stages 2 & 3 over GPU Stages 2 & 3. Therefore, Option 2 is chosen as the final plan of action. In other words, in its entirety, the final plan of action for hardware acceleration is:

Hardware Acceleration: Plan of Action.

1. Work out the entire sequence of permutation values of a large data series in FPGA (implemented by Ce).
2. Transfer the permutation values from FPGA to CPU to GPU, and subsequently compute the FBD hash values of the permutation values in parallel in GPU (GPU Stage 1).
3. Transfer the hash values from GPU back to CPU and insert the permutation values, based on these hash values, into a hash table in CPU (CPU Stage 2).
4. Finally, in CPU, unravel the completed hash table and calculate the PE based on the frequencies of the stored permutation values (CPU Stage 3).

Chapter 6

Conclusion

6.1 Summary of Achievements

The following is a summary of the author's key achievements in permutation hashing:

6.1.1 An Original Permutation Hash

This paper devises an original hashing algorithm specialized in minimizing collisions when hashing permutations. By exploiting the subtle relationships between neighbouring permutation patterns that exist by virtue of data overlaps, this novel hash function contrives to hash different permutations into distinct slots.

The process to exploit these relationships involves the use of number theory concepts to construct an original assignment of weights to every permutation value with respect to each hash table index. These weights are used to compute the Feature-Bias Divergence (FBD) of a table index I, which is a novel concept that the hashing algorithm is based on. For this reason, this original permutation hash is named the FBD hash.

6.1.2 Implementation Details & Performance Evaluation

The FBD hash is implemented using some neat tricks and well-established algorithms in number theory, such as the extended Euclidean algorithm, the Miller-Rabin test and Bertrand's Postulate. As there is a loop in the algorithm's final step, a vigorous, non-trivial proof of termination of that loop is provided. Due to the loop being the algorithm's bottleneck, termination time analysis on the loop is performed by constructing and proving a probability bound on the loop's survival through i iterations.

The performance of the FBD hash is evaluated through the use of a variety of relevant test data types, 4 traditional hash functions functioning as benchmarks, and 3 hash performance metrics. Each of these metrics gives a measure of collisions in the hash table. Among these 3 metrics, 1 is found in literature, while the other 2 are novel hash performance metrics proposed by the author. Also, a discussion about why metrics measuring collisions and not empirical hashing speed are used in this context.

Tests are performed over a range of orders and the results are analysed. From the results, the primary conclusion is that the FBD hash shows promise to be a good permutation hash function, while the secondary conclusion is that the 2 original metrics proposed by the author are credible hash performance metrics. Finally, limitations of the metrics used are discussed.

6.1.3 Hardware Acceleration of the FBD Hash using CUDA

After explaining the infeasibility of hashing in FPGA, this paper breaks down the process from hashing permutations to the final calculation of the PE into 3 main stages. These 3 stages are (1) calculating the FBD hash values from a large input sequence of permutation values, (2) inserting the permutation values into the hash table based on the calculated hash values, and (3) unravelling the completed hash table to calculate the final PE.

The pros and cons of performing each of these 3 stages in CPU and GPU are discussed. Subsequently, the GPU implementation of the 3 stages is described; key features of implementation include modification of the original FBD hashing algorithm, a locking mechanism for hash table insertions, scheduling a warp, and the use of reductions within blocks of threads in CUDA architecture.

The amount of hardware acceleration of the FBD hash offered by the GPU is evaluated by comparing the time taken to complete the 3 stages in both GPU and CPU. After the speed comparisons, the final plan of action for hardware acceleration is decided upon.

6.2 Future Work

Though we have achieved our objectives stipulated in Section 1.3, there is still a great amount of polishing work to be done before the FBD hash can be convincingly used as a fast and effective permutation hash. Possibilities for future work include:

- *Using a fast implementation of big number manipulation to further evaluate the FBD hash's performance.* The main objective of the FBD

hash is to accelerate the hashing of a range of permutations that is larger than the maximum possible size of the table in memory (hence collisions are expected). This means that the range of permutation keys is likely to reach the big number zone (> 64 bits) – this happens when order N is big enough such that $N!$ is a big number. However, due to the lack of ability to handle big numbers, this paper does not use very large range of permutation keys to evaluate the FBD hash's performance. Therefore, the test results in this paper can only give a good, but not convincing, indication of the FBD hash's performance. Implementing a program capable of manipulating big numbers quickly will go a long way in boosting the vigour of performance evaluation.

- *Conducting more comprehensive tests.* The primary purpose of the performance evaluation in Section 4.2 is to provide a blueprint for how future evaluations of the FBD hash can be carried out; it is nowhere near comprehensive enough for its results to be absolutely conclusive. Apart from the aforementioned issue of big number manipulation, more convincing performance evaluation of the FBD hash can be conducted by employing a greater variety of (1) relevant real-world data series, and (2) traditional hash functions functioning as benchmarks.

- *Adopting a good overall hash strategy.* So far all attention has been devoted to coming up with a good permutation hash function, yet adopting a good overall hash strategy can enhance the hash performance even further. Example of hash strategies are cuckoo hashing and 2-choice hashing, both of which are described in Section 2.2.2. Other possibilities include replacing the current simple linked list implementation of a chain with an ordered linked list or a binary search tree. A hash strategy describes an algorithm for hashing which makes use of one or more hash functions; it is not a hash function per se. Therefore, the adoption of a good hash strategy could complement the FBD hash and make performance even better. There are plenty of well-known hash strategies, but finding a suitable one for permutation hashing will require substantial performance tests.

- *Collaborating with Ce to derive the permutation values of large real-world time series using FPGA.* There is a lack of access to large sequences of permutation values that are based on real-world time series. Therefore, the permutation values that are used in our hash performance evaluation do not significantly exhibit the pattern-repeating phenomenon typical of real-world data series. This is the main reason why hashing speed is not used as a performance metric. Yet actual hashing speed measures are important in evaluating the performance of the FBD hash. Hence collaborating with Ce, who currently has a very fast FPGA implementation for the permutation value computa-

tions of large data series, can lead to a better performance evaluation of the FBD hash.

- *Improving/Extending our GPU implementation.* The current implementation of computing permutation values from a large data series is done by Ce in FPGA. The current plan of action is to transfer these permutation values from FPGA to CPU, then to GPU for hashing. An alternative is to simply compute these permutation values directly in GPU. In this way, the parallelism of GPU can be exploited and more importantly, the overhead cost of the FPGA-CPU-GPU transfer is eradicated. That said, it is an open question whether this alternative is faster than the current plan of action; determining the answer requires future implementation and testing! Another possible extension to current GPU implementation is the use of multiple GPUs to make our work even faster.

Bibliography

[1] Zemke, S., *Bagging Imperfect Predictors*, Proceedings of Artificial Neural Networks in Engineering, St. Louis, Missouri, pp. 1067-1072, 1999.

[2] Kaboudan, M. A., *Genetic Programming Prediction of Stock Prices*, Computational Economics, vol. 16, pp. 207-236, 2000.

[3] Walid, G. A., Mohamed, G. E., and Ahmed, K. E., *Incremental, Online, and Merge Mining of Partial Periodic Patterns in Time Series Databases*, IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 3, 2004.

[4] Mallat, S., and Zhong, S., *Characterization of Signals from Multiscale Edges*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, no. 7, July 1992.

[5] C. Bandt and B. Pompe, Phys. Rev. Lett. 88, 174102 (2002); C. Bandt, G. Keller, and B. Pompe, Nonlinearity 15, 1595 (2002).

[6] Shannon, C.E. *A mathematical theory of communication.* Bell Syst. Tech. J. 1948, 27, 379-423.

[7] Zanin M, Zunino L, Rosso OA, Papo D. *Permutation Entropy and Its Main Biomedical and Econophysics Applications: A Review.* Entropy. 2012; 14(8):1553-1577.

[8] Feldman, D.P.; Crutchfield, J.P. *Measures of statistical complexity: Why?* Phys. Lett. A 1998, 238, 244-252.

[9] *Dataflow.* 2014. [e-book] Addison-Wesley Publishing Company. pp. 169-170. `www.nondot.org/sabre/Mirrored/AdvProgLangDesign/finkel06.pdf` [Accessed: 21 Feb 2014].

[10] Mandle, A. 2008. *A Case in Protein Identification.* [e-book] pp. 6-7. `http://scale.engin.brown.edu/theses/mandle.pdf` [Accessed: 21 Feb 2014].

[11] *Hash Table.* Wikipedia. Wikimedia Foundation, 4 Mar 2014. Web. 27 May 2014. http://en.wikipedia.org/wiki/Hash_table.

[12] *Euclidean Algorithm.* Wikipedia. Wikimedia Foundation, 11 May 2014. Web. 27 May 2014. http://en.wikipedia.org/wiki/Euclidean_algorithm.

[13] *CUDA.* Wikipedia. Wikimedia Foundation, 25 May 2014. Web. 18 Apr 2014. http://en.wikipedia.org/wiki/CUDA.

[14] *NVIDIA CUDA.* NVIDIA. NVIDIA. Web. 20 Apr 2014. http://www.nvidia.co.uk/object/cuda_home_new.html.

[15] Shoup, Victor. *A Computational Introduction to Number Theory and Algebra.* Choice Reviews Online 43.11 (2008): n. pag. A Computational Introduction to Number Theory and Algebra. 16 June 2008. Web. 4 May 2014. http://shoup.net/ntb/ntb-v2.pdf.

[16] Weisstein, Eric W. *Rabin-Miller Strong Pseudoprime Test.* MathWorld. http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html.

[17] Walker, Julienne. *Eternally Confuzzled - The Art of Hashing.* Eternally Confuzzled. N.p., n.d. Web. 01 Mar. 2014. eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx.

[18] Aho, A. V., & Sethi, R. (1986). *Compilers, principles, techniques, and tools.* Reading, Mass.: Addison-Wesley Pub. Co..

[19] Wang, Thomas. *Integer Hash Function.* GitHub Gists. N.p., Mar. 2007. Web. 03 May 2014. https://gist.github.com/badboy/6267743.

[20] Andrzejak, Ralph, K. Schindler, and C. Rummel. *Andrzejak RG, Schindler K, Rummel C (2012).* Nonrandomness, Nonlinear Dependence, and Nonstationarity of Electroencephalographic Recordings from Epilepsy Patients. Phys. Rev. E, 86, 046206." Nonlinear Time Series Analysis Group. N.p., 16 Oct. 2012. Web. 05 June 2014. http://ntsa.upf.edu/downloads/andrzejak-rg-schindler-k-rummel-c-2012-nonrandomness-nonlinear-dependence-and.

[21] Quiroga, Rodrigo. *EEG Data, Evoked Potentials, EEG Analysis, Extracelullar Recordings.* EEG, ERP and single cell recordings database. N.p., n.d. Web. 05 June 2014. http://www.vis.caltech.edu/~rodri/data.htm.

[22] *Documentation.* QuantQuote. N.p., n.d. Web. 05 June 2014. https://quantquote.com/support_documentation.php.

[23] *Free Forex Historical Data.* HistDatacom RSS. N.p., n.d. Web. 05 June 2014. http://www.histdata.com/download-free-forex-historical-data/?/ninjatrader/tick-bid-quotes/eurusd/2014/5.

[24] *Dow Jones Industrial Average Stock - Yahoo! UK & Ireland Finance.* Yahoo! Finance. N.p., n.d. Web. 05 June 2014. https://uk.finance.yahoo.com/q/hp?a=&b=&c=&d=5&e=5&f=2014&g=d&s=%5EDJI%2C+&ql=1.

[25] Jason Sanders and Edward Kandrot. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming (1st ed.)*. Addison-Wesley Professional.