DESIGN OF SPECIAL FUNCTION UNITS IN MODERN MICROPROCESSORS

A Dissertation

by

ABBAS A. E. A. FAIROUZ

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2019

Major Subject: Computer Engineering

ABSTRACT

Today's computing systems demand high performance for applications such as cloud computing, web-based search engines, network applications, and social media tasks. Such software applications involve an extensive use of hashing and arithmetic operations in their computation. In this thesis, we explore the use of new special function units (SFUs) for modern microprocessors, to accelerate such workloads.

First, we design an SFU for hashing. Hashing can reduce the complexity of search and lookup from O(p) to O(p/n), where n bins are used and p items are being processed. In modern microprocessors, hashing is done in software. In our work, we propose a novel hardware hash unit design for use in modern microprocessors. Since the hash unit is designed at the hardware level, several advantages are obtained by our approach. First, a hardware-based hash unit executes a single hash instruction to perform a hash operation. In a software-based hashing in modern microprocessors, a hash operation is compiled into multiple instructions, thereby degrading performance. Second, software-based hashing stores hash data in a DRAM (also, hash operation entries can be stored in one of the cache levels). In a hardware-based hash unit, hash data is stored in a dedicated memory module (a hardware hash table), which improves performance. Third, today's operating systems execute multiple applications (processes) in parallel, which entail high memory utilization. Hence the operating systems require many context switching between different processes, which results in many cache misses. In a hardware-based hash unit, the cache misses is reduced significantly using the dedicated memory module (hash table). These advantages all reduce the power consumption and increase the overall system performance significantly with a minimal increase in the microprocessor's die area. We evaluate our hardware-based hash unit and compare its performance with software-based hashing. We start by evaluating our design approach at the micro-architecture level in terms of system performance. After that, we design our approach at the circuit level design to obtain the area overhead. Also, we analyze our design's power and delay for each hash operation. These results are compared with a traditional hashing implementation. Then, we present an

FPGA-based coprocessor for hash unit acceleration, applied to a virus checking application.

Second, we present an SFU to speed up arithmetic operations. We call this arithmetic SFU a programmable arithmetic unit (PAU). In modern microprocessors, applications that require heavy arithmetic computations are done in software. To improve the performance for such computations, we present a programmable arithmetic unit (PAU), a partially reconfigurable methodology for arithmetic applications. The PAU consists of a set of IP blocks connected to a reconfigurable FPGA controller via a fast mesh-based interconnect. The IP blocks in the PAU can be any IP block such as adders, subtractors, multipliers, comparators and sign extension units. The PAU can have one or more copies of the same IP block (for example, 5 adders and 7 multipliers). The FPGA controller is an on-chip FPGA-based reconfigurable control fabric. The FPGA controller enables different arithmetic applications to be embedded on the PAU. The FPGA controller is programmed for different applications. The reconfigurable logic is based on a LUT-based design like a traditional FPGA. The FPGA controller and the IP blocks in the PAU communicate via a high speed ring data fabric. In our work, we use the PAU as an SFU in modern microprocessors. We compare the performance of different hardware-based arithmetic applications in the PAU with software-based implementations in modern microprocessors.

To my parents, my wife, my parents in law, my whole family, and my friends.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my committee chair Dr. Sunil P. Khatri. I thank him for the continuous support of my PhD study and research, for his patience, motivation and enthusiasm. His guidance helped me in all the time of research and writing of this dissertation.

I would also like to thank my committee members: Dr. Paul V. Gratz, Dr. Laszlo B. Kish and Dr. Anxiao (Andrew) Jiang for their valuable suggestions and feedback.

I would like to thank my fellow doctoral students Dr. Monther Abusultan and Dr. Viacheslav V. Fedorov, for their feedback, cooperation and of course friendship. Also, special thanks to my colleagues Andrew Douglass, Kunal Bharathi and Mian Qin for their feedback and support.

Finally and most importantly, I would like to thank my parents, siblings, parents in law for supporting me spiritually throughout writing this dissertation and my life in general. Special and many thanks to my lovely wife for her endless support. I would like to thank her from my heart for standing next to me through my tough journey of my PhD study. Also, I would like to thank my friends for accepting nothing less than excellence from me.

I would like to thank Kuwait University for their support. Also, I would like to thank the Texas A&M University Office of Graduate and Professional Studies for allowing me to construct this LaTeX dissertation template.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Sunil P. Khatri (advisor), Paul V. Gratz and Laszlo B. Kish of the Department of Electrical and Computer Engineering and Professor Anxiao (Andrew) Jiang of the Department of Computer Science and Engineering.

The experimental work conducted for the dissertation was completed by the student, under the supervision and guidance of Dr. Khatri. The experiments in Chapter 6 were conducted with assistance from Andrew Douglass and Kunal Bharathi.

Funding Sources

Graduate study was supported by Kuwait University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Design of Modern Microprocessors

Modern Microprocessors typically use a *complementary metal oxide semiconductor* (CMOS) transistors to build *integrated circuits* (ICs), using *very large scale integration* (VLSI). There are two types of CMOS devices: *N-channel metal oxide semiconductor* (NMOS) and *P-channel metal oxide semiconductor* (PMOS) transistors. In 1965, Gordon Moore observed that the number of transistors in an IC will double every 18 months (Moore's law [11]). The transistor count for each

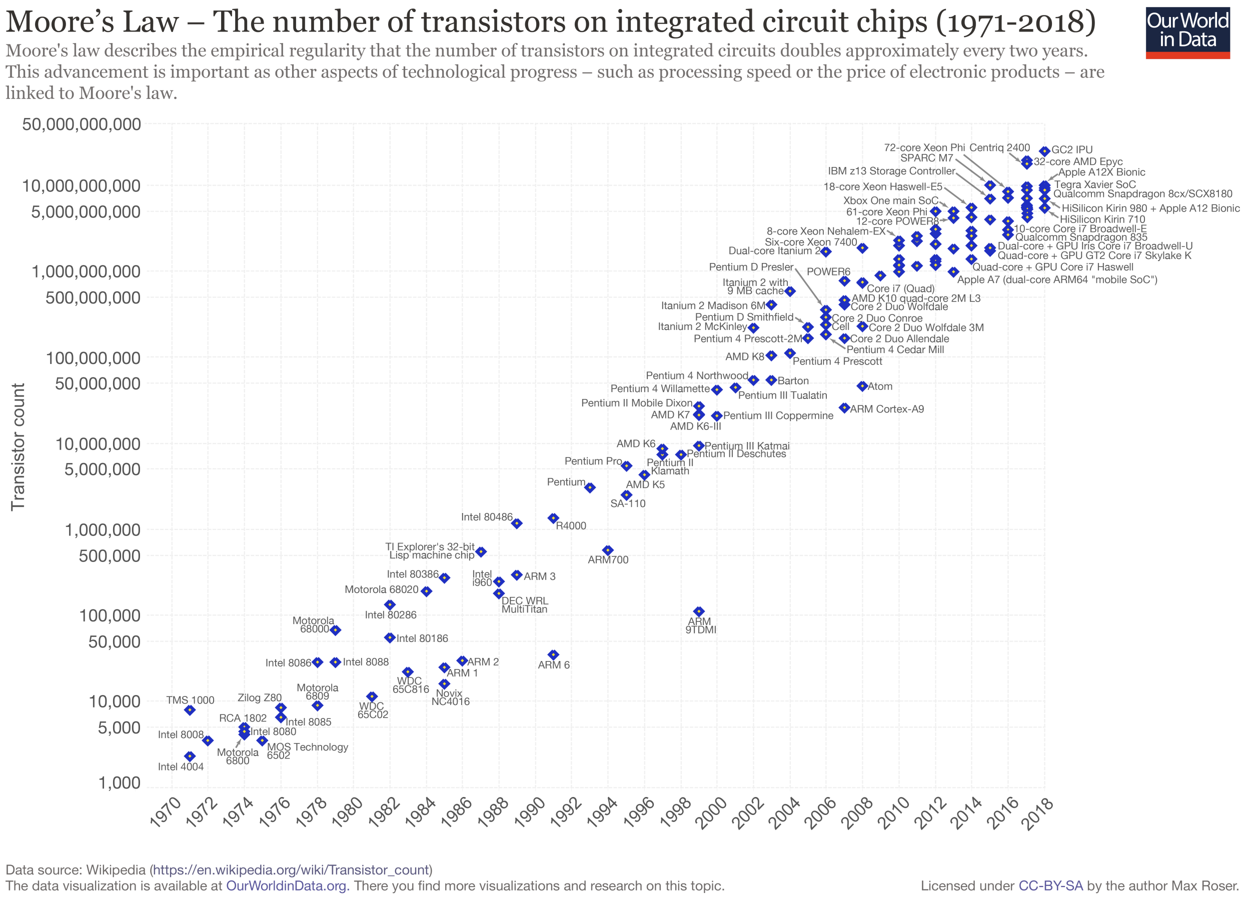


Figure 1.1: Moore's Law Trend from 1971 to 2016 (Our World in Data [1], Max Roser)

IC in the period from 1971 to 2016 is shown in Figure 1.1. In Figure 1.1, the x-axis represents the year of introduction and the y-axis represents the number of transistors in each IC on a logarithmic axis. The dots in Figure 1.1 represent each IC. Also, the curve shows the transistor count doubling every two years, which is in keeping with Moore's law. Until recently, researchers in the field (in both the academy and industry) have worked together to sustain Moore's law. More recently, the scaling of transistor sizes cannot keep up with Moore's law because transistor dimensions have reached lithographic limits. It is becoming extremely challenging to shrink the transistor device to smaller sizes. Today's applications require higher performance and lower power consumption. As a result, the IC industry is working hard to find new technologies and approaches as an alternative to scaling CMOS transistor dimensions. One of these approaches is hardware acceleration. Hardware acceleration is used to speedup common software algorithms in today's applications. First, we will discuss the modern microprocessor performance metrics. Then, we will study the tradeoffs of these metrics to improve the performance of today's applications.

1.2 Modern Microprocessor Performance Metrics

In modern microprocessors, the key performance metrics are delay, power dissipation and area. Microprocessor designers usually use one or more of these performance metrics as their target (ie. high performance computers, low power processors, small design footprints, etc.). For microprocessor designers, it is extremely hard to achieve an improvement in all microprocessor performance metrics. Usually, achieving an improvement in one of the microprocessor performance metrics results in a degradation in another microprocessor performance metric. In the case of using a smaller technology node (ie. moving from 45nm technology node to 32nm technology node), microprocessor designers can achieve improvements in all of these performance metrics.

Next, we will discuss the *delay* as the first microprocessor performance metric.

1.2.1 Delay

In digital circuit design, there are different types of *delays* [2] for combinational and sequential circuits. In combinational circuits, the most important types of delay are: *propagation delay* (t_{pf}

or t_{pr}), *fall time* (t_f), *and rise time* (t_r). The *propagation delay* (t_{pf} or t_{pr}) is the time that it takes an input signal of a logic gate to appear at the output. Usually the t_{pf} or t_{pr} is measured at 50% of the voltage source from input signal to the output signal. As shown in Figure 1.2, the *falling*

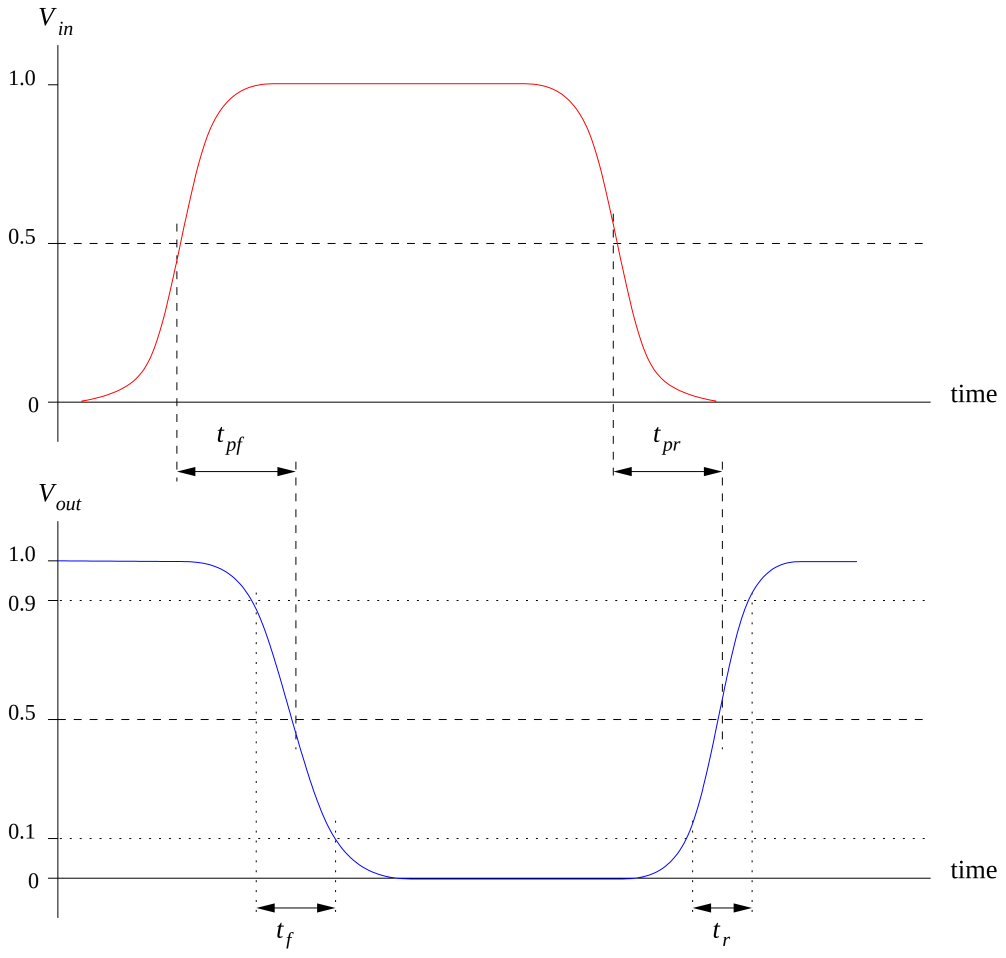


Figure 1.2: Propagation delay (t_{pf} or t_{pr}), fall time (t_f) and rise time (t_r) [2]

propagation delay t_{pf} is the time that it takes a transition at the input signal of a logic gate to appear as the falling output signal of the same logic gate. The *rising propagation delay* t_{pr} is the time that it takes a transition at the input signal of a logic gate to appear as the rising output signal of the same logic gate (Figure 1.2). The *fall time* (t_f) is the time that it takes an output signal to fall from 90% of the supply voltage to 10% (Figure 1.2). The *rise time* (t_r) is the time that it takes an output signal to rise from 10% of the supply voltage to 90% (Figure 1.2). Sometimes t_r and t_f are measured with the reference levels set to 20% and 80% of the supply voltage. In sequential circuits, we consider other types of delay. The most important types of delay in sequential circuits are: *setup time* (t_{setup}), *hold time* (t_{hold}) and *clock-to-Q* propagation delay (t_{cqp}). In a rising (falling) edge triggered flip-flop, the input signal D must arrive by a specific time (*setup time, t_{setup}*) before the rising (falling) edge of the clock, and stay stable until another specific time (*hold time, t_{hold}*) after the rising (falling) edge of the clock. The *clock-to-Q* propagation delay (t_{cqp}) is the time that takes the input signal D to propagate through the flip-flop until it appears at the output signal Q of the same flip-flop after the rising (falling) edge of the clock.

We measure the delay of a circuit design using *static timing analysis* (STA). In any circuit design, we calculate the maximum delay using STA tools. The STA tools report the maximum delay after calculating the maximum delay in the design. The critical path is the circuit path that results in the worst case delay in the design.

Next, we will discuss the *power dissipation* as the second microprocessor performance metrics.

1.2.2 Power Dissipation

In the digital circuit design, the average *power* dissipation is represented by the average of the power consumed over some time interval *T* [2].

$$P_{avg} = \frac{1}{T}\int_0^T P(t)dt \tag{1.1}$$

Here $P(t)$ is the instantaneous power consumed at time t. The power dissipation in digital circuits consists of two parts: *dynamic power* (P_{dyn}) and *static power* (P_{st}). The *dynamic power* (P_{dyn}) is the power consumed during the switching activity (during transitions from logic "0" to logic "1" or vice versa) of the circuit. The *static power* (P_{st}) is the power consumed while the circuit is idle (no switching activity). One of the causes of static power in CMOS digital circuits is the subthreshold leakage when the CMOS transistor is OFF. Several efforts have been made to address this issue. One of the ways to reduce the leakage in CMOS circuits is power gating, which works by isolating the power source from idle parts in the circuit [12].

The total power dissipation in the circuit is the sum of dynamic and static power., as shown in the following formula:

$$P_{total} = P_{dyn} + P_{st} \tag{1.2}$$

Next, we will discuss the *area* as the third microprocessor performance metrics.

1.2.3 Area

Circuit *area* is an important metric in digital circuit design since microprocessor designers are limited to a chip die area. By using a smaller process node technology in CMOS transistors (usually, each new technology node is 30% smaller than the previous one [2]), microprocessor designers are capable to embed more CMOS transistors in the same area. Microprocessor designers, in early design stages, usually estimate the area by calculating the *active area* of all the channel regions in all CMOS transistors in the circuit. This is also called the *active area* of the circuit. As

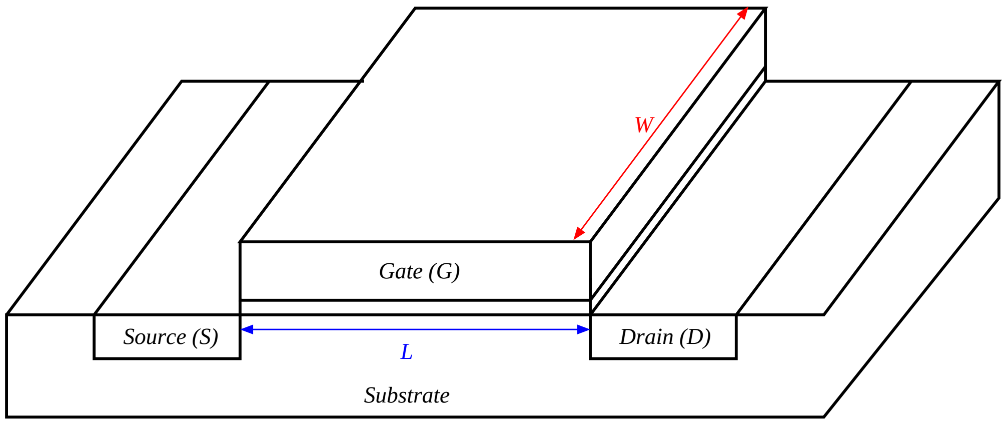


Figure 1.3: Physical structure of a CMOS transistor

shown in Figure 1.3, the *active area* of a single CMOS transistor is composed of the *width (W)* and the *length (L)* of the channel:

$$Area_{active} = W \times L \tag{1.3}$$

The *active area* does not represent the actual physical area of the design. For a more accurate area representation, microprocessor designers use the *cell area*. The *cell area* includes the area of the entire cell and the local interconnects with respect to the cell.

1.3 Modern Microprocessor Design Challenges and Solutions

In this section, we will discus the key issue in microprocessor design: the scaling of CMOS. We then propose hardware acceleration as one way to address it.

1.3.1 Scalability of CMOS

CMOS transistor scaling is reaching its limits. The size of the CMOS transistor is limited by lithographic challenges. As a result, researchers are working on finding another alternative to overcome this issue and improve performance. One way to address this issue is to use hardware accelerators to accelerate portions of common software algorithms. Next, we will discuss the hardware acceleration in modern microprocessors.

1.3.2 Hardware Acceleration

Hardware acceleration is one solution to overcome the CMOS scaling issue. The main challenge for hardware accelerators is to find common algorithms for today's applications that would benefit from acceleration. Today's applications have a heavy utilization of hashing and arithmetic computations. In this thesis, we will study the possibility of implementing hardware accelerators for hashing and arithmetic computation.

In modern microprocessors, applications run as processes for a certain number of central processing unit (CPU) *clock cycles*, where the CPU *clock cycle* is a time unit in modern microprocessors. The CPU *clock cycle* is the inverse of the CPU frequency (*f*).

$$CPU_{cycle} = \frac{1}{CPU_f} \tag{1.4}$$

Assuming that a hardware accelerator in a modern microprocessor can speed up a portion (p) of a process by a factor s [13, 14], then, based on Amdahl's law [13], the overall speedup of a process can be calculated by the following equation:

$$Speedup_{overall} = \frac{Cycles_{old}}{Cycles_{new}} = \frac{1}{(1-p)+\frac{p}{s}} \tag{1.5}$$

As a results, the overall speedup can be increased by the increment of speedup portion (p) of a process or by increasing the speedup (s) of that portion of the process. To maximize the speedup portion p, it is important to focus on common algorithms that are utilized by most of today's applications.

1.4 Special Function Units (SFUs) in Modern Microprocessors

In modern microprocessors, there are several special function units (SFUs) such as integer unit (IntU), floating point unit (FPU), memory management unit (MMU) and vector extension unit (VXM) [14]. Figure 1.4 shows the common SFUs in a CPU. The IntU is used for integer arithmetic operations, while the FPU is use for floating point arithmetic operations. The MMU is used for memory operations such load or store data from the memory system in modern microprocessors. The VXM is used for vector processing applications that utilize a single instruction on multiple data (SIMD) operations. These SFUs are intended to speedup operations for common algorithms. In this thesis, we focus on exploring new hardware accelerators such as a hardware hash unit and a programmable arithmetic unit, to be used as new SFUs in modern microprocessors.

1.5 Field Programmable Gate Array (FPGA)

The *filed programmable gate array* (FPGA) is a flexible hardware platform, that can be reconfigured for multiple applications [15, 16]. Recently, the utilization of FPGA based designs

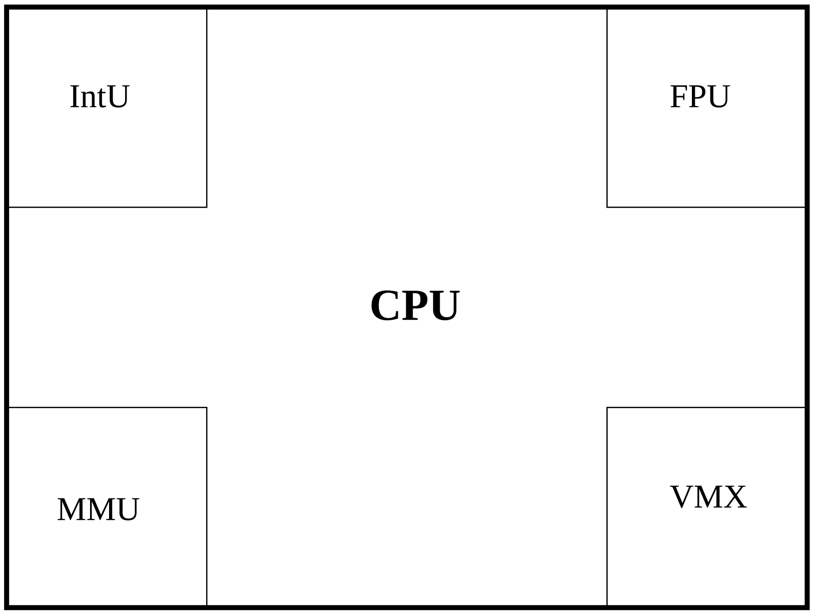


Figure 1.4: Common special function units (SFUs) in modern microprocessors

has been increased for low and medium volume applications, compared to *application specific integrated circuits* (ASIC). Using FPGAs have several advantages compared to an ASIC. For example, FPGAs have a faster design time and a lower non-recurring engineering (NRE) cost. The NRE cost includes an engineering design cost and a prototype manufacturing cost [2]. Also, the cost of ASIC IC fabrication masks has dramatically increased recently, which results in a reduction in ASIC designs [17]. As a result, FPGAs are increasingly favored over ASIC designs. As shown in Figure 1.5, FPGAs consists of four major parts: configurable logic blocks (CLBs), routing switches, vertical and horizontal routing wires, and configurable I/O pins. The CLB consists of one or more *look-up tables* (LUTs). Each LUT can be programmed to implement any logic function with a specific number of inputs (n) (usually, n is between 4 to 6, depends on the FPGA IC), using static random access memory (SRAM) cells. The routing switches are configured based on the configuration of the CLBs, to route the configured logic (using routing wires as well) from one CLB to another one.

A comparison between FPGA and ASIC has been reported in [18]. For a set of applications,

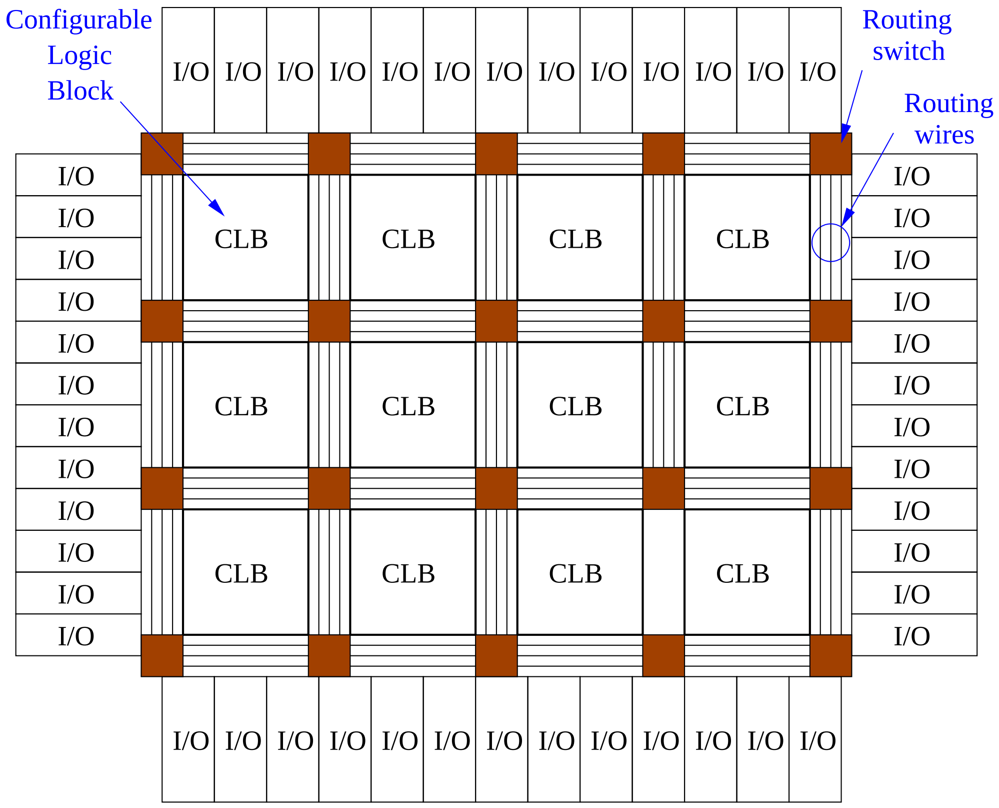


Figure 1.5: General FPGA architecture

FPGAs have 4.5× larger delay, 54× larger area, 14× more dynamic power and 87× more static power compared to ASICs. The FPGA has lower speed and higher power dissipation compared to ASICs, due to reconfigurability (which often results in redundant logic) and high wire density in the FPGA ICs. As a results, FPGAs is generally not a good candidate for low power designs. However, logic designers often use FPGAs, due to the fact that FPGAs have a faster design turn-around time, a lower NRE cost, and great flexibility.

1.6 Hashing in Computer Systems

In computer systems, there are several ways to store and retrieve data in a database such as link lists, arrays and hash tables. Usually, database entries have unique *keys* and multiple *values*. For example, entries in a database can be social security numbers (as a *key*) associated with their first names (as a *value1*) and last names (as a *value2*). Assume you have p (*key*, *value*) pairs

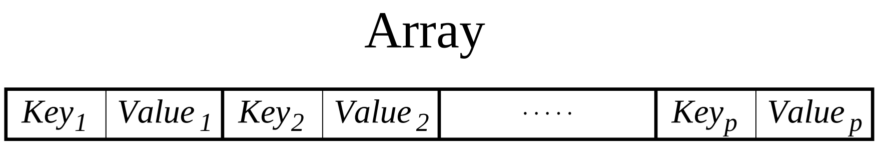


Figure 1.6: (*Key*, *Value*) pair in an array in Computer Systems

(further assume that a key is associated with a single value) to be stored in an array, as shown in Figure 1.6. The search complexity for a key will be $O(p)$. Hashing can reduce the complexity of search and lookup from $O(p)$ to $O(p/n)$, where n bins are used, as shown in Figure 1.7. In modern microprocessors hashing is done in software. As a result, software-based hashing often entails high CPU utilization and memory utilization. Hashing is performed on an entry of a (*Key*, *Value*)

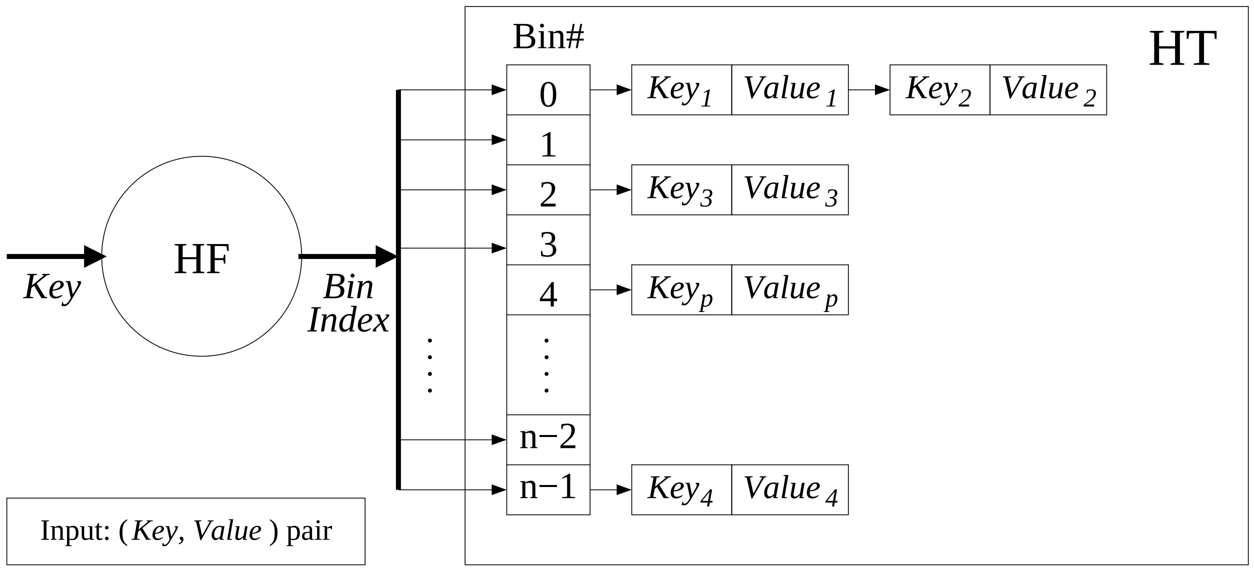


Figure 1.7: Hashing in computer systems

pair, as shown in Figure 1.7. Hashing consists of two major parts: a *hash function* (*HF*) and a *hash table* (*HT*). The *hash function* (*HF*) takes a *Key* as an input to produce a *Bin Index* that refers to one of the bins in the hash table (HT). The *hash table* (*HT*) stores the (*Key*, *Value*) pair entry in the corresponding bin. In software-based hashing, each bin is typically implemented as a linked list, with a search time that is $O(m)$, where m is the number of entries in a linked list, where $m \sim \frac{p}{n}$, assuming that the HF selects each bin index with equal probability.

1.6.1 The Hash Function (HF)

The hash function (HF) is typically used for the hash table data structure, to speedup data entry lookups (hash lookups). The hash function performs a logical or mathematical computations on a *Key* (hash function input), to produce a *Bin Index* to one of the bins in the hash table, as shown in Figure 1.7. For general purpose hash applications, universal classes of hash functions [19] are utilized. One of the best candidates for hardware-based hash applications is a hash function of class H3 [5]. This class of hash functions perform logical AND and XOR operations on a *Key* input of the hash function, which perform a fast hash operation.

Also, hash functions are used for different kind of applications, such as cryptography and message authentication. In our work, we will implement a hardware-based hashing for use in modern microprocessors. Therefore, we will focus on universal hash functions of class H3 [5]. These hash functions perform a fast hash operation, which is a perfect match for a hardware-based hashing application.

1.6.2 The Hash Table (HT)

The hash table (HT) is the data structure storage part of a hash operation. It stores (*Key*, *Value*) pairs in its bins. In the software-based hashing, the hash table bins are typically implemented as a linked list, as shown in Figure 1.7. In each bin of the hash table, the time complexity of a hash lookup entry is $O(m)$, where m is the number of entries in the linked list. Hash tables are typically stored in a dynamic random access memory (DRAM) in modern microprocessors.

In our work, we will implement a hardware-based hash unit, which consists of a hash function and a hash table. A content-addressable memory (CAM) [20] is mostly used in fast cache memory applications. In our hardware-based implementation of the hash unit, we choose a CAM to realize each bin. This provides $O(1)$ search time when the entry is in the CAM.

2. THESIS OUTLINE

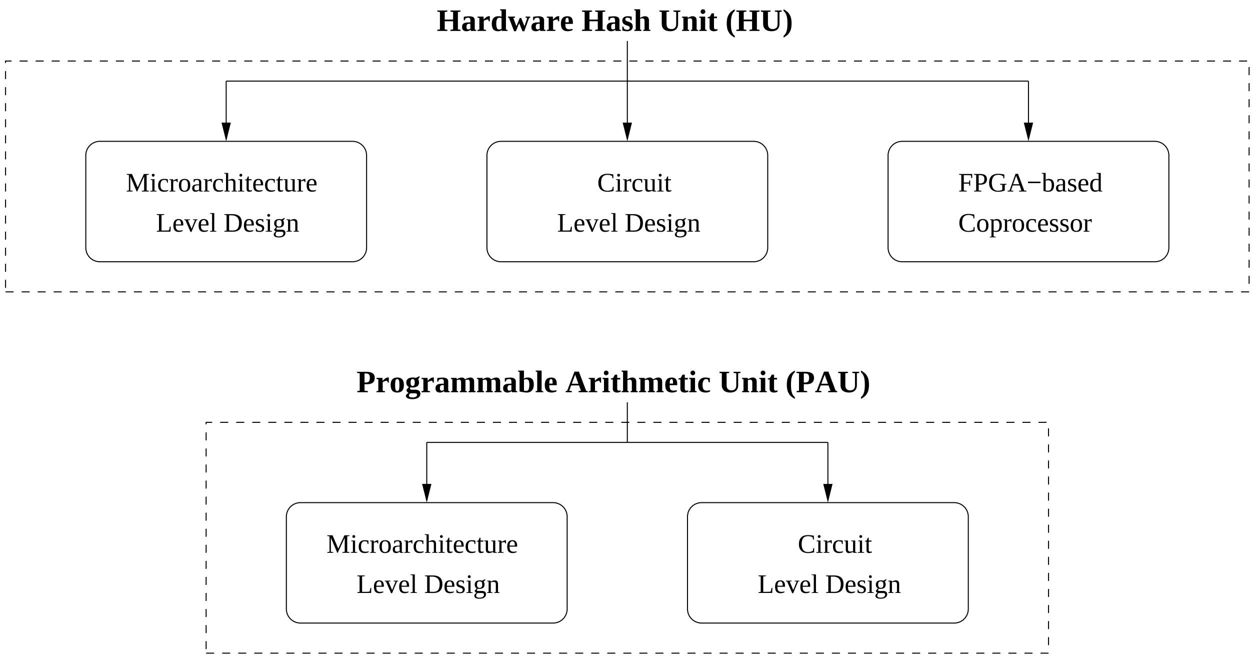

Hardware Hash Unit (HU)
Microarchitecture Level Design
Circuit Level Design
FPGA−based Coprocessor
Programmable Arithmetic Unit (PAU)
Microarchitecture Level Design
Circuit Level Design

Figure 2.1: Thesis outline diagram

In this thesis, we study the possibility of implementing a hardware hash unit and a programmable arithmetic unit for use in modern microprocessors, as new SFUs. The hardware hash unit is used to speed up hash operations, while the programmable arithmetic unit is used to accelerate a sequence of arithmetic operations.

For the hardware hash unit (HU), we present a detailed study at the microarchitecture level and at the circuit level, as illustrated in the top portion of Figure 2.1. In the HU microarchitecture level design, we present a detailed structure of the HU and how to embed it in modern microprocessors. We compare the number of cycles of the HU over a software-based implementation. We observe a significant speedup of the HU over the software-based hash. In the HU circuit level design, we present a detailed circuit design of the HU. We verify the correctness of all hash operations in the hash unit. We measure the delay, area and power of the HU, and compare them with a traditional CAM design. Furthermore, based on our HU at microarchitecture level and at circuit level, we implement an FPGA-based coprocessor for virus checking applications. We compare the speedup of our FPGA-based virus checking application with a software-based implementation. We demonstrate a good speedup of the FPGA-based virus checking application over the software-based implementation.

For the programmable arithmetic unit (PAU), we present a detailed study at the microarchitecture level and at the circuit level, as illustrated in the bottom portion of Figure 2.1. In the PAU microarchitecture level design, we present a general structure of the PAU and how to use it in modern microprocessors. We compare the speedup of the PAU (over three examples that use arithmetic operations heavily) over a software-based implementation. We demonstrate a significant speedup of the PAU for different arithmetic applications over the software-based implementation. In the PAU circuit level design, we present a circuit design of the PAU, and its design flow. We measure the delay, area and power of each arithmetic application in the PAU. We compare the area of the PAU with a 32kB level one data cache (L1D) in a 16nm technology. Also, we compare the power increase of the PAU with the average power of an Intel [21] i7 processor.

In the remainder of this chapter, we will describe each part of this thesis in more details.

2.1 Hardware Hash Unit (HU)

In this section, we will briefly describe our hardware hash unit (HU) at the microarchitecture level and at the circuit level. Also, we describe briefly our FPGA-based coprocessor for virus checking applications.

2.1.1 Design of a Hardware HU at Microarchitecture Level

In Chapter 3, we describe the implementation of our hardware hash unit (HU) at the microarchitecture level. We show the HU structure in detail and describe how to use it in modern microprocessors. The hash unit (HU) consists of two major parts: a hash function (HF) and a hash table (HT). The hash function (HF) is constructed from AND and XOR gates, to perform a high speed hash operation. The hash table (HT) in the HU consist of multiple bins. Each bin in the hash table is realized as a content-addressable memory (CAM), which can perform a single clock cycle hash search (lookup). We evaluate the performance of our HU and compare it with the performance of a software-based implementation. We demonstrate a significant speedup of our HU over the software-based implementation.

2.1.2 Design of a Hardware HU at Circuit Level

In Chapter 4, we describe the implementation of our hardware hash unit (HU) at the circuit level. We show the HU components in detail, and describe how to perform hash operations on it. As we mentioned earlier in the previous section, the hash unit (HU) consists of the hash function (HF) and the hash table (HT). In addition, we describe the control signal unit (CSU) in the HU and how to perform different hash operations. For the hash table (HT), we implement each bin using CAM cells (for *keys*) and SRAM cells (for *values*). We measure the delay, area and power of the HU and compare it with a traditional CAM design. We demonstrate a significant average power reduction of the HU compared to a traditional CAM, with a minimal area increase.

2.1.3 An FPGA-based Coprocessor for Virus Checking Applications

In Chapter 5, we describe the implementation of our FPGA-based coprocessor for virus checking applications, based on the ideas of Chapter 3 and Chapter 4. The FPGA-based coprocessor communicates with a CPU via the PCI Express (PCIe) bus. In the FPGA, we implement our HU, with each hash table bin in separate BRAMs, to implement a CAM-based structure. We perform a burst hash lookup operations (message-digest5 (MD5) virus signatures), so that we achieve a better utilization of the PCIe interface and increase the overall performance of our FPGA-based coprocessor for the virus checking applications. Also, we implement our FPGA-based coprocessor in a pipelined structure. We compare the performance of our FPGA-based coprocessor with a software-based implementation.

2.2 Programmable Arithmetic Unit (PAU)

In this section, we describe our programmable arithmetic unit (PAU) at microarchitecture level and at circuit level (in Chapter 6).

We show the PAU general structure in detail and describe how to use it for multiple arithmetic applications. The PAU consists of three major parts: tiles (IP blocks), a control logic (FPGA controller) and a fast ring data fabric. The tiles in the PAU can be any common IP block such as adders, subtractors, multipliers and comparators. The PAU can have one or more of the same tile. Each tile can be implemented using a register-transfer level (RTL) synthesis tool. The control logic (FPGA controller) in the PAU is FPGA-based. The FPGA controller enables different arithmetic applications to be embedded on the PAU. The FPGA controller is programmed for different applications. The reconfigurable logic of the FPGA controller is based on a LUT design like a traditional FPGA. We extract the control logic of the FPGA controller using one of the FPGA synthesis tools. The FPGA controller and the tiles in the PAU communicate via a fast ring data fabric. The ring data fabric operates at a high speed of up to 20$\times$ of the FPGA controller and the tile in the PAU. We study three arithmetic benchmarks in the PAU and measure their delay, area and power. We compare the performance of the PAU with a software-based implementation. We demonstrate a significant speedup of the PAU over the software-based implementation. We compare the area of the PAU with a 32kB of L1D cache of the same technology node. Also, we calculate the increase of power consumption of the PAU compared to an average power of an Intel [21] i7 processor.

3. MICROARCHITECTURE LEVEL DESIGN OF HARDWARE HASH UNIT FOR USE IN MODERN MICROPROCESSORS [1]

In this chapter, we present a hardware hash unit at the michroarchitecture level for use in modern microprocessors. We start with a background in Section 3.1. Then, in Section 3.2, we discuss the previous work. In Section 3.3, we present our proposed hardware hash unit at the michroarchitecture level design. After that, we present our experimental results in Section 3.4. We conclude in Section 3.5.

3.1 Background

Modern microprocessors are required to execute a complex and diverse set of applications. Historically, their performance has been optimized for integer and floating point benchmarks. Today's applications perform more diverse tasks such as those used in cloud computing, web-based search, networking, and social media. Therefore, modern microprocessors are required to perform high speed computation for an increasingly diverse set of tasks, while being constrained by memory utilization and fabrication technology. As a result, new techniques need to be explored to achieve high performance for applications that place diverse demands on modern microprocessors.

There have been many Special Function Units (SFUs) introduced to speedup commonly occurring tasks in the typical application set. Some of these include cryptographic units, floating point units and memory management units. To the best of our knowledge, there has been no work on a general purpose hash unit. In order to implement such hashing-intensive algorithms, microprocessors simply compile the application code and run the resulting instructions. As we demonstrate in this chapter, implementing a hardware hash unit (HU) can improve performance significantly. The proposed hash unit includes a special hash table memory, to speedup hashing operations. Hashing is one of the most important and commonly employed technique to store and lookup data. Design-

[1] Part of the data (including some figures and tables) reported in this chapter is reprinted with permission from [3] "A Novel Hardware Hash Unit Design for Modern Microprocessors" by A. Fairouz, M. Abusultan and S. P. Khatri, in 2016 IEEE 34th International Conference on Computer Design (ICCD), pp. 412–415, Oct 2016., Copyright 2016 by IEEE.

ing a special function unit to accelerate hashing-intensive algorithms can yield significant speedups for a wide class of applications.

As an example, several networking applications focus on hashing approaches to increase network packets' throughput. Fast internet protocol (IP) lookup is a significant application in networking. In order to achieve a high throughput for IP lookup, hashing is commonly employed. Such hashing implementations are done in software, and require the design of hash function as well as software-based hash tables. Another common use of hash functions involves search and membership checks.

Modern microprocessors do not have a hardware-based hash unit. In this chapter, we propose a new Hash Unit (HU) as a special function unit for modern microprocessors. The hash unit uses a special hash table memory, to store hash operations. The hash table memory module uses a content-addressable memory (CAM) to enable fast memory access. Our HU is embedded in the architecture pipeline of a modern microprocessors. Our HU obtains a speedup of up to 15$\times$ compared to a software-based hash application, with minimal increase in the area.

The key contributions of this chapter are:

- Design a hardware hash unit (HU) to speedup hash operations in modern microprocessors. To the best of our knowledge, this has not been undertaken to date.

- We simulated the design using GEM5 [22] (a Computer Architecture Platform) in the x86 ISA (augmented with our hash instructions) and verified the correctness of all hash operations.

- We observe a speedup of up to 15$\times$ while varying the size of the hardware hash table, CPU speed, cache size, memory technology and DRAM latency.

- We report the effect on cache misses when the HU is used, and also quantify the scaling of the speedup of the HU when multiple applications are run in parallel. The HU reduces cache misses for non-hash intensive benchmarks and increase the performance of these applications.

- Our approach supports multiple applications running hash operations in parallel, without affecting the correctness of any of the applications.

The rest of the chapter is organized as follows. In Section 3.2, we discuss related previous work. Section 3.3 describes our approach, and Section 3.4 presents experimental results. We summarize the chapter in Section 3.5.

3.2 Previous Work

Several prior research efforts has been focused on hash functions implementations in hardware [23, 24, 25, 26]. These hardware hash functions are useful for checking the identity of two copies of large data files on different nodes in the network. Hash functions are heavily used in networking applications, for tasks such as hashing internet protocol (IP) addresses [27] and for message authentication [28]. The SHA-3 hash function has been simulated and implemented on an FPGA in [29] for cryptographic network applications. They proposed a pipelined model to increase the performance of hash operations. There are some attempts to provide a dynamic perfect hash table [30] for embedded devices.

The hash table data structure provides a fast means to perform the lookup operation. A high-throughput online hash table implementation on an FPGA platform has been proposed in [31]. The authors of [31] increased the throughput of network applications using a pipeline architecture for hash operations. Their design has been implemented on an FPGA, using external DRAM for the hash table. A solution for packet processing using a hashing scheme design has been proposed in [32]. The authors propose a set-associative CAM memory hash table for packet processing in an internet router, and simulated this structure in C++ for different hash functions. They also implemented their hash table accesses using multiple hash functions, to reduce the collision list of hashed entries. In [33], the authors proposed an FPGA implementation of an extended bloom filter using a CAM memory module, for fast IP lookup in networking applications. The key idea is to store the collision list of hash entries in a CAM memory module instead of a linked list.

All the above referenced efforts have attempted to *only* accelerate the hashing operation. In contrast, the design of a hardware hash unit and its associated hash table has not been addressed as part of modern microprocessors. There has been no previous efforts, in modern microprocessors, to include a special function unit for hashing (including hash function as well as dedicated memory to store hash table). Therefore, our work stands apart from the previous research by focusing on the design of a hash unit in modern microprocessors, and quantifying the resulting performance gains.

3.3 The HU Microarchitecture Design

In this section, we discuss our proposed hash unit (HU) in modern microprocessors, at the microarchitecture level design. We start with an overview discussion of our HU. Then, we present a brief overview of the hash function (HF). After that, we introduce the hardware structure of the hardware hash table (HT) used in our approach. Next, we discuss the overall functionality of the HU. After that, we discuss the latency of our hash table implementation, along with cache and main memory latencies. Then, we discuss the hash lookups distribution. After that, we discuss the HU replacement operation. Finally, we discuss the benchmarks used for quantifying the usefulness of the HU.

3.3.1 Overview: Hash Unit (HU)

The hash unit (HU) is designed to accelerate the hash table operations in a modern microprocessor. The HU consists of two blocks – the hash function (HF) block and the hash table (HT) as shown in Figure 3.1. The figure shows the usual CPU pipeline flow, starting from the instruction fetch stage to the instruction issue stage. The HF and the HT are added in the execution stage of the pipeline.

The role of the HF is to perform hash insert, lookup, and delete instructions. The HT is a memory module that holds key and value pairs. The HT performs fast hash operations since it is implemented as a CAM [34, 35]. We implement one CAM per bin of the HT. We will discuss the HF in Section 3.3.2 and the HT in Section 3.3.3.

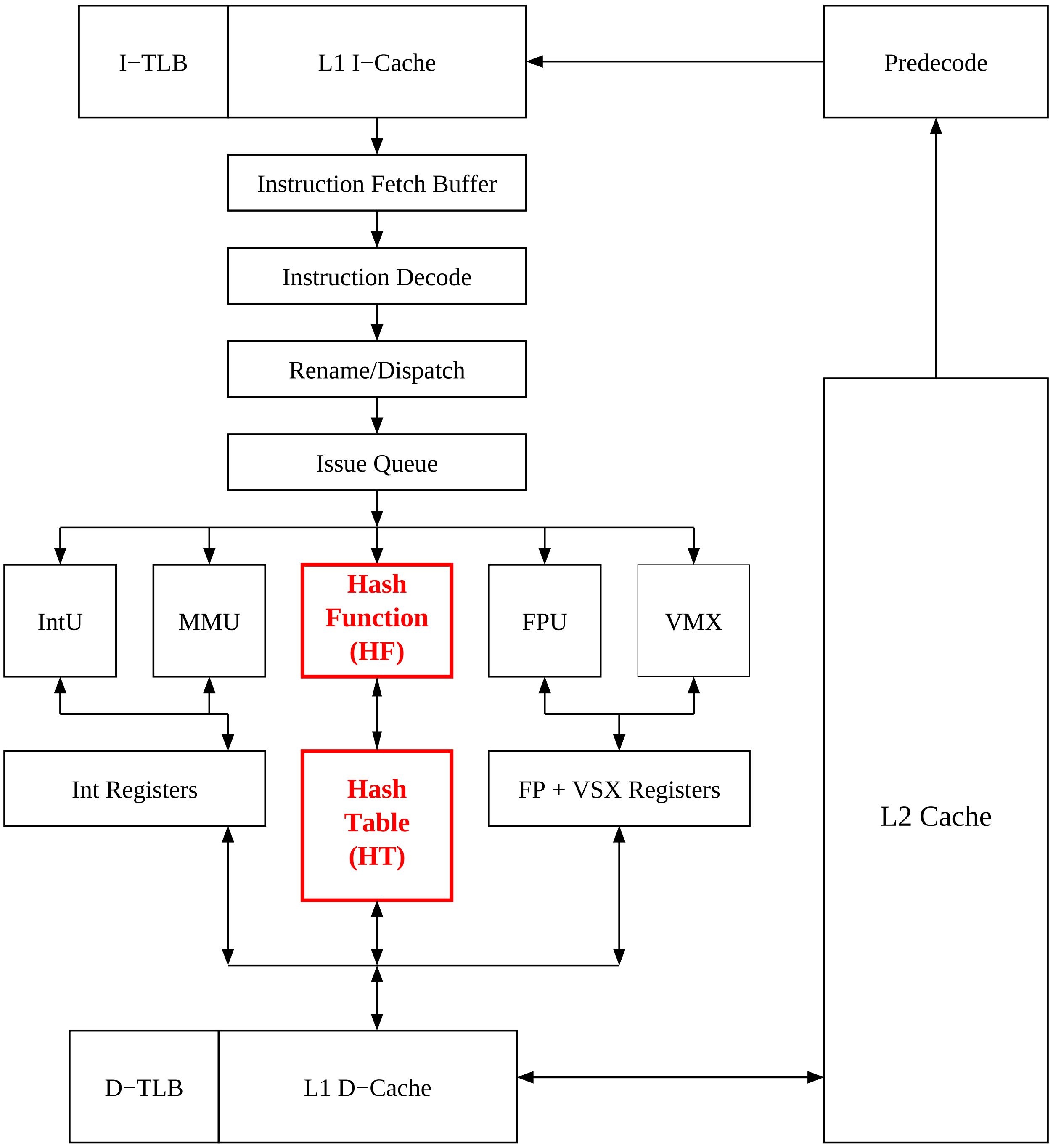


Figure 3.1: Proposed Hash Unit (HU) for Modern Microprocessors (Modified [3])

3.3.2 Hash Function (HF)

The hash function performs a hash operation based on the key of a (key, value) pair. The HF takes a key as an input to produce a bin index as shown on the left side of Figure 3.2. The hash value points to the intended bin number in the HT. Hash functions vary in complexity. Our main purpose is to reduce the number of cycles to perform hash operation, therefore, we select a hash function that performs integer operations. This is easily generalized to perform hashes on pointers. Our hash function is of class H3 [36, 5]. H3 utilizes bit-wise AND and bit-wise XOR operations only. Thus, it yields a fast hash function operation which completes well within one clock cycle. The associated logic in such hash functions is minimal, and adding support for hashing pointers or strings is applicable.

3.3.3 Hash Table (HT) Configuration

In order to provide fast memory access, we design our HT using CAM memory blocks [37, 38]. Each bin in the HT is realized as a CAM, CAMs provide fast memory access, allowing a one-cycle lookup of a HT bin. We use a CAM to store the contents of the HT bin in order to enable searching of the whole bin in parallel, yielding fast hashing operation times. As mentioned in Section 3.3.2, the bin index produced by the HF will point to a bin in the HT. If an entire bin is full, the next available entry in the insert operation will go to DRAM, to effectively extend the bin. Each bin consists of control registers and entry memory module (these store key and value pairs). The HT bin holds the entries corresponding to the collision chain as shown in Figure 3.2. The fields of each bin are as follows:

- PID: This is a register that stores a process ID of the running process that owns this bin of the HT. This register enables the HU to support multiple processes that are simultaneously performing hash operations, to run in parallel and avail of the HU functionality.
- Bin Pointer: This is a register that stores the pointer of the HT bin in DRAM. The entire contents of the HT are stored in DRAM, and hence we have to keep track of DRAM location of each bin of the HT, in order to maintain the consistency of the data entries. Each bin of

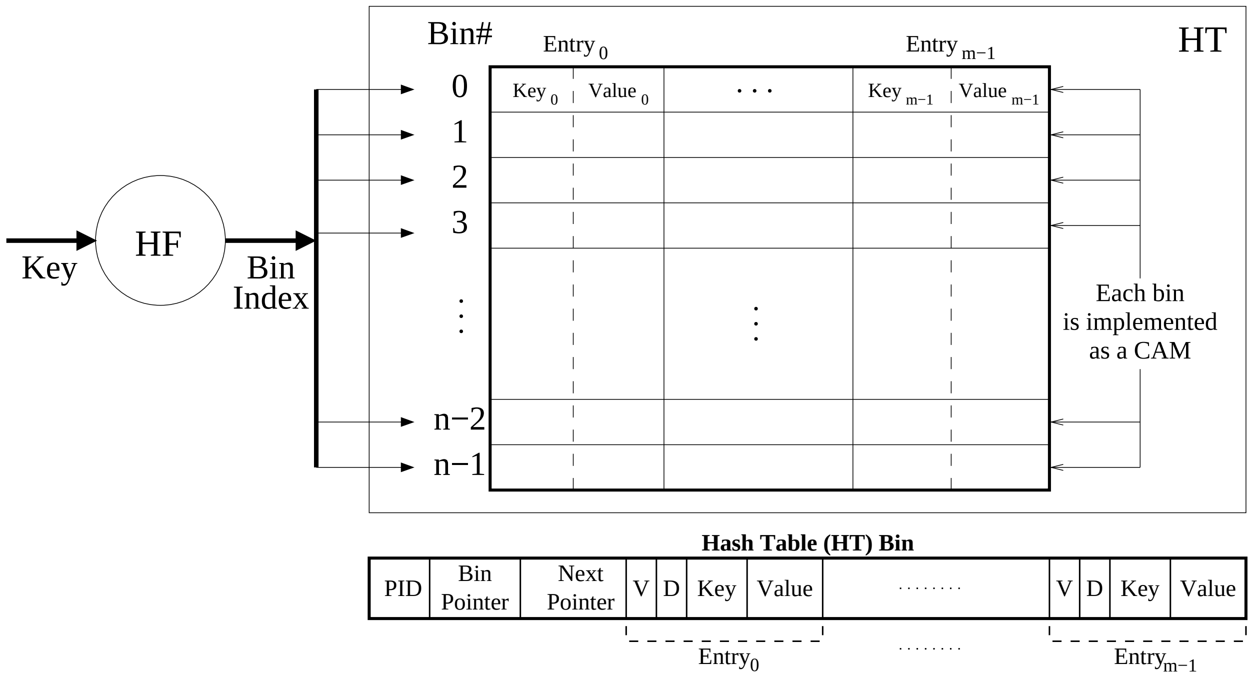


Figure 3.2: HF, HT, and HT Bin Configuration (Modified [3])

the HT can be replaced, in real-time by another bin belonging to a different application that is simultaneously performing hash operations.

- Next Pointer: This is a register that stores the pointer to the first entry of the extended bin in DRAM. In general, if there are m entries per bin, and the bin pointer address is A, then the next pointer address will be $A + m$, where A is the DRAM address for the start of this bin. This pointer helps reducing the latency of accessing the extended bin in DRAM, by skipping any entries in DRAM that are already stored in the HT bin.

- Key: This is a CAM cell that holds the key used as an input to the HF. The key is a unique value, and uniqueness in maintained by the insert operation, which first checks membership before insertion, and does not perform insertion if the key already exists.

- Value: This is an SRAM cell that holds the value corresponding to each key. It is not necessary that values are unique.

- Valid bit (V): This is a bit that represents the validity of the (key, value) pair in each entry in the HT bin. If the valid bit is 0, it means that the (key, value) pair entry is empty. Otherwise, the entry is valid and occupied.

- Dirty bit (D): This is a bit that is used to keep the data in HT updated in DRAM. If a (key, value) pair entry is modified in the HT bin and not yet mirrored in DRAM, the dirty bit is set to 1. Once the (key, value) pair copied back to DRAM, the dirty bit is set to 0 again. The updated entry has to be reflected through all cache levels starting from level one data cache (L1D).

During HT initialization, all valid and dirty bits are set to logic '0'. This indicates that the HT is empty.

3.3.4 HU Functionality

The HU performs three major operations: Lookup, Insert, and Delete. These operations are provided by the HU as new instructions in the microprocessor instruction set. Since our HT is constructed using CAMs [34, 35], HT lookups take one clock cycle, unless the bin data does not fit in the HT bin. These operations are described as follows:

- Lookup: This is an important operation of the HU. All other hash operations rely on the lookup operation. After the key is hashed by the HF to produce a bin index, the lookup operation will be performed on the bin whose index matches the bin index. Then the key is looked-up in parallel within the matched bin of the HT (recall that each HT bin is implemented as a CAM). If the key is in the HT bin, then the (key, value) pair is returned. If the key does not exist in the HT bin and the next pointer register is $null$, then the lookup operation would conclude that the key is not found. Otherwise, the next pointer points to the remaining portion of the bin in DRAM, where the lookup operation continues the search. Since the HT is part of the processor's pipeline execution, the different levels of cache are used during this phase of lookup. Thus, many of the remaining bin entries can be cached in one or more of the cache levels.

- Insert: Before an insert, a lookup operation is executed. If the key exists in the HT, the insert operation will be averted. Otherwise, the new (key, value) pair will be added to the HT bin. If the HT bin is fully occupied, the insert operation uses the next pointer register to continue the insert operation at the end of the linked list data structure in DRAM, for the corresponding bin.
- Delete: This operation is preceded by a lookup operation. If the lookup operation succeeds, the delete operation simply sets the valid bit of the corresponding entry of the HT bin to 0. Otherwise, the delete operation is aborted.

3.3.5 Memory Latency

The main hash table data is stored in DRAM. The HT copies an HT bin from DRAM upon context switch between multiple processes that use the HU. Each process has its own process ID (PID). If an HT bin belongs to a hash process (x), and another hash process (y) accesses that bin, then we effectively have an *HT bin miss*. In this case, we request the HT bin that belongs to process (y) from DRAM. This context switch between HT bins incurs a DRAM latency in the worst case, or the latency of one of the cache levels if the HT bin is available in cache.

3.3.6 Hash Lookups Distribution

We implement our hash function of class H3 [5], as discussed in Section 3.3.2. This type of hash function distributes hash input entries uniformly in hash table bins. Therefore, each bin in the hash table has almost equal number of entries. Hash lookups can have a uniform or a non-uniform input distribution. Our hash function guarantees that each bin in the hash table will have the same distribution of hash lookups. Therefore, we conduct a simple experiment in our hash function with uniform and normal distributions of hash lookups.

In our experiment, we applied a 100K hash lookups to our hash function of class H3. The hash function produces a 2-bit *Bin Index* for 4 bins. The range of hash lookups is from 0 to 100. For uniform hash lookups, the distribution of the hash lookups input appears in all 4 bins, as shown in Figure 3.3. For normal hash lookups ($\sigma = 20, \mu = 50$), the normal distribution appears in each

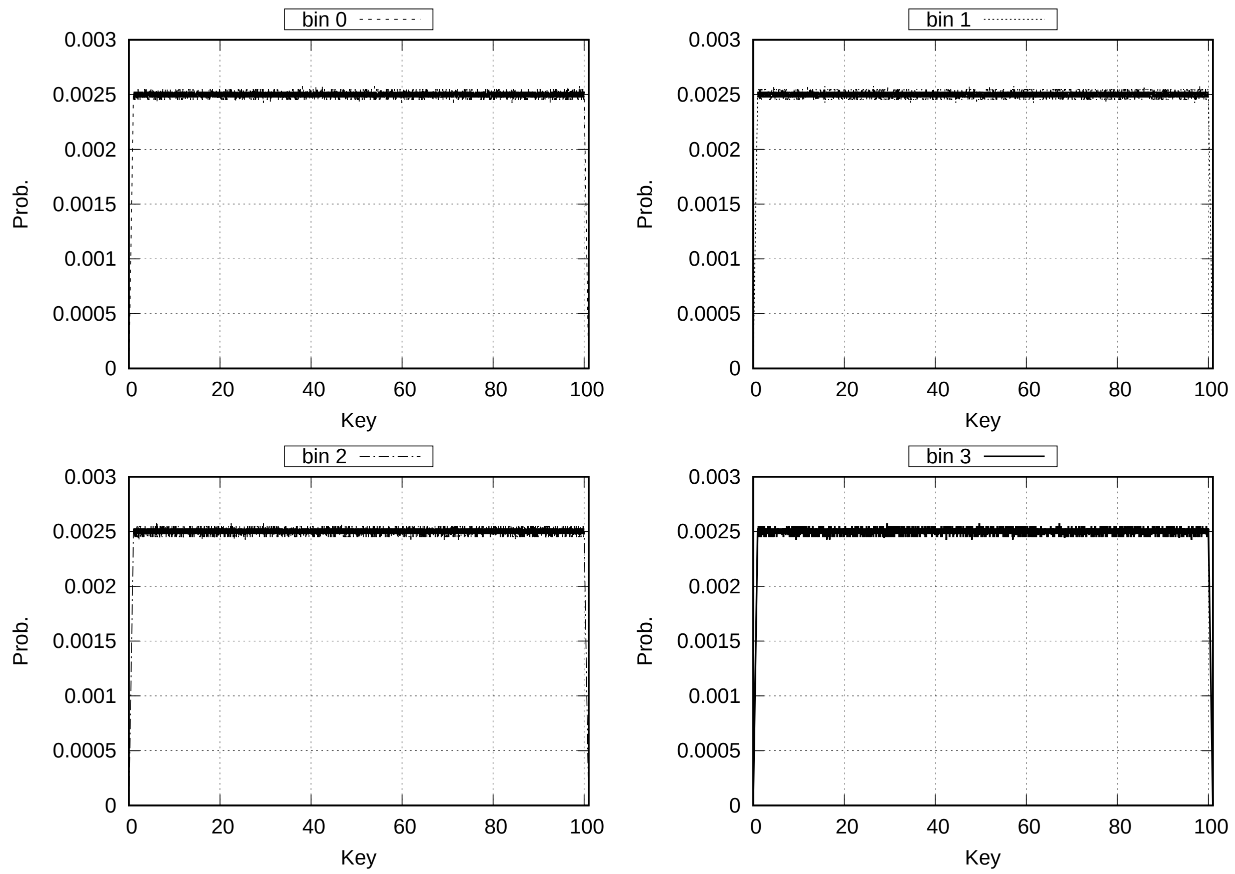


Figure 3.3: Hash table bin entries for Uniform distribution hash lookups

bin, as shown in Figure 3.4.

3.3.7 Replacement Operation in the HU

In a lookup operation in the HU, a replacement operation occurs when there is *a HT miss and a DRAM hit*. In this situation, the hash lookup entries will be placed in the HT, and the rest of the entries will be placed in the DRAM (a golden copy of the hash table entries will be always in the DRAM). As shown in Figure 3.5, the *Bin Pointer* refers to the beginning of the bin in the DRAM, while the *Next Pointer* refer to the first entry stored in the HT. Therefore, a lookup operation in the DRAM starts at the *Bin Pointer* entry, and skips the entries from the *Next Pointer* entry to the *Next Pointer* entry plus $(m - 1)$ entries (where m is the number of entries in the HT bin). In the DRAM, the hash table entries are allocated in blocks of 10kB for each bin, as shown

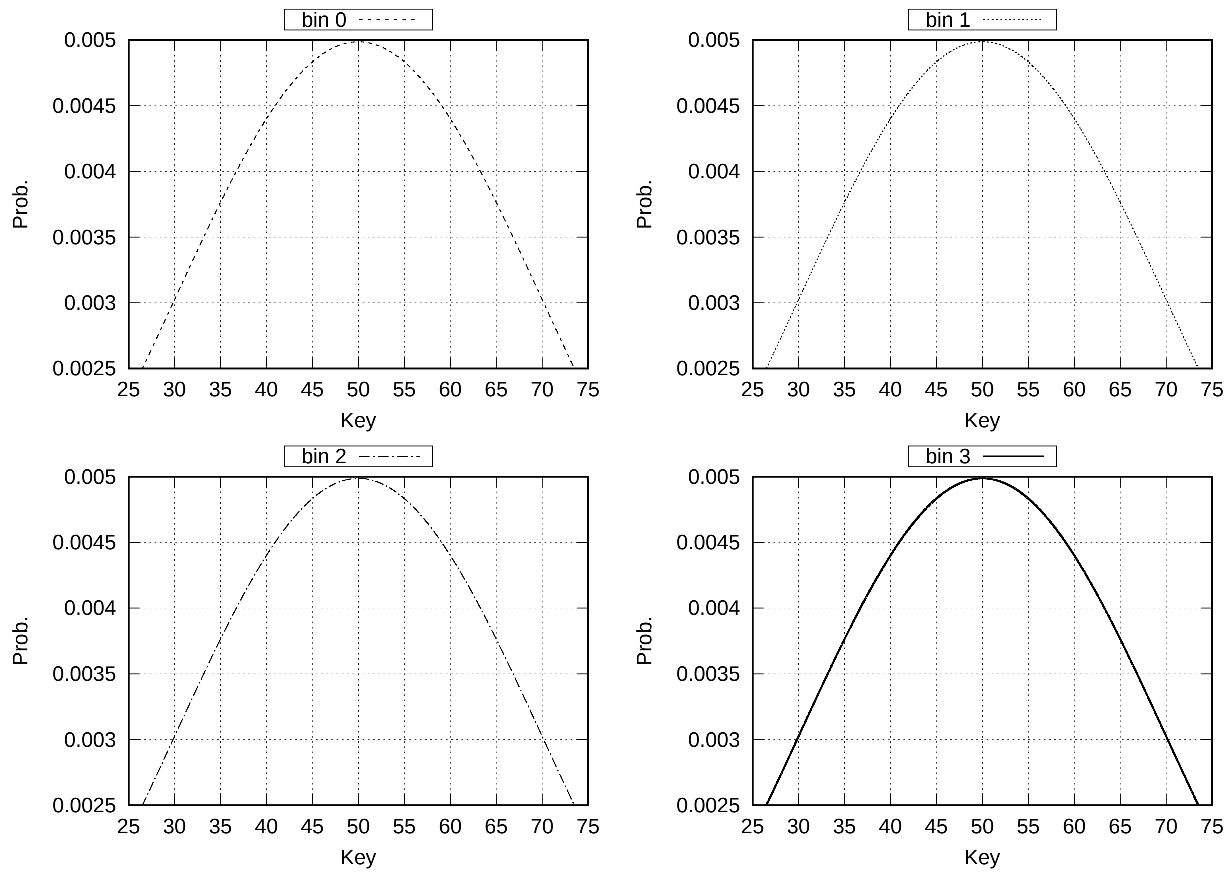


Figure 3.4: Hash table bin entries for Normal distribution hash lookups

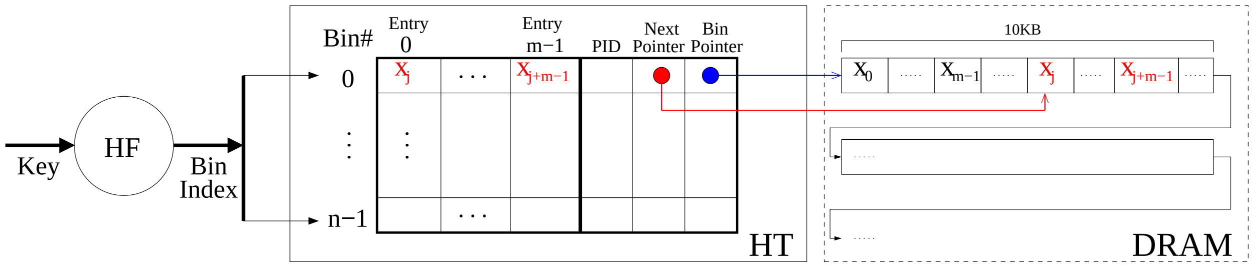


Figure 3.5: Replacement operation in the HU – in case of a HT miss and a DRAM hit of a lookup operation

in Figure 3.5. An Extra 10kB of DRAM memory block will be allocated whenever the current 10kB DRAM memory is fully utilized. In a replacement operation, if the DRAM hit occurs at the first half of the 10kB DRAM block (let's say it occurs at the j^{th} location in the 10kB DRAM block), then the replacement of the entries from the DRAM to the HT will start from the j^{th} location to the $(j + (m - 1))^{th}$ location in the 10kB DRAM block (where j is the DRAM hit entry index). Otherwise, the replacement of the entries from the DRAM to the HT will start from the $(j - (m - 1))^{th}$ location to the j^{th} location in the 10kB DRAM block.

3.3.8 Benchmarks Used

We created three hash benchmarks that provides both insert and lookup operations with random entries:

1. B_{Hy}: we use Yahoo! Cloud Serving Benchmark (YCSB) [39], for hash operations.
2. B_{Hu}: we use uniform distribution for hash operations.
3. B_{Hn}: we use normal distribution for hash operations.

In YCSB benchmark, we use a read only YCSB Core Package workload with a zipfian distribution. In the YCSB workload, we configure each entry in the hash table as a 32-bit key and a 32-bit value (we apply the same *key and value* configurations in B_{Hu} and B_{Hn}). For constructing the hash benchmark B_{Hy}, we use Memcached [40] as a backend in the DRAM to get YCSB workload traces.

The total number of entries is varied in our simulations, but it is chosen to be much larger than the size of the HT. Once the HT has been fully occupied, the rest of the hash table entries reside in the DRAM. This allows us to test how the performance of the HU scales for hash tables that are much larger than the size of the HT. In our experiments, we increase the number of hash table entries up to 5$\times$ the size of the HT. We also study the performance of the HU as the number of parallel instances of B_{Hy}, B_{Hu}, or B_{Hn} is increased. We also use programs from the PARSEC benchmarks suite [41, 10] to test the performance of the HU under a diverse configuration of other applications running in parallel with the B_{Hy}, B_{Hu}, or B_{Hn} benchmarks. We use $simsmall$ simulations in PARSEC benchmarks. We refer to the PARSEC benchmark as B_P. By varying the ratio of the number of B$_{\text{P}}$ instances versus the number of B$_{\text{Hy}}$, B$_{\text{Hu}}$, or B$_{\text{Hn}}$ instances, we can study the HU performance under varying system loads.

The B$_{\text{Hy}}$, B$_{\text{Hu}}$, and B$_{\text{Hn}}$ benchmarks performs only hash operations. In order to create benchmarks in which a user-defined fraction of instructions are hash operations, we modified a Noise-based Boolean Satisfiability (NBSAT) [42] benchmark. In this benchmark, we injected hash instructions, to obtain benchmarks (B$_{\text{HVy}}$, B$_{\text{HVu}}$, and B$_{\text{HVn}}$) with a user-defined fraction of hash instructions. Assume that the original NBSAT benchmark has P instructions, as shown in Fig-

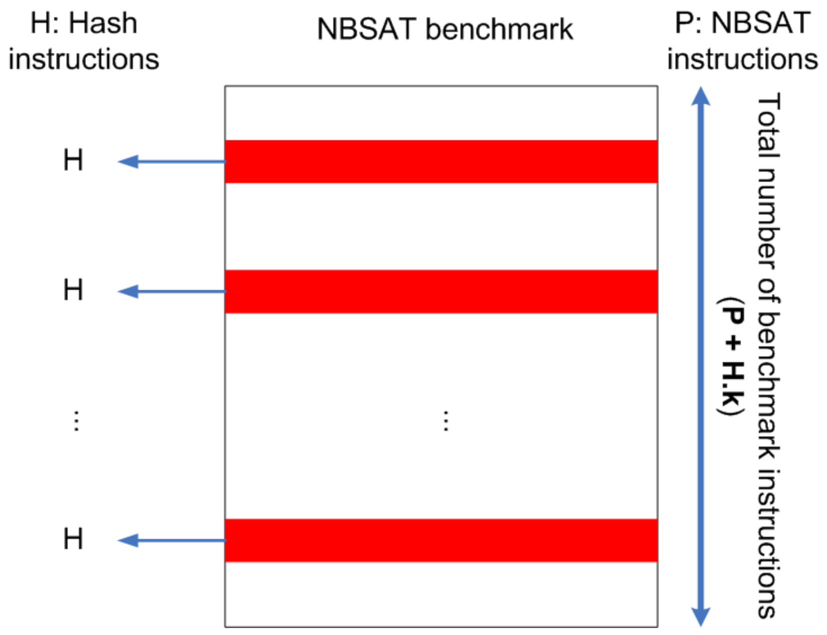


Figure 3.6: Percentage of Hash Instructions in NBSAT Benchmark (Reprinted [3])

ure 3.6. We inject k hash instruction bundles into the NBSAT code, with each hash instruction bundle containing H instructions. Then, the percentage of hash instructions in B$_{\text{HVy}}$, B$_{\text{HVu}}$, and B$_{\text{HVn}}$ (%Hash$_{\text{inst}}$) is shown in Equation 3.1. By varying k, we vary the *density* of hash instructions in B$_{\text{HVy}}$, B$_{\text{HVu}}$, and B$_{\text{HVn}}$, and test the performance of the HU while varying k.

$$\%Hash_{inst} = \frac{H.k}{P + (H.k)} \tag{3.1}$$

In Section 3.4, we will quantify the performance of the HU, and compare it with a software-based hashing approach.

3.4 Experimental Results

In this section, we present our experimental results of the HU for use in modern microprocessors at the microarchitecture level. We discuss our simulation environment in Section 3.4.1. In Section 3.4.2, we discuss the simulation parameters and groups. Finally, we present our simulation results along with a discussion in Section 3.4.3.

3.4.1 Simulation Environment

We implement the HU in GEM5 [22] (a Computer Architecture Simulator). We use x86 ISA as our base instruction set. In case the DRAM is not involved, it is assumed that a lookup operation on the HU takes one cycle. The insert operations takes one cycle if the entry is available in the hash table, otherwise it takes two cycles. The delete operations takes one cycle if the entry is not available in the hash table, otherwise it takes two cycles. We augment the x86 ISA in GEM5 with our hash instructions. In GEM5, we use the O3 detailed CPU model in full system (FS) mode and MOESI_hammer ruby memory model. In all our simulations, we verify the correctness of all hash operations in the HU and the validity of the HT entries. We use uniform distribution (UNIF), normal distribution (NORM) and YCSB workload (YCSB) to create our hash benchmarks: (B_{Hu}, B_{Hn}, and B_{Hy}) and (B_{HVu}, B_{HVn}, and B_{HVy}), as discussed in Section 3.3.8.

The basic configurations of the system and HT in our simulations are as follows:

- CPU: single core 1GHz.
- DRAM: 1GB DDR3 1600MHz x64 (64-bit bus width).
- L1 instruction cache (L1I): 32kB, 64 byte per block size, 8-way set associative, PSEUDO_LRU replacement policy.
- L1 data cache (L1D): 64kB, 64 byte block size, 8-way set associative, PSEUDO_LRU replacement policy.

- L2 cache: 2MB, 64 byte block size, 8-way set associative, PSEUDO_LRU replacement policy.
- L3 cache: 16MB, 64 byte block size, 16-way set associative, PSEUDO_LRU replacement policy.

3.4.2 Simulation Parameters and Groups

In our simulations of the HU, we vary several parameters:

- Number of entries – This is varied from 4K to 80K entries, each entry is a size of 4 bytes.
- CPU speed – This is varied from 1GHz to 5GHz, in steps of 1GHz.
- HT size – This is varied from 8kB to 124kB in our experiments.
- DRAM technology – We perform our experiments on LPDDR3, DDR3, and DDR4 DRAM technologies.
- DRAM latency – We perform our experiments with a DRAM latency of 30ns, 60ns, 90ns, 120ns, and 150ns.
- L1D cache size – In our simulations, we test the HU with an L1D cache of size 32kB, 64kB, and 128kB. These values are based on L1D cache sizes from processors offered by Intel, AMD, and IBM. The L1D cache sizes of Intel processors are 32kB [21], while some AMD Athlon parts have L1D of size 128kB [43]. The IBM Power8 processors use an L1D cache of size 64kB [44].

We partition our simulations into two groups:

1. The first group (G1): the size of (L1D + HT) stays the same as the size of (L1D) that we use in the software-only implementation. This group, in effect, models the scenario where the total CAM area stays fixed when we implement the HU.

2. The second group (G2): the size of (L1D) stays fixed, and in the HU implementation, the size of (L1D + HT) is greater than the size of (L1D) for the software-only implementation. Essentially, G1 assumes that the CPU area is fixed (i.e. area conservative), while G2 relaxes this assumption.

The speedup of the HU is computed by the ratio of number of cycles when hashing is done in software without the HU (SW_{cycles}) versus the number of cycles when hashing is done with the HU (HW_{cycles}), as shown in the following equation:

$$Speedup = \frac{SW_{cycles}}{HW_{cycles}} \tag{3.2}$$

We refer to a simulation with the HU as HW_{hash}, while SW_{hash} refer to a software-only simulation (using a software-based hashing).

For the HT, 64kB results in 16K entries (i.e. key-value pairs), since each key is 4 bytes long.

3.4.3 Results and Analysis

In this section, we present and analyze our experimental results. We divide our experiments into four major subsections: uniform distribution (UNIF), normal distribution (NORM), HU replacement, and YCSB benchmark (YCSB). First, we present the results for uniform distribution hash operations experiments. Then, we present the results for normal distribution hash operations experiments. After that, we compare the uniform and normal distribution experiments with/without HU replacement. Finally, we present and discuss the results of our hash operations experiments using YCSB benchmark.

3.4.3.1 Uniform Distribution (UNIF)

In this section, we use the uniform distribution to generate our hash benchmarks B_{Hu} and B_{HVu} for our experiments, without a replacement operation between the HT and the DRAM. We will discuss the results using the replacement operation later in Section 3.4.3.3. The experiments in this section will be in the following order:

1. Vary CPU speed.
2. Vary the HT size.
3. Vary the main memory (DRAM) technology.
4. Vary the DRAM latency.
5. Run multiple hash benchmark (B_{Hu}) instances in parallel with multiple PARSEC benchmarks (B_P).
6. Run multiple B_{Hu} instances alone.
7. Vary a fraction of hash instructions in the benchmark B_{HVu}.

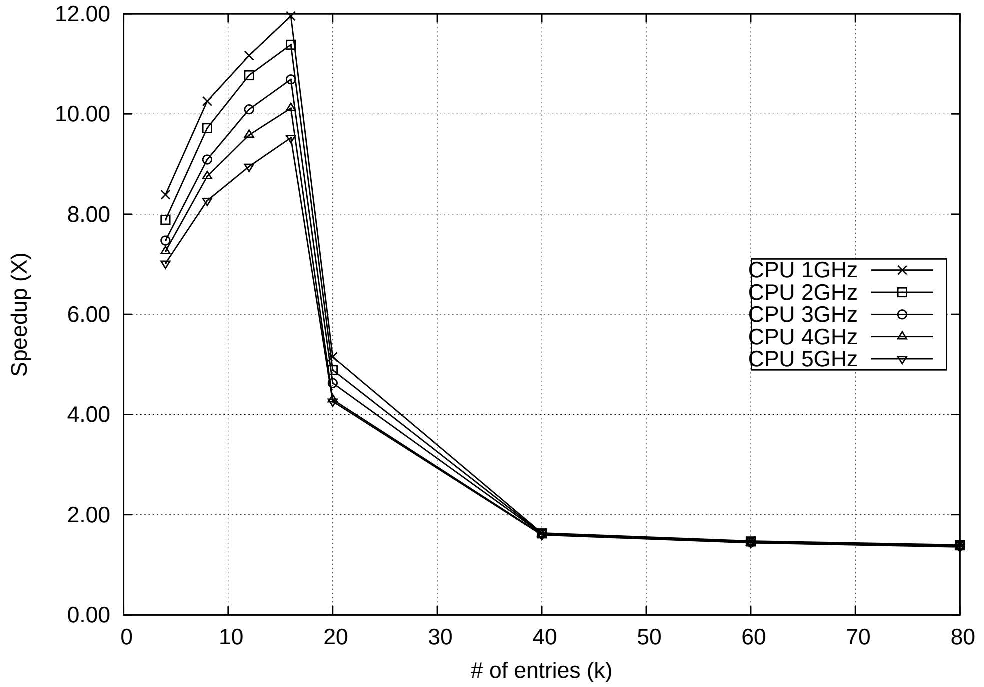


Figure 3.7: (UNIF) Speedup as CPU Speed is Varied – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. DRAM: DDR3 1600MHz x64. (Reprinted [3])

The CPU speed plays an important role for SW_{hash}. In Figure 3.7, we vary the CPU speed to study its effect on the performance. The application run is a single instance of B_{Hu}, and the simulation is of type G1. We note that the peak speedup is between 9.5$\times$ and 12$\times$. The speedup is maximum when the entire hash table fits in the HT (i.e. when the hash table has $\sim$16K entries). As the CPU speed increases, the relative speedup of HW_{hash} is reduced, since SW_{hash} can perform its operations faster (with fixed HU operation times).

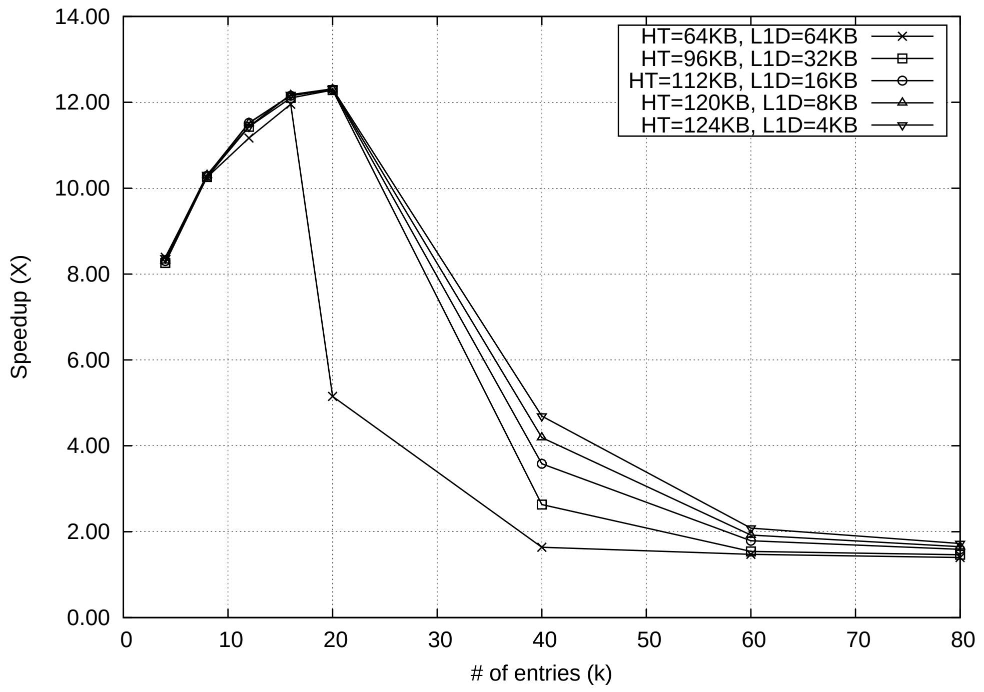


Figure 3.8: (UNIF) Tradeoff of HT and L1D Sizes – SW_{hash}: L1D=128kB. DRAM: 1GB DDR3 1600MHz x64. CPU=1GHz. (Reprinted [3])

Figure 3.8 illustrates the tradeoff between the size of the HT and the L1D size, for a simulation of type G1. The L1D size for the software-only implementation is 128kB, and for the HU implementation, size (L1D + HT) is fixed at 128kB. We note that the peak speedup is about 12.3$\times$. The speedup of hash operations in the HU implementation increases as the HT size increased up to 124kB. Since the L1D size gets progressively smaller as the HT size increases, it reduces the speed of the non-hash instructions. Therefore, the *sweet spot* is when the HT size is the same as the L1D size, or slightly greater.

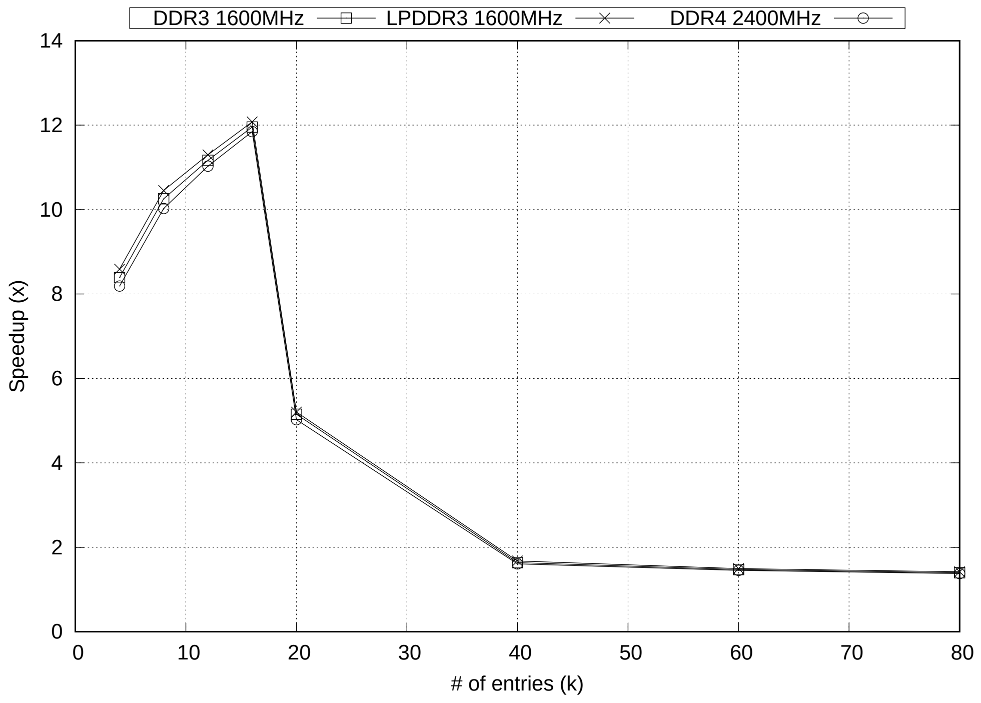


Figure 3.9: (UNIF) Vary Memory Types – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz.

Figure 3.9 illustrates the effect of main memory (DRAM) technologies on the HU implementation. There is a minimal effect on the speedup of the HU implementation, as the DRAM technology changes, since the DRAM is utilized in both HU and software-based implementations. As a result, both implementations benefit from the performance of the DRAM technology.

To illustrate the effect of caching and DRAM latency on the HU performance, we turned off caching in Figure 3.10. Of course this would not be done in practice, but we did this experiment to check the contributions of caching and DRAM latency. We note that without L1D cache, the HU provides peak speedups between 40$\times$ and 87$\times$ for a single instance of B_{Hu}, This speedup is substantially inversely proportional to DRAM latency.

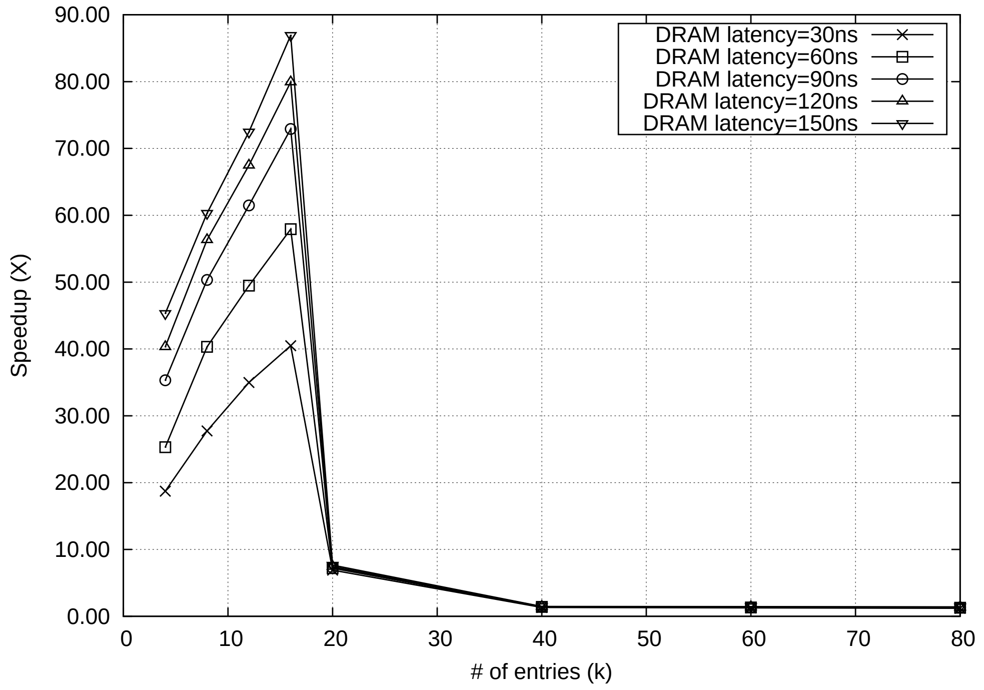


Figure 3.10: (UNIF) Vary DRAM Latency – No Caches. HW_{hash}: HT=64kB, L1D=0kB. SW_{hash}: L1D=0kB. CPU=1GHz. (Reprinted [3])

The experiments so far were run for a single instance of B_{Hu}. In the experiments that follow, we report the performance for a diverse computational load, with multiple B_{Hu} and B_P processes running in parallel. This would result in cache and HT *pollution*, providing a more realistic idea of the value of the HU. For all these experiments, we compare the time required to complete the entire set of applications, and compare the speedup of HW_{hash} over SW_{hash}. In these experiments, we run several instances of our hash benchmark (B_{Hu}) along with several PARSEC benchmarks (B_P), to observe the effect of context switches of a total of 10 benchmarks, we vary the number of B_{Hu} and B_P instances as shown in Table 3.1. The percent of hash benchmarks is therefore the number of B_{Hu} instances divided by 10. This value varies between 10% and 90%. As the percentage of hash benchmarks increases, the performance of the hash benchmarks increases as shown in Figures (3.11,3.12). Figure 3.11, shows the G2 configuration group, while Figure 3.12, shows the G1 results. For both Figures 3.11 and 3.12, for each plot, we note that the speedup

Percentage of Hash benchmarks (%)	PARSEC benchmarks (B_P)									Number of instances of (B_{Hu}, B_{Hn}, or B_{Hy})
	bodytrack	canneal	dedup	facesim	ferret	fluidanimate	freqmine	vips	x264	
10	✓	✓	✓	✓	✓	✓	✓	✓	✓	1
20	✓	✓	✓	✓	✓	✓	✓	✓		2
30	✓	✓	✓	✓	✓	✓	✓			3
40	✓	✓	✓	✓	✓	✓				4
50	✓	✓	✓	✓	✓					5
60	✓	✓	✓	✓						6
70	✓	✓	✓							7
80	✓	✓								8
90	✓									9

Table 3.1: PARSEC [10] Benchmarks Utilization in Parallel with Hash Benchmarks B_{Hu}, B_{Hn}, or B_{Hy} (Reprinted [3])

increases as the HT size increases, and also as the fraction of hash benchmarks increases. Also for each figure, the speedup is higher when the total amount of CAM memory is higher. The speedups for configuration G2 (Figure 3.11) is higher than for configuration G1 (Figure 3.12), as expected, since the total amount of CAM in G2 is larger. The speedup reaches up to 5.1×, in Figure 3.11, when size(HT)=64kB and size(L1D)=128kB, and running 9 instances of B_{Hu}.

Figures 3.13 and 3.14 report the results from our experiments to quantify the effect of sharing the hardware HT among several B_{Hu} instances. We observe each G2 plot achieves a greater speedup than the corresponding G1 plot, as expected. For each of the 6 plots in Figures 3.13 and 3.14, the speedup is slightly higher for a larger size(HT). In each of the 6 plots, the speedup is highest when there are fewer instances of B_{Hu} contending for the HT resource. When 9 instances of B_{Hu} are running, the speedup for all 6 plots is ranges from 50% to 10%. The 50% speedup is obtained for size(HT)=64kB and size(L1D)=128kB (Figure 3.13). For a single instance of B_{Hu}, the speedup is as high as 13.3× (Figure 3.13), in which size(HT)=64kB and size(L1D)=128kB. For up to 3 instances of B_{Hu}, in both plots, the speedup is higher than 80%, reducing gradually as the number

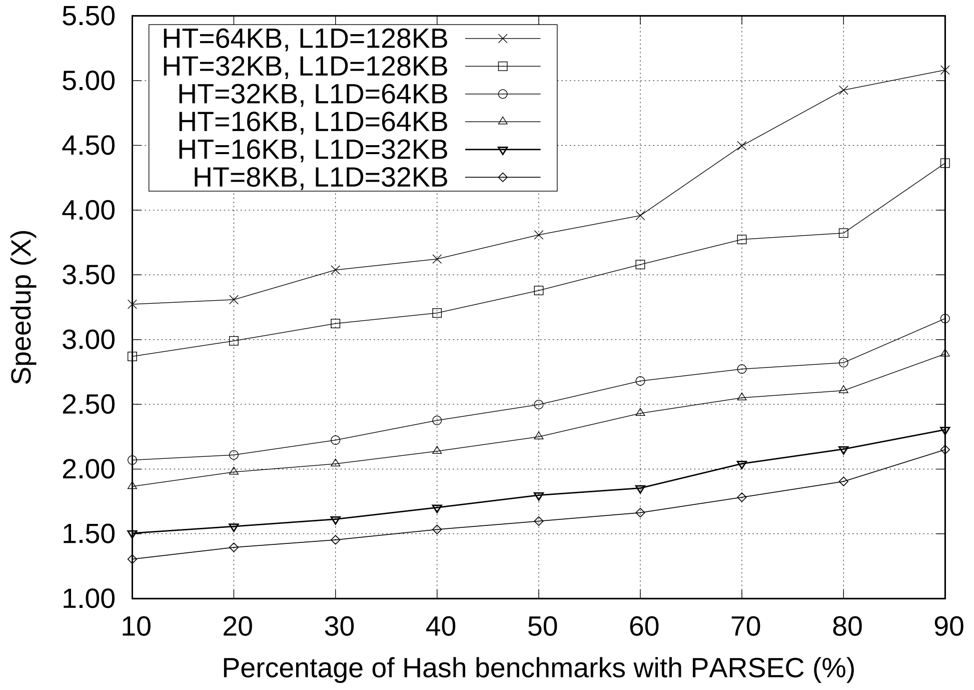


Figure 3.11: (UNIF) Hash and PARSEC Benchmarks in Parallel (B_{Hu} and B_P) – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash}) (Modified [3])

of B_{Hu} instances increases.

Finally, we perform an experiment to test the speedup due to the HU, for a single benchmark, as the fraction of hash instructions in the benchmark ($\%Hash_{inst}$) varies, as discussed in Section 3.3.8 and illustrated in Figure 3.6. The benchmark we chose was NBSAT, and the results are shown in Figures 3.15 and 3.16. We note that the speedup of the single NBSAT instance (with injected hash instructions) increases as $\%Hash_{inst}$ increases, in all 6 plots of Figures 3.15 and 3.16. For each plot in Figure 3.15, the corresponding plot of Figure 3.16 exhibits greater speedup, as expected. These plots suggest that for all the HT and L1D sizes studied, including the HU enhances the performance of any application, even if it has a relatively small fraction of hash operations. For a $\%Hash_{inst}$ value of 10%, the speedup ranges between 22% and 6%. A speedup value of 18% is obtained for size(HT)=64kB and size(L1D)=64kB (Figure 3.16).

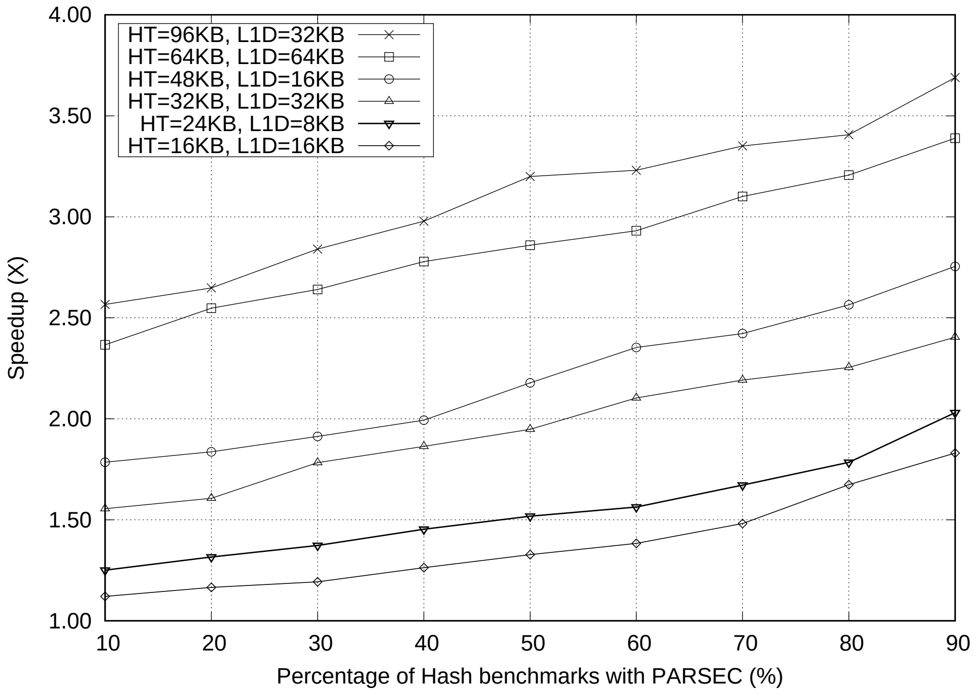


Figure 3.12: (UNIF) Hash and PARSEC Benchmarks in Parallel –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash}) (Modified [3])

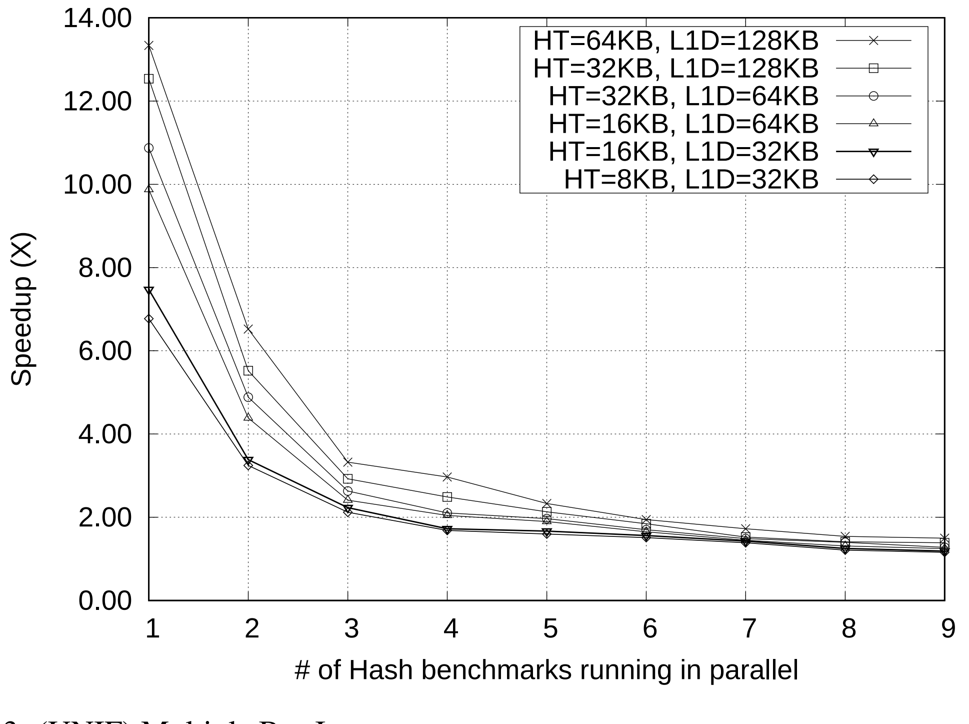


Figure 3.13: (UNIF) Multiple B_{Hu} Instances –
Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash}) (Modified [3])

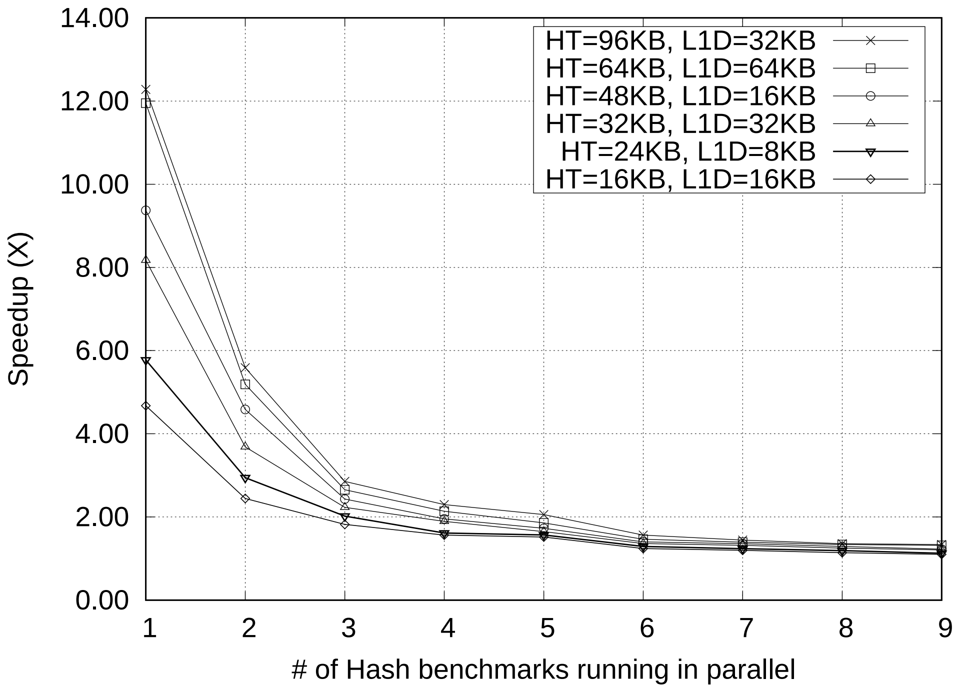


Figure 3.14: (UNIF) Multiple B_{Hu} Instances –
Simulation Type G1 (size(HT+L1D) in HW_{hash} = size of (L1D) in SW_{hash}) (Modified [3])

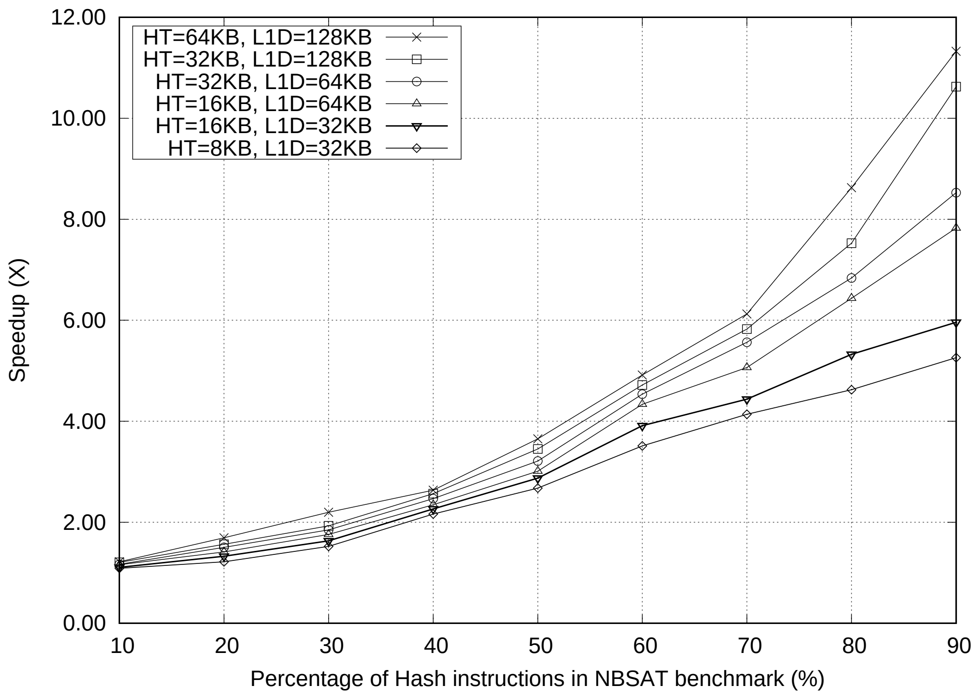


Figure 3.15: (UNIF) NBSAT Benchmark with $\%Hash_{inst}$ Varying –
Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash}) (Modified [3])

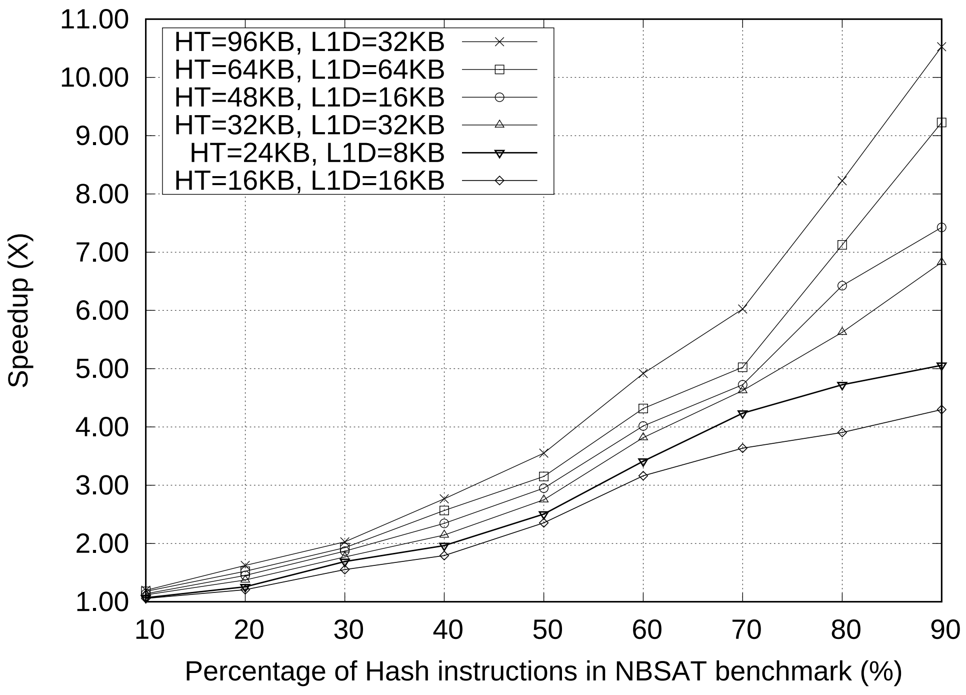


Figure 3.16: (UNIF) NBSAT Benchmark with %$\text{Hash}_{\text{inst}}$ Varying –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash}) (Modified [3])

3.4.3.2 Normal Distribution (NORM)

In this section, we use the normal distribution to generate our hash benchmarks $\mathrm{B_{Hn}}$ and $\mathrm{B_{HVn}}$ for our experiments, without a replacement operation between the HT and the DRAM. For the normal distribution hash lookup operations, we use: $Range = [1 : 100k]$, $\mu = 50k$, and $\sigma = 20k$. The experiments in this section will be in the following order:

1. Vary CPU speed.

2. Vary the HT size.

3. Vary the main memory (DRAM) technology.

4. Vary the DRAM latency.

5. Run multiple hash benchmark ($\mathrm{B_{Hn}}$) instances in parallel with multiple PARSEC benchmarks ($\mathrm{B_P}$).

6. Run multiple $\mathrm{B_{Hn}}$ instances alone.

7. Vary a fraction of hash instructions in the benchmark $\mathrm{B_{HVn}}$.

In Figure 3.17, we vary the CPU speed to study its effect on the performance. The application run is a single instance of $\mathrm{B_{Hn}}$, and the simulation is of type G1. We note that the peak speedup is between 9.7$\times$ and 11.2$\times$. The speedup is maximum when the entire hash table fits in the HT (i.e. when the hash table has $\sim$16K entries). As CPU speed increases, the relative speedup of $\mathrm{HW_{hash}}$ is reduced, since $\mathrm{SW_{hash}}$ can perform its operations faster (with fixed HU operation times).

Figure 3.18 illustrates the tradeoff between the size of the HT and the L1D size, for a simulation of type G1. The L1D size for the software-only implementation is 128kB, and for the HU implementation, size (L1D + HT) is fixed at 128kB. We note that the peak speedup is about 11.76$\times$. The speedup of hash operations in the HU implementation increases as the HT size increased up to 124kB. Since the L1D size gets progressively smaller as the HT size increases, it reduces the

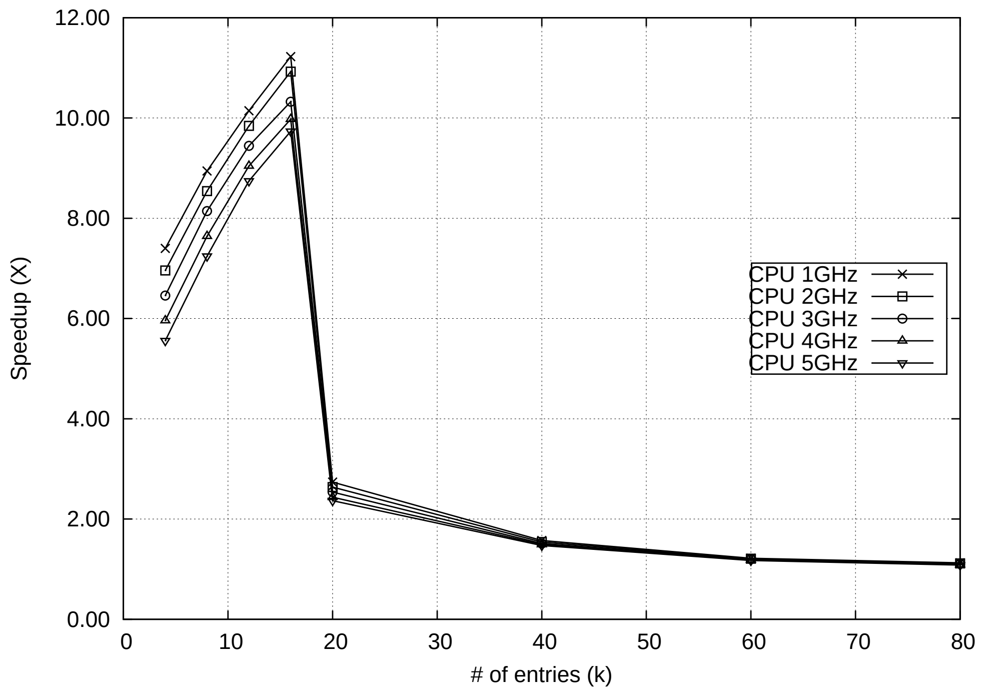


Figure 3.17: (NORM) Speedup as CPU Speed is Varied – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. DRAM: DDR3 1600MHz x64.

speed of the non-hash instructions. Therefore, the *sweet spot* is when the HT size is the same as the L1D size, or slightly greater.

Figure 3.19 illustrates the effect of main memory (DRAM) technologies on the HU implementation. There is a minimal effect on the speedup of the HU implementation, as the DRAM technology changes, since the DRAM is utilized in both HU and software-based implementations. As a result, both implementations benefit from the performance of the DRAM technology.

To illustrate the effect of caching and DRAM latency on the HU performance, we turned off caching in Figure 3.20. Of course this would not be done in practice, but we did this experiment to check the contributions of caching and DRAM latency. We note that without L1D cache, the HU provides speedups between $60\times$ and $77\times$ for a single instance of B_{Hn}, This speedup is substantially inversely proportional to DRAM latency.

The experiments so far were run for a single instance of B_{Hn}. In the experiments that follow,

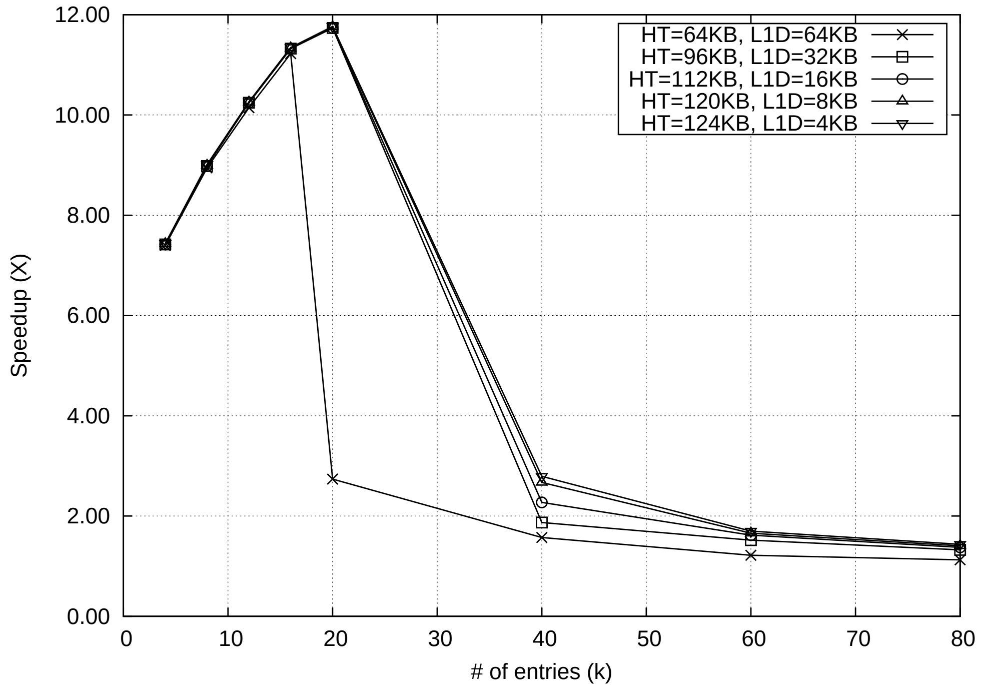


Figure 3.18: (NORM) Tradeoff of HT and L1D Sizes – SW_{hash}: L1D=128kB. DRAM: 1GB DDR3 1600MHz x64. CPU=1GHz.

we report the performance for a diverse computational load, with multiple B_{Hn} and B_P processes running in parallel. As we mentioned in Section 3.4.3.1, this would result in cache and HT *pollution*, providing a more realistic idea of the value of the HU. For all these experiments, we compare the time required to complete the entire set of applications, and compare the speedup of HW_{hash} over SW_{hash}. In these experiments, we run several instances of our hash benchmark (B_{Hn}) along with several PARSEC benchmarks (B_P), to observe the effect of context switches of a total of 10 benchmarks, we vary the number of B_{Hn} and B_P instances as shown in Table 3.1. The percent of hash benchmarks is therefore the number of B_{Hn} instances divided by 10. This value varies between 10% and 90%. As the percentage of hash benchmarks increases, the performance of the hash benchmarks increases as shown in Figures 3.21 and 3.22. Figure 3.21 shows the G2 configuration group, while Figure 3.22 shows the G1 results. For both Figures 3.22 and 3.22, for each plot, we note that the speedup increases as the HT size increases, and also as the fraction of hash

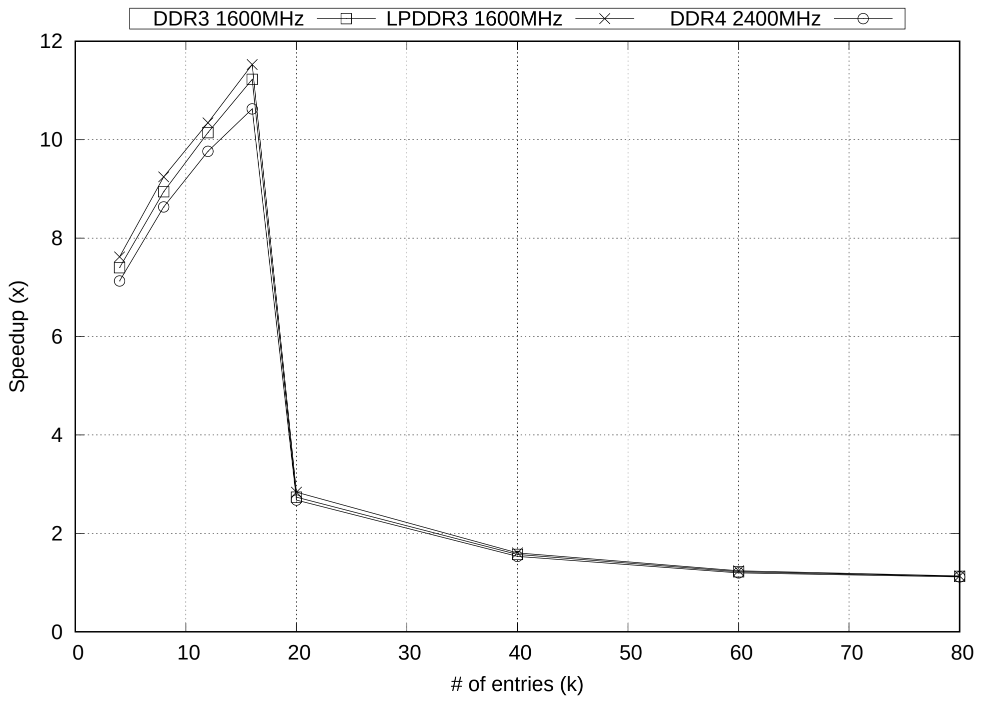


Figure 3.19: (NORM) Vary Memory Types – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz.

benchmarks increases. Also for each figure, the speedup is higher when the total amount of CAM memory is higher. The speedups for configuration G2 (Figure 3.21) is higher than for configuration G1 (Figure 3.22), as expected, since the total amount of CAM in G2 is larger. The speedup reaches up to 4.6×, in Figure 3.21, when size(HT)=64kB and size(L1D)=128kB, and running 9 instances of B_{Hn}.

Figures 3.23 and 3.24 report the results from our experiments to quantify the effect of sharing hardware HT among several B_{Hn} instances. We observe each G2 plot achieves a greater speedup than the corresponding G1 plot, as expected. For each of the 6 plot in Figures 3.23 and 3.24, the speedup is slightly higher for a larger size(HT). In each of the 6 plots, the speedup is highest when there are fewer instances of B_{Hn} contending for the HT resource. When 9 instances of B_{Hn} are running, the speedup for all 6 plots is ranges from 43% to 10%. The 43% speedup is obtained for size(HT)=64kB and size(L1D)=128kB (Figure 3.23). For a single instance of B_{Hn}, the speedup

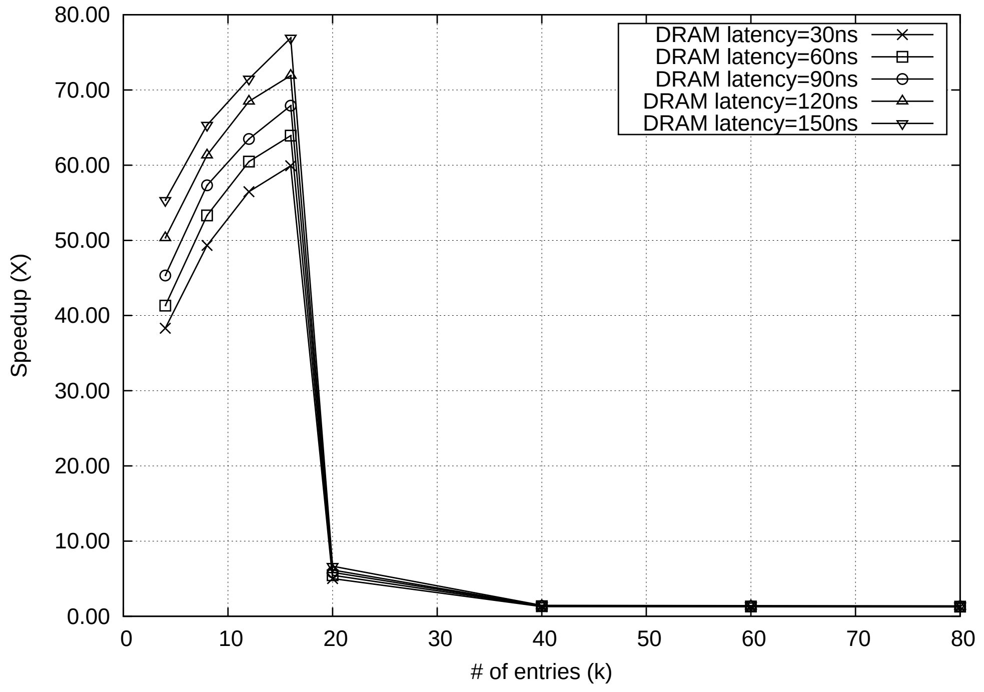


Figure 3.20: (NORM) Vary DRAM Latency – No Caches. HW_{hash}: HT=64kB, L1D=0kB. SW_{hash}: L1D=0kB. CPU=1GHz.

is as high as 11.4× (Figure 3.23), in which size(HT)=64kB and size(L1D)=128kB. For up to 3 instances of B_{Hn}, in all 6 plots, the speedup is higher than 62%, reducing gradually as the number of B_{Hn} instances increases.

Finally, we perform an experiment to test the speedup due to the HU, for a single benchmark, as the fraction of hash instructions in the benchmark ($\%Hash_{inst}$) varies, as discussed in Section 3.3.8 and illustrated in Figure 3.6. The benchmark we chose was NBSAT, and the results are shown in Figures 3.25 and 3.26. We note that the speedup of the single NBSAT instance (with injected hash instructions) increases as $\%Hash_{inst}$ increases, in all 6 plots of Figures 3.25 and 3.26. For each plot in Figure 3.25, the corresponding plot of Figure 3.26 exhibits greater speedup, as expected. These plots suggest that for all the HT and L1D sizes studied, including the HU enhances the performance of any application, even if it has a relatively small fraction of hash operations. For a $\%Hash_{inst}$ value of 10%, the speedup ranges between 20% and 4%. A speedup value of 16% is

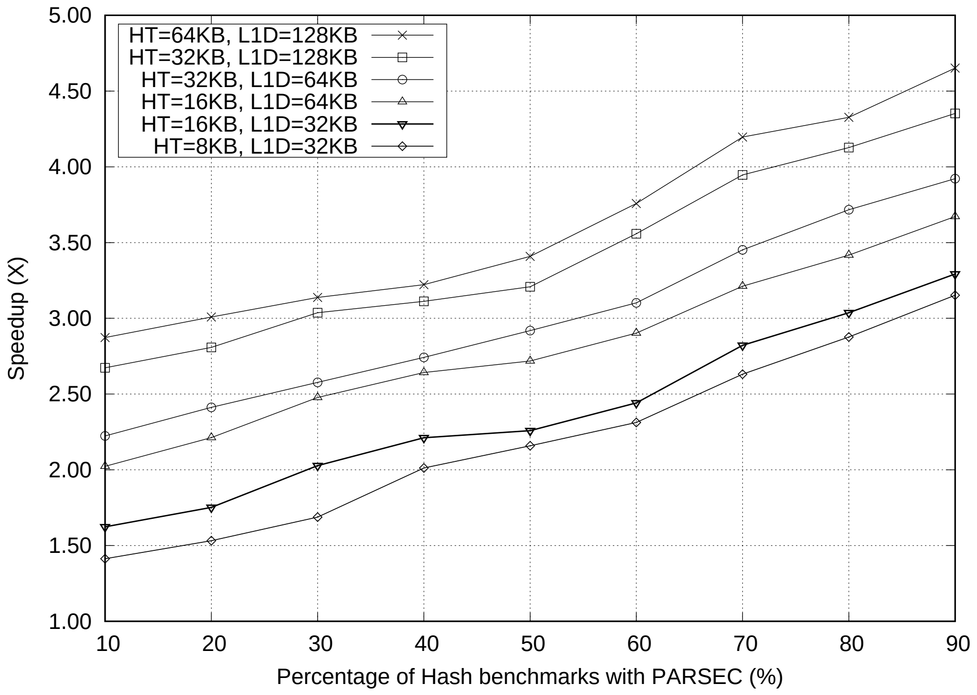


Figure 3.21: (NORM) Hash and PARSEC Benchmarks in Parallel (B_{Hn} and B_P) – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

obtained for size(HT)=64kB and size(L1D)=64kB (Figure 3.26).

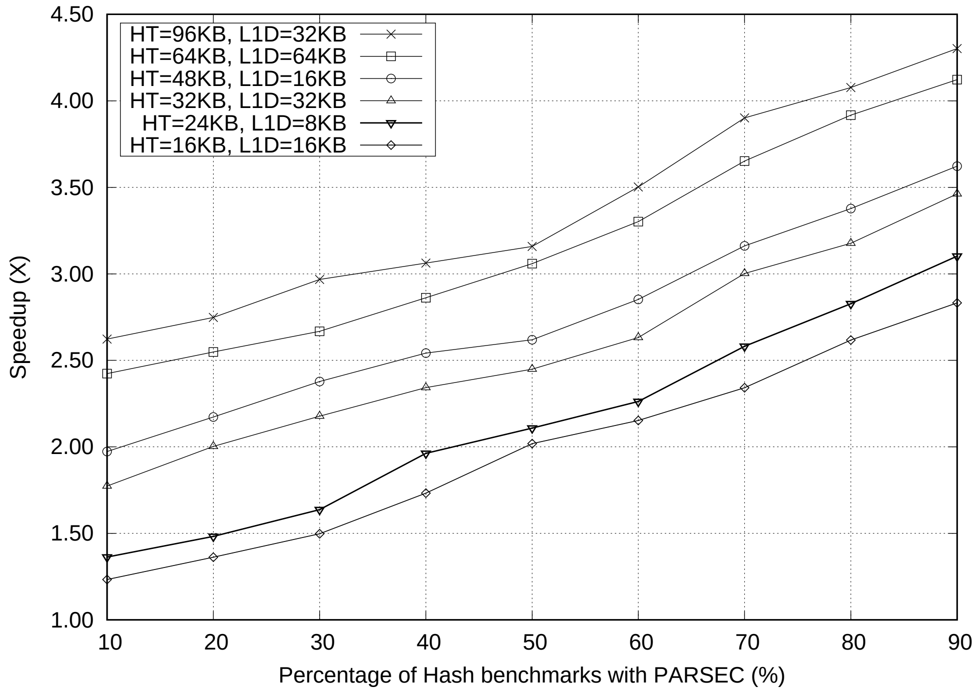


Figure 3.22: (NORM) Hash and PARSEC Benchmarks in Parallel –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

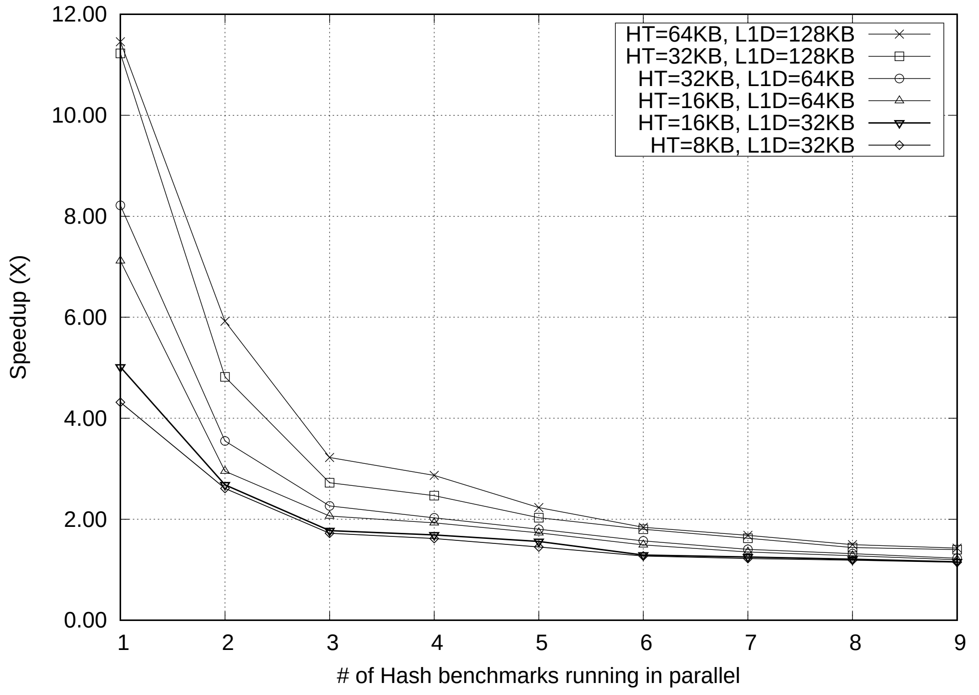


Figure 3.23: (NORM) Multiple B_{Hn} Instances –
Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

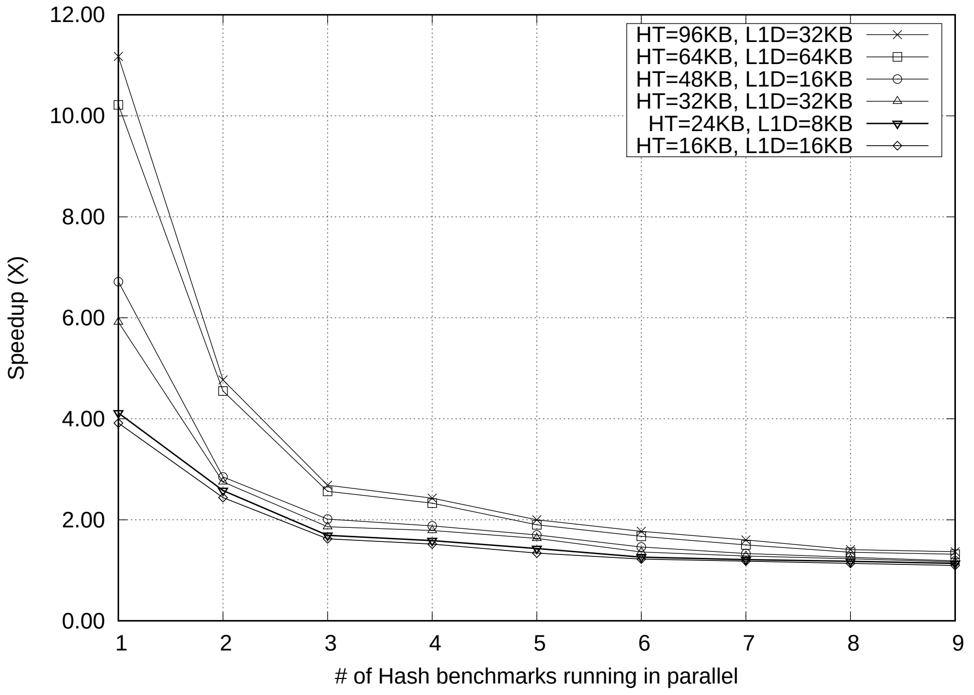


Figure 3.24: (NORM) Multiple B_{Hn} Instances –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

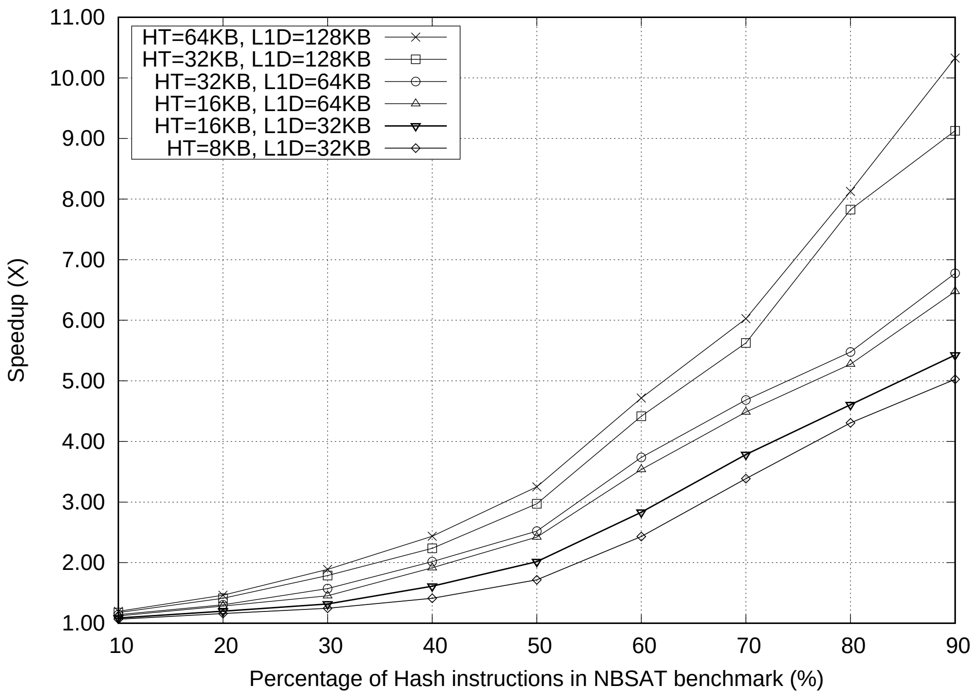


Figure 3.25: (NORM) NBSAT Benchmark with $\%Hash_{inst}$ Varying –
Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

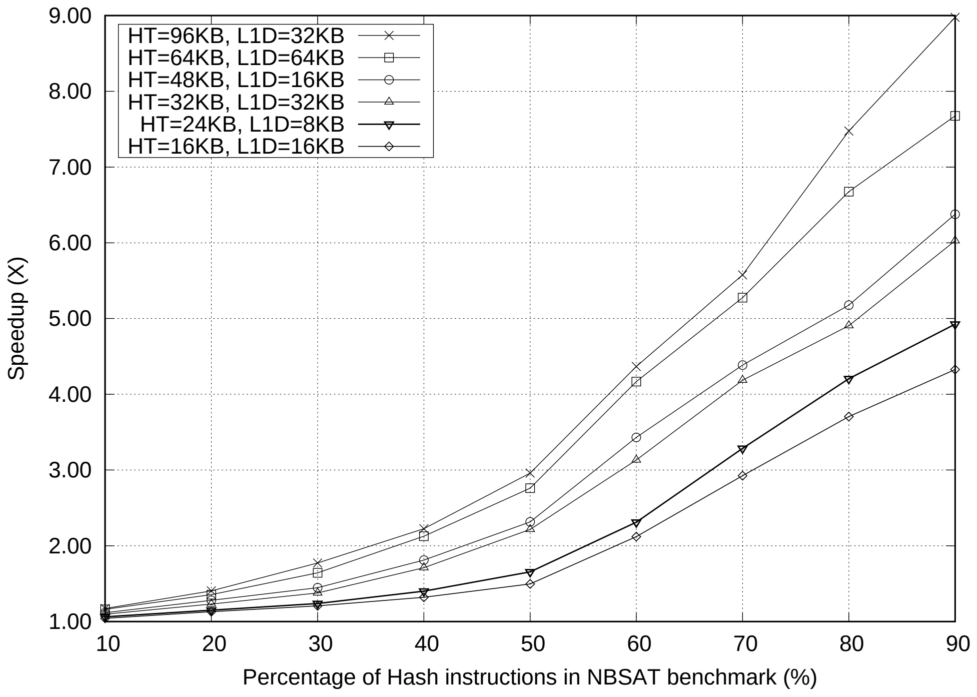


Figure 3.26: (NORM) NBSAT Benchmark with %$\text{Hash}_{\text{inst}}$ Varying –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

3.4.3.3 HU Replacement

In this section, we compare the speedup of the HU for the uniform distribution and the normal distribution of the hash lookup operations, with and without HU replacement.

First, we compare the use of the HU replacement in the uniform distribution of hash lookup operations. As shown in Figure 3.27, the HU without replacement has a better speedup than the

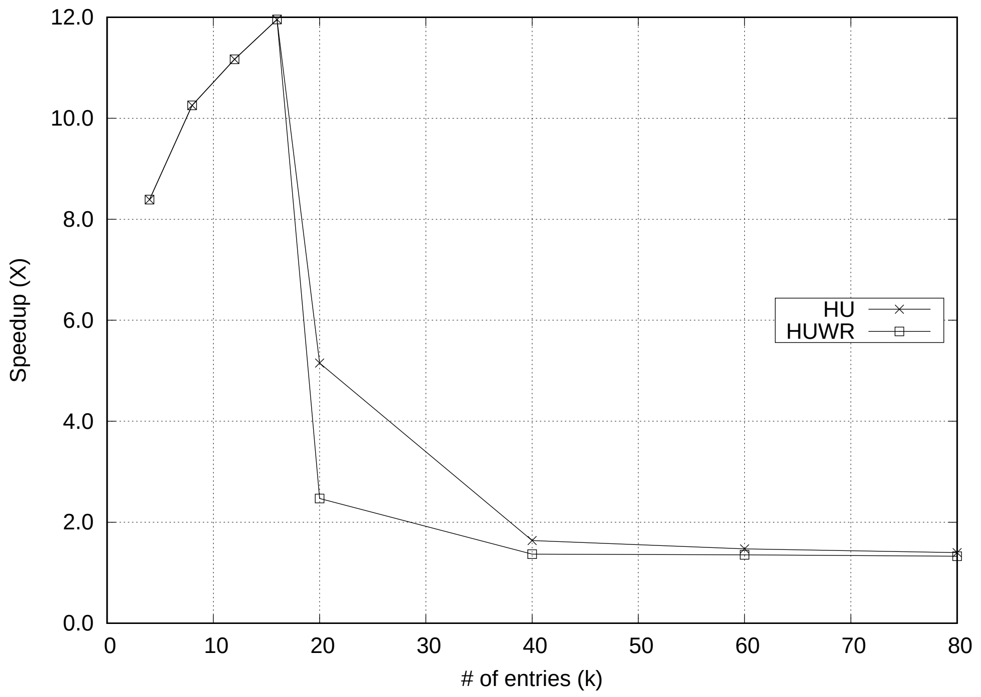


Figure 3.27: HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz. DRAM: DDR3 1600MHz x64.

HU with replacement. The reason is that it has an extra overhead for replacing entries back-and-forth between the HT and the DRAM. As a result, for uniform distribution hash lookup operations, the HU *without* replacement is a better implementation than the HU with replacement.

Second, we compare the use of the HU replacement in the normal distribution of hash lookup operations. In the normal distribution, we use a $Range = [1 : 100k]$ and $\mu = 50k$. We vary the σ in the normal distribution. As shown in Figures (3.28, 3.29, and 3.30), the HU with replacement

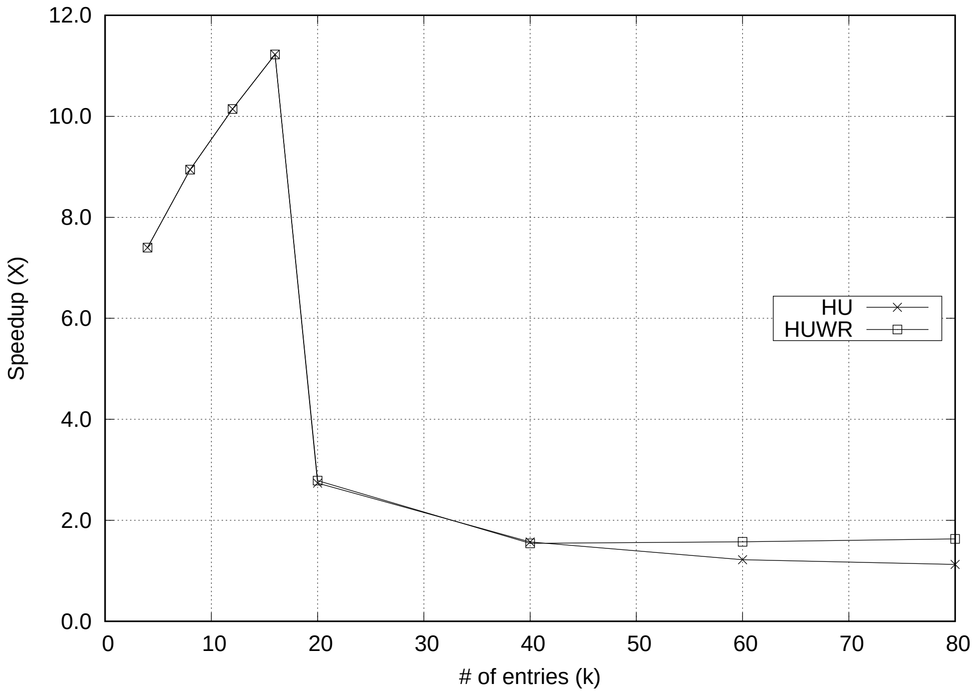


Figure 3.28: ($\sigma = 12k$) HW$_{\text{hash}}$: HT=64kB, L1D=64kB. SW$_{\text{hash}}$: L1D=128kB. CPU=1GHz. DRAM: DDR3 1600MHz x64.

has a better speedup than the HU without replacement. The reason is the HU replacement brings the highly lookup entries in the normal distribution into the HT, and increasing the HT lookup hits. As a result, for normal distribution hash lookup operations, the HU *with* replacement is a better implementation than the HU without replacement.

In the next section, we will study the performance of the HU (with HU replacement) using YCSB benchmark [39].

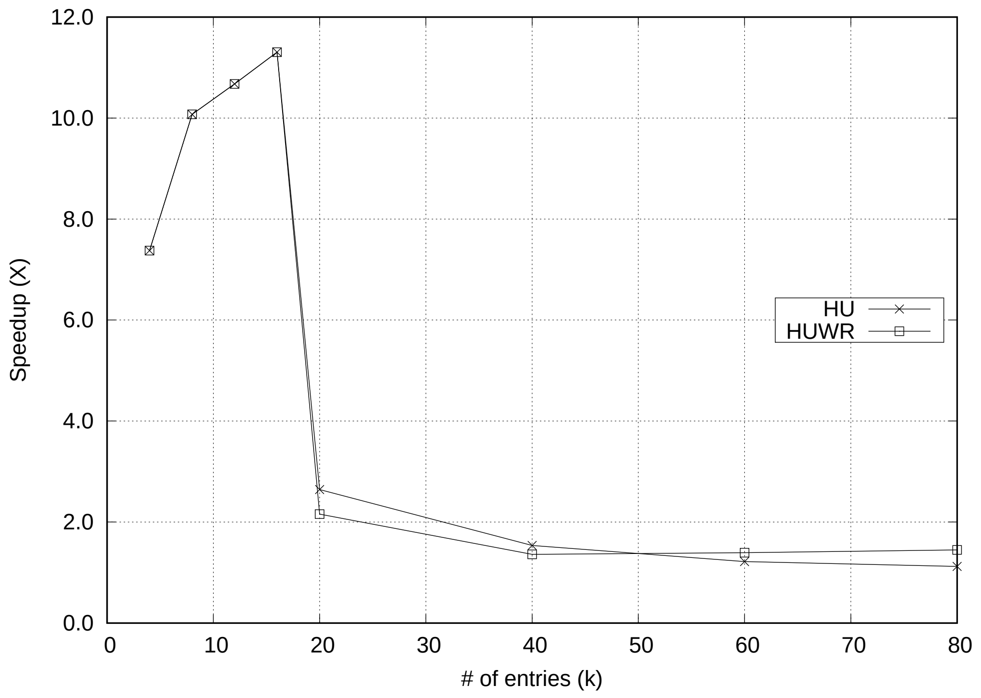


Figure 3.29: ($\sigma = 16k$) HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz. DRAM: DDR3 1600MHz x64.

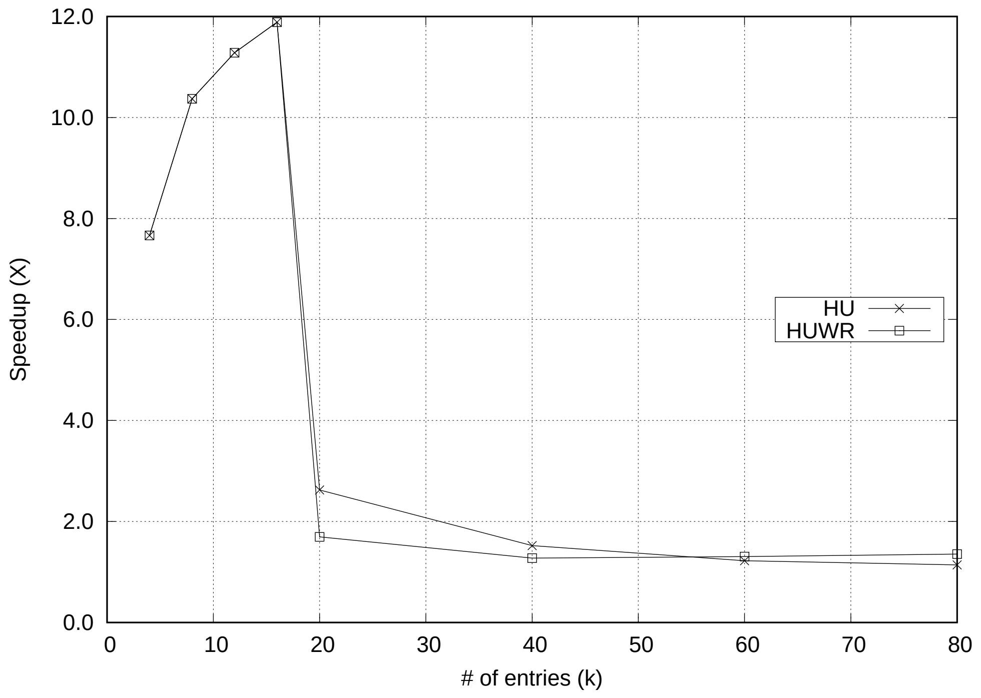


Figure 3.30: ($\sigma = 20k$) HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz. DRAM: DDR3 1600MHz x64.

3.4.3.4 YCSB benchmark (YCSB)

In this section, we use YCSB benchmark [39] to generate our hash benchmarks B_{Hy} and B_{HVy}. As we mentioned in the Section 3.4.3.3, the HU replacement increases the performance of hash operations for normally distributed hash lookups. Therefore, for the experiments in this section, we use a HU replacement between the HT and the DRAM. The experiments in this section will be in the following order:

1. Vary CPU speed.
2. Vary the HT size.
3. Vary the main memory (DRAM) technology.
4. Vary the DRAM latency.
5. Run multiple hash benchmark (B_{Hy}) instances in parallel with multiple PARSEC benchmarks (B_P).
6. Run multiple B_{Hy} instances alone.
7. Vary a fraction of hash instructions in the benchmark B_{HVy}.

In Figure 3.31, we vary the CPU speed to study its effect on the performance. The application run is a single instance of B_{Hy}, and the simulation is of type G1. We note that the peak speedup is between 10.8$\times$ and 11.8$\times$. The speedup is maximum when the entire hash table fits in the HT (i.e. when the hash table has $\sim$16K entries). As CPU speed increases, the relative speedup of HW_{hash} is reduced, since SW_{hash} can perform its operations faster (with fixed HU operation times).

Figure 3.32 illustrates the tradeoff between the size of the HT and the L1D size, for a simulation of type G1. The L1D size for the software-only implementation is 128kB, and for the HU implementation, size (L1D + HT) is fixed at 128kB. We note that the peak speedup is about 15$\times$. The speedup of hash operations in the HU implementation increases as the HT size increased up to 124kB. Since the L1D size gets progressively smaller as the HT size increases, it reduces the

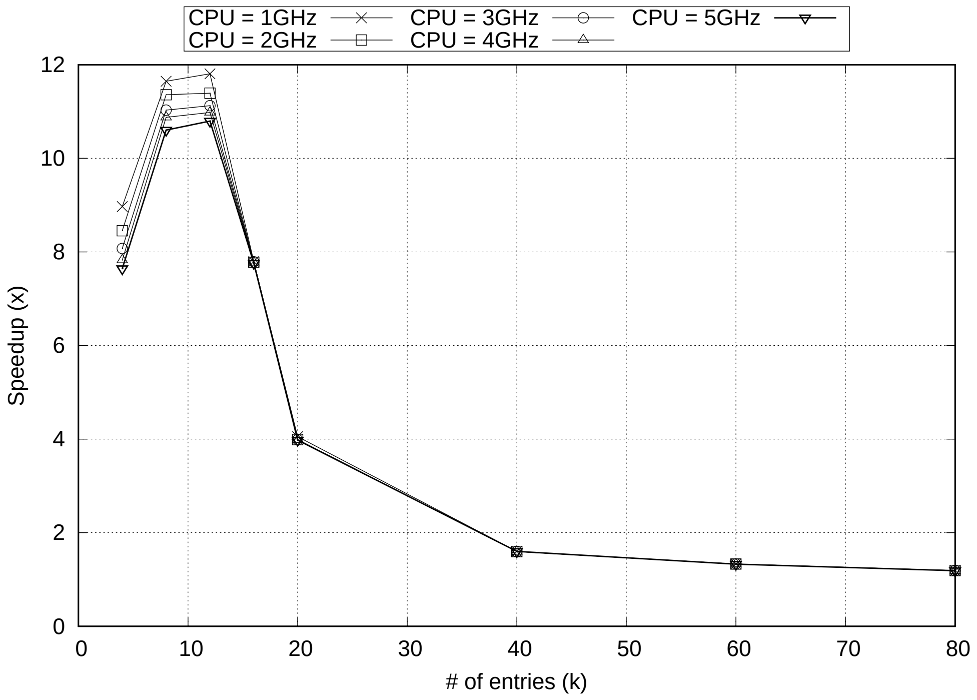


Figure 3.31: (YCSB) Speedup as CPU Speed is Varied – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. DRAM: DDR3 1600MHz x64.

speed of the non-hash instructions. Therefore, the *sweet spot* is when the HT size is the same as the L1D size, or slightly greater.

Figure 3.33 illustrates the effect of main memory (DRAM) technologies on the HU implementation. There is a minimal effect on the speedup of the HU implementation, as the DRAM technology changes, since the DRAM is utilized in both HU and software-based implementations. As a result, both implementations benefit from the performance of the DRAM technology.

To illustrate the effect of caching and DRAM latency on the HU performance, we turned off caching in Figure 3.34. Of course this would not be done in practice, but we did this experiment to check the contributions of caching and DRAM latency. We note that without L1D cache, the HU provides speedups between 36$\times$ and 61$\times$ for a single instance of B_{Hy}, This speedup is substantially inversely proportional to DRAM latency.

The experiments so far were run for a single instance of B_{Hy}. In the experiments that follow,

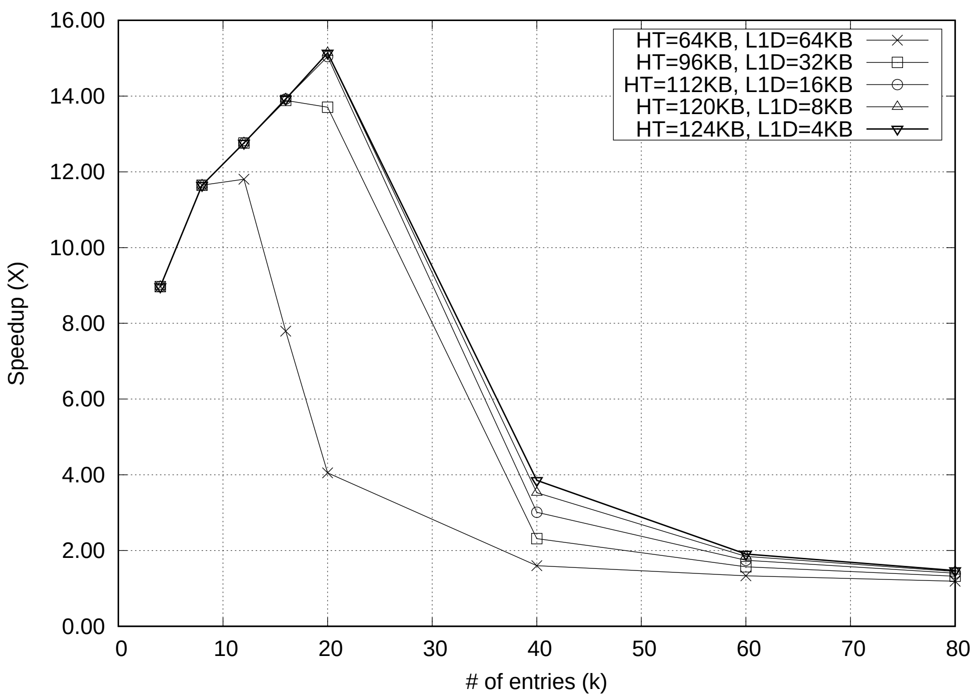


Figure 3.32: (YCSB) Tradeoff of HT and L1D Sizes – SW_{hash}: L1D=128kB. DRAM: 1GB DDR3 1600MHz x64. CPU=1GHz.

we report the performance for a diverse computational load, with multiple B_{Hy} and B_P processes running in parallel. As we mentioned in Section 3.4.3.1, this would result in cache and HT *pollution*, providing a more realistic idea of the value of the HU. For all these experiments, we compare the time required to complete the entire set of applications, and compare the speedup of HW_{hash} over SW_{hash}. In these experiments, we run several instances of our hash benchmark (B_{Hy}) along with several PARSEC benchmarks (B_P), to observe the effect of context switches of a total of 10 benchmarks, we vary the number of B_{Hy} and B_P instances as shown in Table 3.1. The percent of hash benchmarks is therefore the number of B_{Hy} instances divided by 10. This value varies between 10% and 90%. As the percentage of hash benchmarks increases, the performance of the hash benchmarks increases as shown in Figures 3.35 and 3.36. Figure 3.35 shows the G2 configuration group, while Figures 3.36 shows the G1 results. For for each plot in both Figures 3.35 and 3.36, we note that the speedup increases as the HT size increases, and also as the fraction of hash

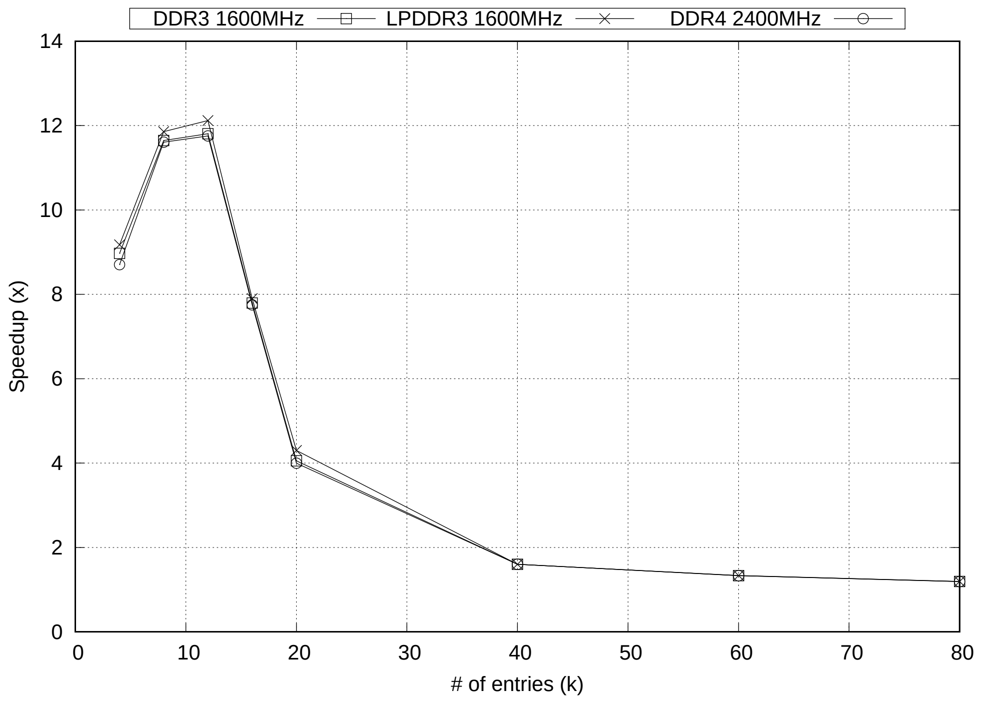


Figure 3.33: (YCSB) Vary Memory Types – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz.

benchmarks increases. Also for each figure, the speedup is higher when the total amount of CAM memory is higher. The speedups for configuration G2 (Figure 3.35) is higher than for configuration G1 (Figure 3.36), as expected, since the total amount of CAM in G2 is larger. The speedup reaches up to 5.7×, in Figure 3.35, when size(HT)=64kB and size(L1D)=128kB, and running 9 instances of B_{Hy}. The hash benchmarks running purely in software incur more L1D cache misses, since they share the L1D cache with other PARSEC benchmarks. The HU reduces the L1D cache misses (and accesses) as shown in Figure 3.37. In Figure 3.37, the first 3 bars show the cache misses for 9 PARSEC benchmarks, for varying L1D sizes. As expected, the cache misses increase as the L1D size reduces. When this application set is run with the HU (HW_{hash}), the cache misses are reported in the fourth bar. In the fifth bar, we simulate the same 9 PARSEC benchmarks along with one instance of B_{Hy}, in the software-only (SW_{hash}). Note that the HU reduces the cache misses by about 2×.

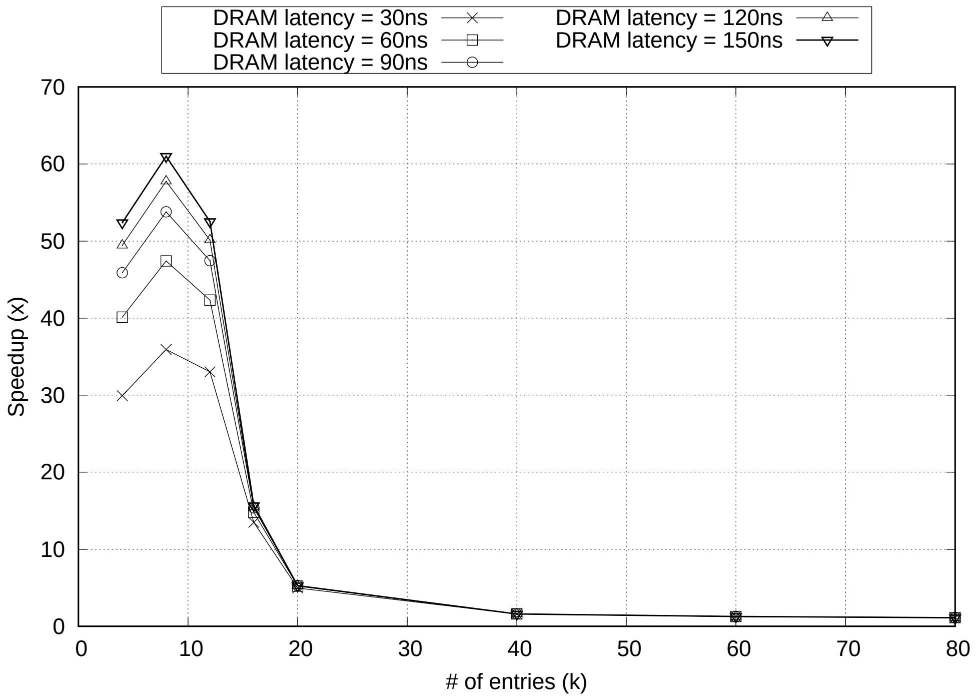


Figure 3.34: (YCSB) Vary DRAM Latency – No Caches. HW_{hash}: HT=64kB, L1D=0kB. SW_{hash}: L1D=0kB. CPU=1GHz.

Figures 3.38 and 3.39 report the results from our experiments to quantify the effect of sharing hardware HT among several B_{Hy} instances. We observe each G2 plot achieves a greater speedup than the corresponding G1 plot, as expected. For each of the 6 plots in Figures 3.38 and 3.39, the speedup is slightly higher for a larger size(HT). In each of the 6 plots, the speedup is highest when there are fewer instances of B_{Hy} contending for the HT resource. When 9 instances of B_{Hy} are running, the speedup for all 6 plots is ranges from 4.15× to 81%. The 4.15× speedup is obtained for size(HT)=64kB and size(L1D)=128kB (Figure 3.38). For a single instance of B_{Hy}, the speedup is as high as 12.6× (Figure 3.38), in which size(HT)=64kB and size(L1D)=128kB. For up to 3 instances of B_{Hy}, in all 6 plots, the speedup is higher than 3.25×, reducing as the number of B_{Hy} instances increases.

Finally, we perform an experiment to test the speedup due to the HU, for a single benchmark, as the fraction of hash instructions in the benchmark ($\%Hash_{inst}$) varies, as discussed in Section 3.3.8

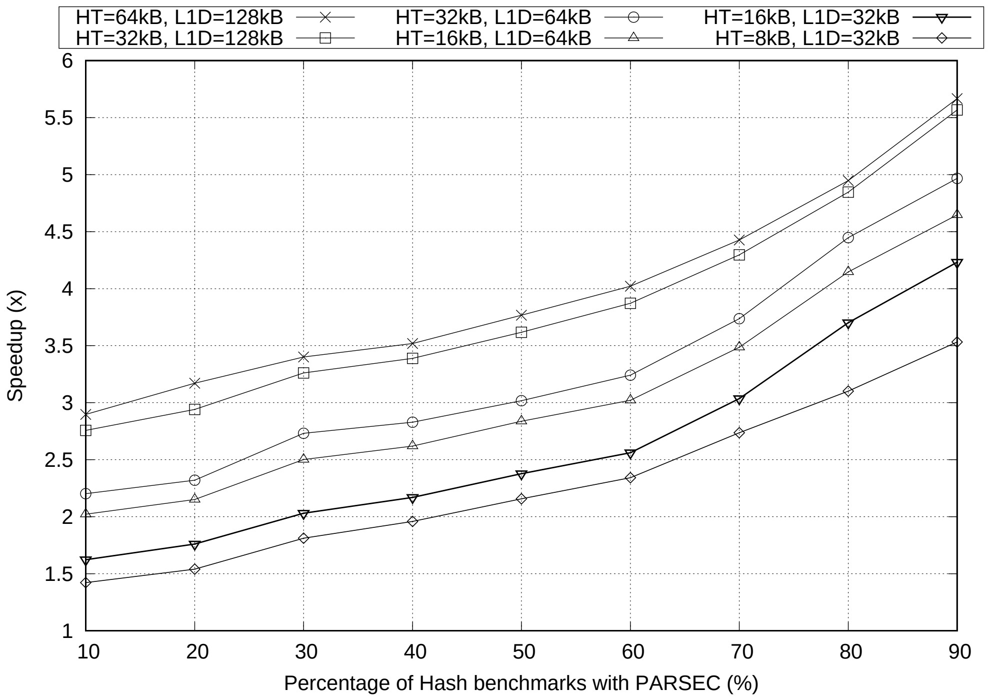


Figure 3.35: (YCSB) Hash and PARSEC Benchmarks in Parallel (B_{Hy} and B_P) – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

and illustrated in Figure 3.6. The benchmark we chose was NBSAT, and the results are shown in Figures 3.40 and 3.41. We note that the speedup of the single NBSAT instance (with injected hash instructions) increases as $\%Hash_{inst}$ increases, in all 6 plots of Figures 3.40 and 3.41. For each plot in Figure 3.40, the corresponding plot of Figure 3.41 exhibits greater speedup, as expected. These plots suggest that for all the HT and L1D sizes studied, including the HU enhances the performance of any application, even if it has a relatively small fraction of hash operations. For a $\%Hash_{inst}$ value of 10%, the speedup ranges between 2.37× and 28%. A speedup value of 2.01× is obtained for size(HT)=64kB and size(L1D)=64kB (Figure 3.41).

In the next chapter, we will discuss our HU at the circuit level for use in modern microprocessors.

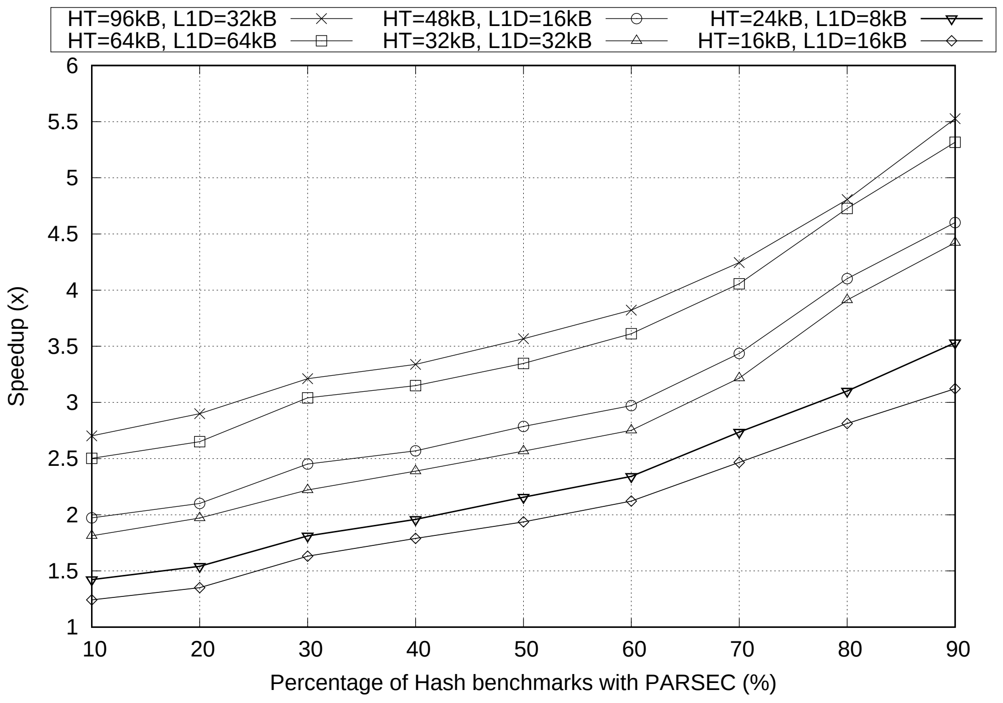


Figure 3.36: (YCSB) Hash and PARSEC Benchmarks in Parallel –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

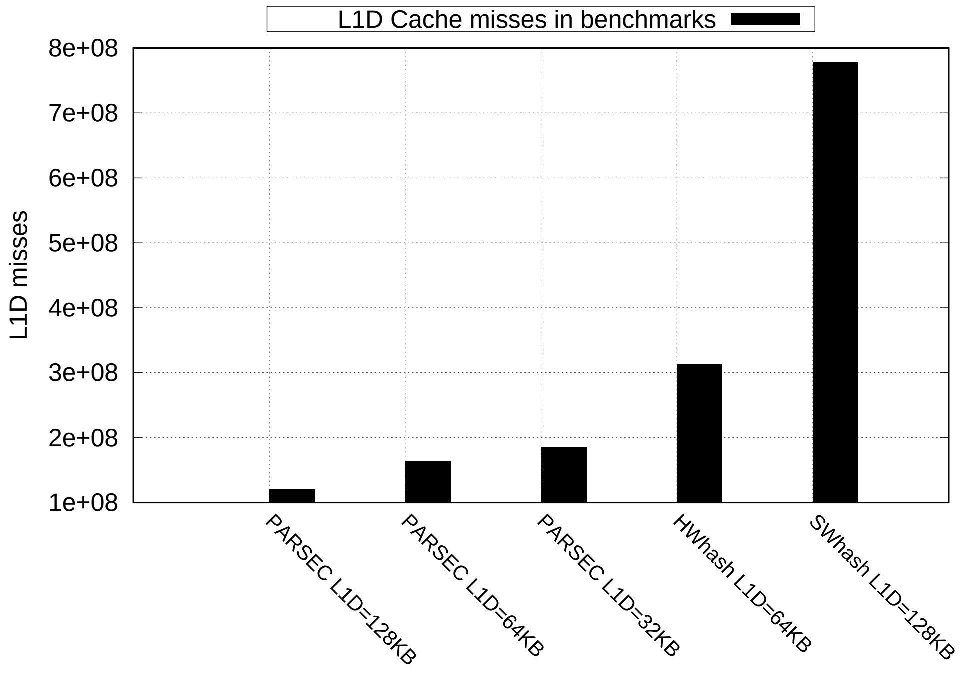


Figure 3.37: (YCSB, HW_{hash}: HT=64kB) First Three Bars: 9 PARSEC Benchmarks with Varying L1D. Fourth Bar: 9 PARSEC + 1 B_{Hy} Running on HW_{hash}. Fifth Bar: 9 PARSEC + 1 B_{Hy} Running on SW_{hash}.

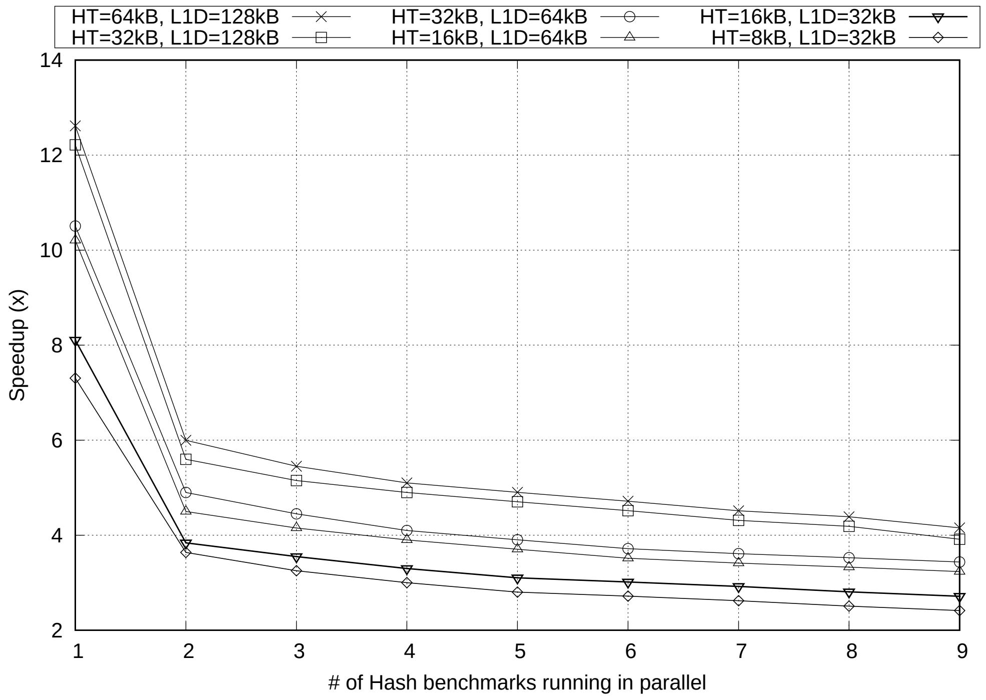


Figure 3.38: (YCSB) Multiple B_{Hy} Instances –
Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

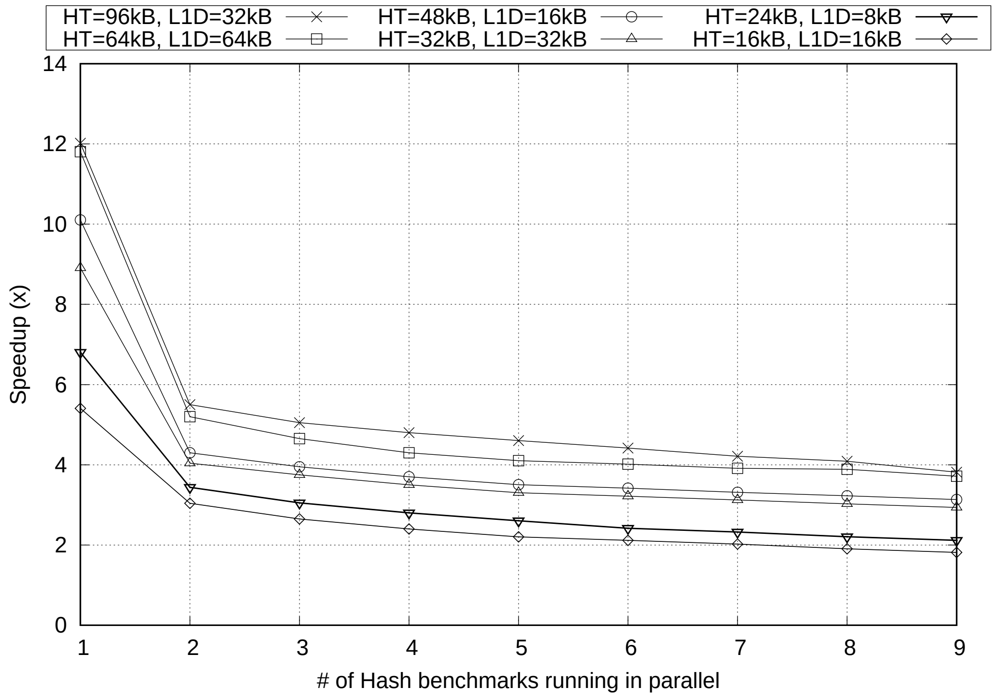


Figure 3.39: (YCSB) Multiple B_{Hy} Instances –
Simulation type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

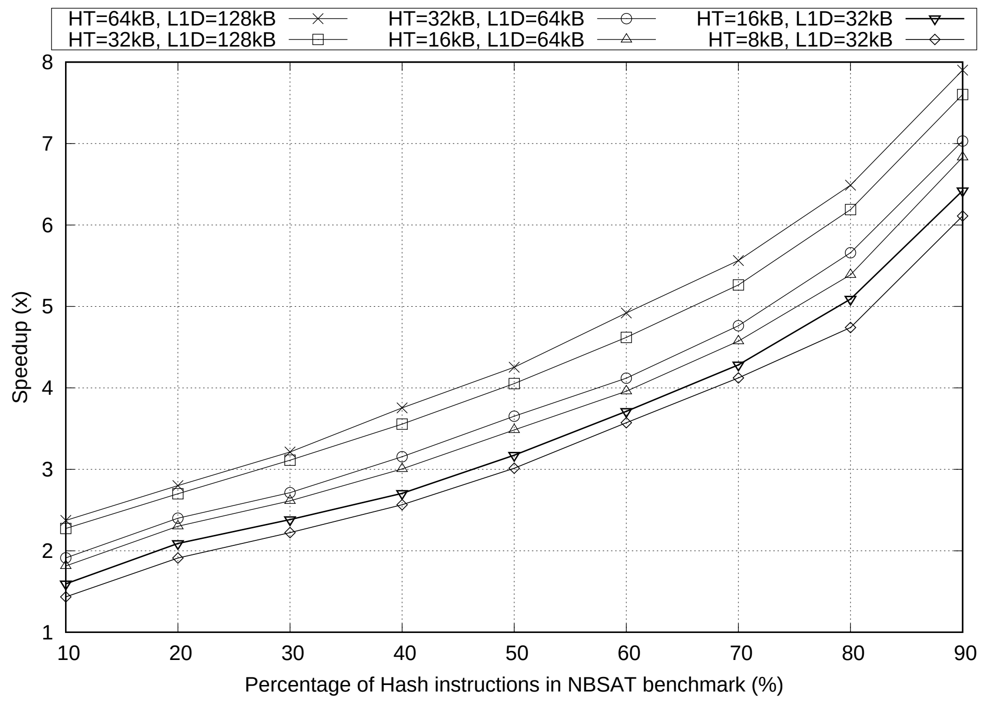


Figure 3.40: (YCSB) NBSAT Benchmark with %Hash$_{inst}$ Varying – Simulation Type G2 (size(L1D) in SW$_{hash}$ = size(L1D) in HW$_{hash}$)

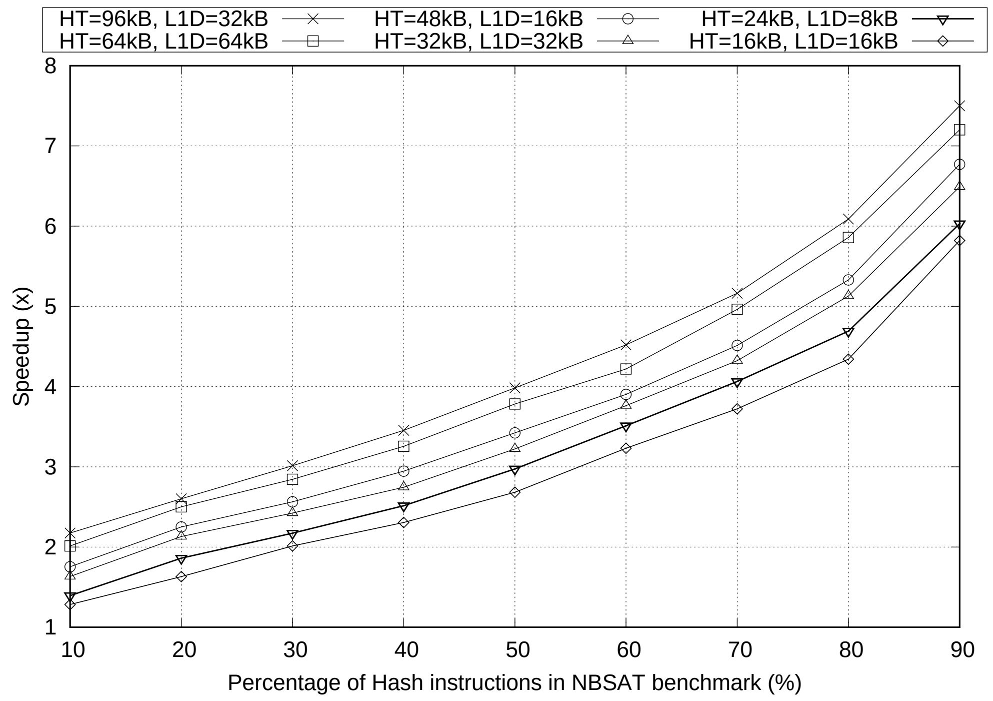


Figure 3.41: (YCSB) NBSAT Benchmark with %Hash$_{inst}$ Varying – Simulation type G1 (size(L1D+HT) in HW$_{hash}$ = size(L1D) in SW$_{hash}$)

3.5 Chapter Summary

In this chapter, we proposed a novel hardware hash unit (HU) design for modern microprocessors. We embedded the HU in the modern microprocessor's execution pipeline. The hash table entries of the HU are stored in a CAM structure. The HU can be shared among multiple applications, and still enable significant speedups. The HU reduces the L1D cache misses for non-hash applications as well. We have implemented (at the micro-architecture level) and verified the operation of the HU. We showed that the HU obtains a speedup of up to 15$\times$ over the software hash implementation with a reduced cache miss rate.

4. CIRCUIT LEVEL DESIGN OF A HARDWARE HASH UNIT FOR USE IN MODERN MICROPROCESSORS [2]

In this chapter, we present our hardware hash unit design at the circuit level. In Section 4.1, we start with a background. Then, we discuss previous work in Section 4.2. In Section 4.3, we present our proposed hardware hash unit at the circuit level design. In Section 4.4, we present and discuss our experimental results. We conclude in Section 4.5.

4.1 Background

In Chapter 3, we proposed a new special function unit (SFU) to speedup hashing operations in modern microprocessors. This SFU was called a hardware hash unit (HU). We conducted our study at the architectural level, with no discussion on the circuit realization aspects of the design. This chapter presents the circuit level details of a HU which is based on the system architecture presented in Chapter 3.

In a software-based hashing implementation, each bin is typically implemented as a linked list, with a search time that is $O(k)$ where k is the number of entries in the bin. In our hardware-based implementation of the HU, we chose a CAM to realize each bin. This provide $O(1)$ search time when the entry is in the CAM.

A content-addressable memory (CAM) [20] is mostly used in fast cache memory applications. The HU design utilize a hash function of class H3 [5] and a CAM-based implementation of the hash table bins. Architectural simulations of this HU were reported in Chapter 3. In this chapter, we implement our ideas of Chapter 3 in a 45nm technology, demonstrating a HU circuit design that obtains an average power reduction of 5.48$\times$ compared to a CAM circuit design, and a clock frequency of up to 1.39 GHz.

The key contributions of this chapter are:

[2] Part of the data (including some figures and tables) reported in this chapter is reprinted with permission from [4] "Circuit Level Design of a Hardware Hash Unit for Use in Modern Microprocessors" by A. Fairouz, M. Abusultan and S. P. Khatri, in Proceedings of the on Great Lakes Symposium on VLSI (GLSVLSI) 2017, pp. 101–106, ACM, May 2017., Copyright 2017 by ACM.

- We implement a hardware hash unit (HU) circuit design for modern microprocessor based on Chapter 3.

- We simulated the HU circuit design in Synopsys VCS and HSPICE [45], using a 45nm PTM [46] and verified the correctness of lookup, insert and delete hash operations. RC parasitic are extracted using Synopsys Raphael [47].

- We demonstrate an average power reduction of 5.48$\times$ using HU over the traditional CAM circuit design. The power reduction arises from the fact that our approach disables all but one bin in any clock cycle.

- Our circuit-level simulations show that the design can operate at a maximum frequency of 1.39 GHz.

The rest of the chapter is organized as follows. In Section 4.2, we discuss related previous work. Section 4.3 presents our approach and design details, while Section 4.4 presents and discusses our experimental results. We summarize the chapter in Section 4.5.

4.2 Previous Work

In this work, we present a hardware-based realization of a hash function *and* a hash table.

In this section, we discuss previous research in which the focus was on implementing hash functions and hash tables. Hardware hash function implementations have been proposed in [23, 24, 25, 26]. These implementations were used to check the equivalence of a pair of large files on different nodes in a network. Networking applications often benefit from the use of hash functions. Such applications are IP addresses hashing schemes [27] and detect and authenticate messages [28]. A hashing technique has been proposed in [48] for high speed networks. In [29], the authors simulated and implemented the SHA-3 on FPGA for cryptographic network applications using a pipeline model to speed up hash operations. An efficient SHA-256 hash function implementation has been proposed in [49]. All these efforts reported work on hardware based hash functions. Other software-based hash functions have been reported as well. Real-time facial identification can benefit from the use of hashing to increase performance [50]. Image compression techniques use hashing in [51] to speed up data compression. Unlike these techniques, our work implements both a hash function as well as a hash table in hardware and can be integrated into a modern microprocessor as an SFU.

There has been some previous work in implementing hash tables as well. In [31], the authors proposed an online hash table implementation on an FPGA using a hash table on external DRAM. A hash join engine using hardware hash table was proposed in [52]. A hashing scheme design has been utilized in [32] for packet processing. The authors implemented the hash table as a set-associative memory module. They simulated their design with various hash function in C++. Furthermore, they proposed a multiple hash functions to reduce collision list in hash table. A key difference of [31, 52, 32] from our approach is that in [31, 52, 32], the hash table bins are not implemented using CAMs, thus sacrificing performance.

In contrast to the previous work, we present a hardware hash function and hash table SFU (with CAM-based bins), for use in modern microprocessors.

4.3 The HU Circuit Design

In Chapter 3, we discussed the hash unit (HU) microarchitecture design. In this section, we discuss our circuit implementation of the HU for use as a new SFU in modern microprocessors. The block diagram of the HU is shown in Figure 4.1. We start with a top-level discussion of the HU circuitry. Next, we discuss the design of the hash function (HF) of class H3 [5] (as shown in Figure 4.2) and the design of the *Bin Selector* circuit. Then, we introduce the control signals unit (CSU) circuit design as shown in Figure 4.3. Finally, we discuss the hash table (HT) circuit design as shown in Figure 4.6.

4.3.1 Hardware Hash Unit (HU)

The hash unit (HU) is an SFU that can be used in modern microprocessors to speedup hash table operations. The HU consists of 3 units: the control signals unit (CSU), the hash function (HF) and the *Bin Selector*, and the hash table (HT) as shown in Figure 4.1. This figure shows the

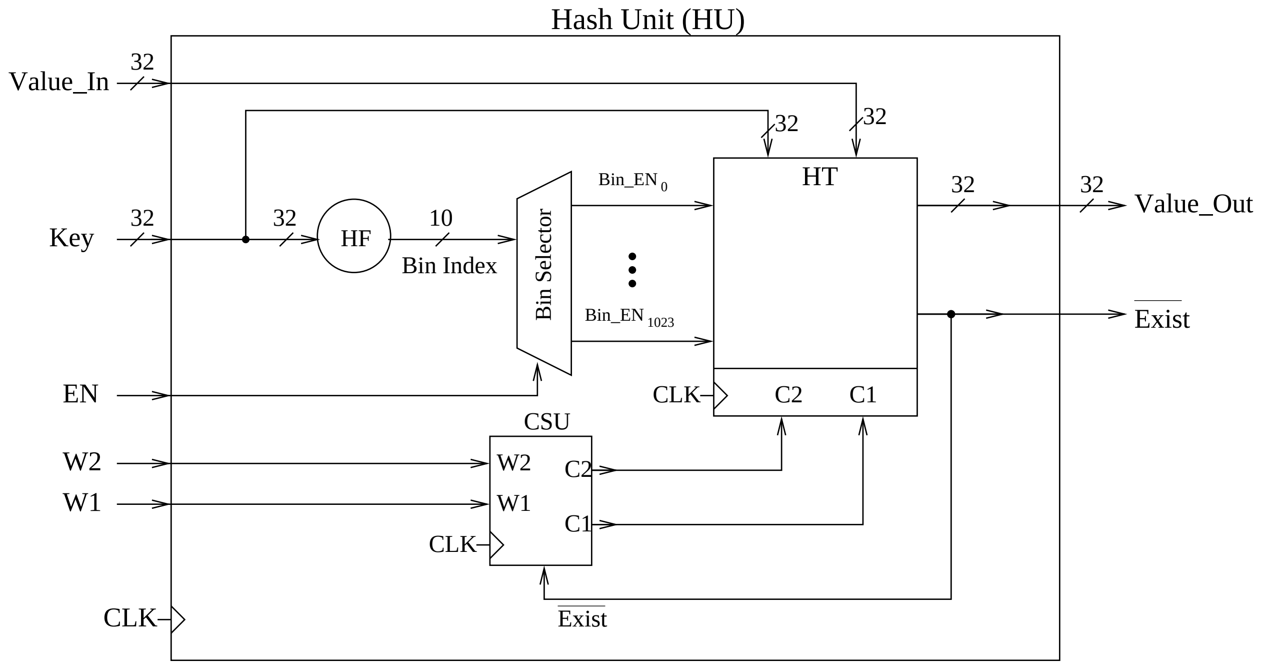


Figure 4.1: Hardware Hash Unit (HU) Block Diagram (Reprinted [4])

flow of hash operation. It takes a (*key*, *value*) pair as an input and return a result based on the operation requested. There are 3 major operations, which are encoded by the W2 and W1 inputs:

- Lookup: it takes a *key* as an input and searches for it in the HT. If the *key* found, it returns the *value* associated with the *key* in the HT, and drives $\overline{Exist}$ low. If the key is not found, it drives $\overline{Exist}$ high.
- Insert: it takes a (*key*, *value*) pair as an input. It first performs a lookup operation in the HT. If there is a match in the HT, it aborts the insert. Otherwise, it inserts the new (*key*, *value*) pair in the HT.
- Delete: it takes a *key* as the input. First, it executes a lookup operation in the HT. If the *key* is found, it deletes this entry. This is accomplished by invalidating the entry and writing a '0' to the valid bit of the entry.

As shown in Figure 4.1, the inputs and outputs of the HU are:

- Value_In: it is the *value* associated with a (*key*, *value*) pair that is input to the HT.
- Key: it is the *key* in a (*key*, *value*) pair that is input to the HT.
- EN: it is the enable signal that enables the HU.
- CLK: main clock signal.
- W2 & W1: op-code signals which encode the micro-instruction to the HT.

The outputs of the HU are the *Value_Out* and $\overline{Exist}$. The *Value_Out* signal is the output of a lookup operation. The $\overline{Exist}$ signal indicates the outcome of a lookup operation.

4.3.2 Hash Function (HF) and Bin Selector

The hash function (HF) takes a *key* as an input, and generates a *Bin Index*, as shown in Figure 4.1. The *Bin Index* is fed to a *Bin Selector* to generate several *Bin_EN*$_i$ signals. The *Bin Selector* is a decoder, that decodes a *Bin Index* to generate one of n *Bin_EN*$_i$ signals ($n = 1024$ in Figure 4.1, where n is the number of bins in the HT). The *Bin_EN*$_i$ signals are one-hot. If the *Bin Index* value (in decimal) is *k*, then the *Bin_EN*$_k$ signal is high, and all other *Bin_EN*$_j$ signals are low. The HF is of class H3 [5]. It performs hash operations based on AND and XOR logic as shown in Figure 4.2. The AND and XOR operations result in a fast hash operation. Hashing is performed using a Q matrix which is stored in latches. This Q matrix is a random number of class H3 [5]. The Q matrix values are stored during the HU initialization. The HF can be changed for different applications, by changing the Q matrix contents.

4.3.3 Control Signals Unit (CSU)

The control signals unit (CSU) generates the C1 and C2 signals to perform the appropriate sequence of atomic operations in the HT, as shown in Figure 4.1. The CSU takes 4 input signals CLK, W1, W2, and $\overline{Exist}$ and generates C1 and C2, the output control signals. The operations of the HU (encoded by (W2,W1)) require a sequence of atomic operations (encoded by (C2,C1)) to be done. For example, the insert operation (W2,W1 = 11) requires a lookup atomic operation

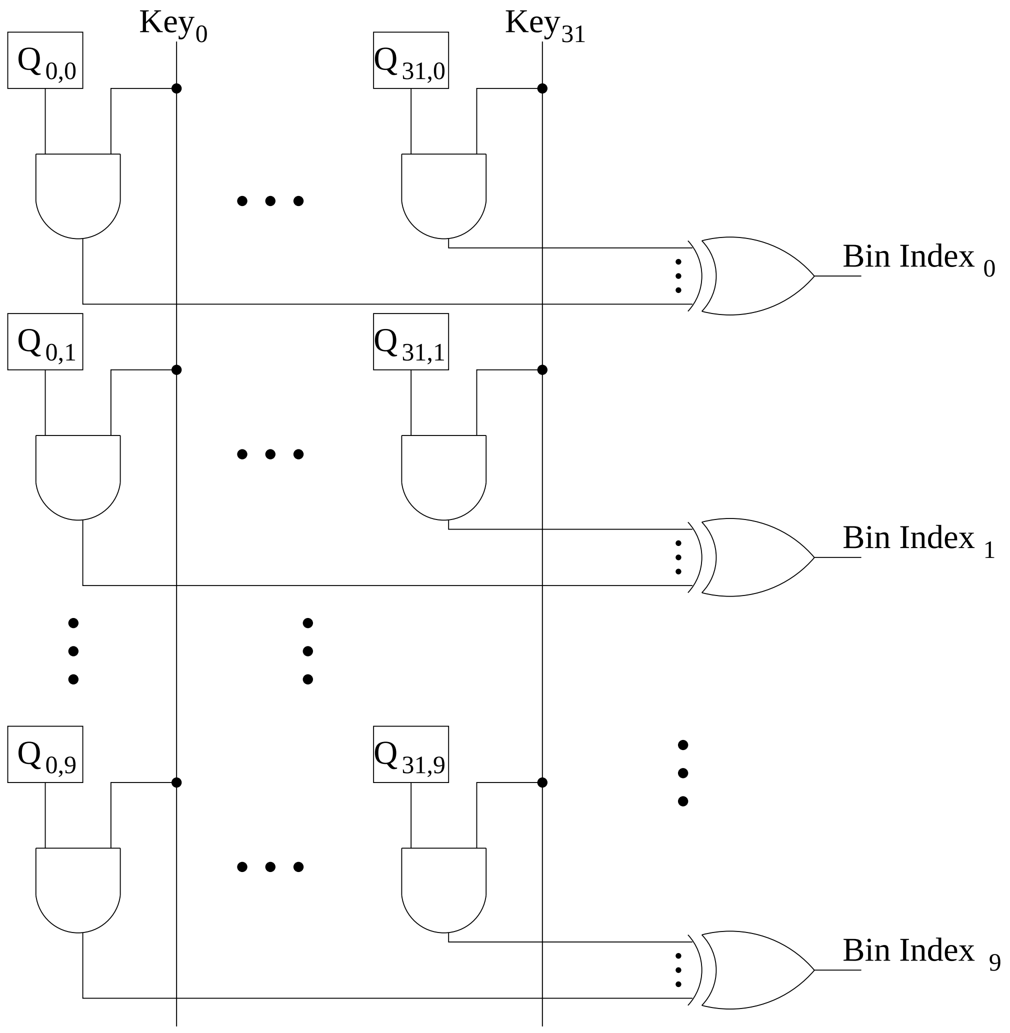


Figure 4.2: Hardware Hash Function (HF) of Class H3 [5] (Reprinted [4])

(C2,C1 = 01) to be performed, and then possibly an insert atomic operation (C2,C1 = 11). The CSU encodes the W1, W2 and $\overline{Exist}$ signals to perform the sequence of atomic operations on the HT (indicated by the CSU outputs (C2,C1)). Table 4.1 illustrates the mapping between the operations and the values of (W2,W1) and the (C1,C2) signals.

Note that for every operation (i.e. every op-code on (W2,W1)), the HU has to perform a lookup operation, except for the 00 op-code (no-op operation). As a result, the delete and insert operations can take two cycles in the worst case. In the first cycle, the CPU drives (W2,W1) to 11 (insert) or 10 (delete). In the second cycle, (W2,W1) are driven to 00 in both cases. For example, when the

Operation	W2	W1	$\overline{Exist}$	C2	C1
Insert	1	1	1	1	1
Delete	1	0	0	1	0
Lookup	0	1	x	0	1
no-op	0	0	x	0	0

Table 4.1: CSU op-codes for the HT operations

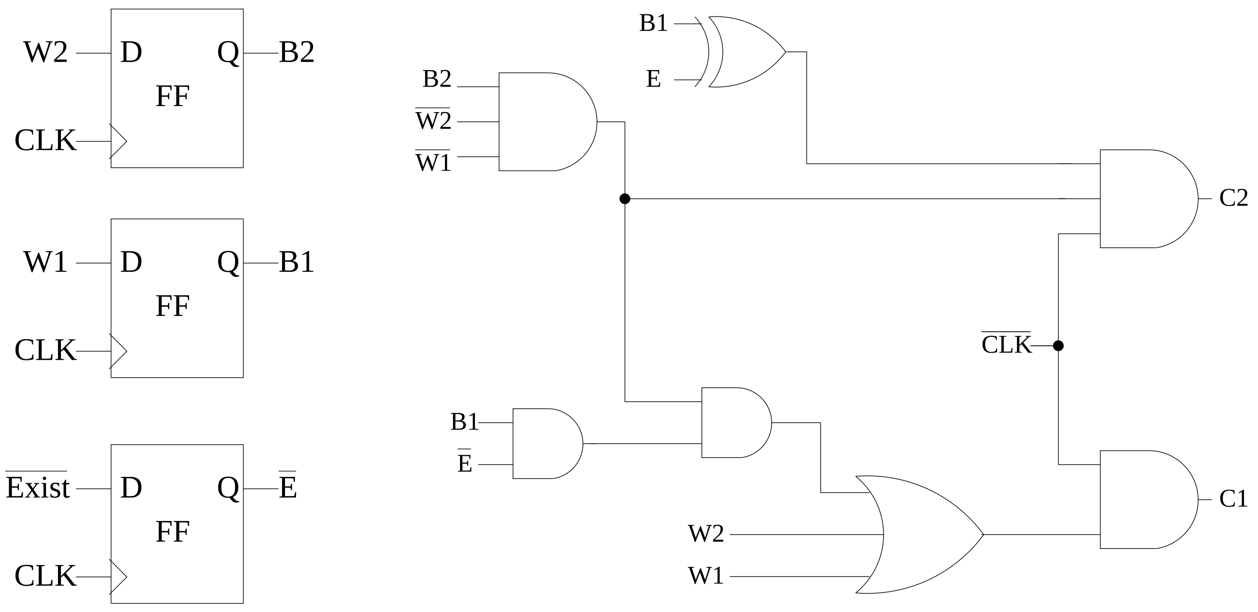


Figure 4.3: Control Signals Unit (CSU) (Reprinted [4])

insert operation (W2,W1=11) is issued, a lookup atomic operation is performed in the first cycle by driving (C2,C1 = 01). If $\overline{Exist} = 1$ results, an insert atomic operation is performed in the next cycle by driving (C2,C1 = 11). If $\overline{Exist} = 0$, then (C2,C1 = 00) in the second cycle. Note that, in the second cycle, (W2,W1) are driven to 00. Similarly for the delete operation, the CPU will first issue a (W2,W1 = 10) op-code in the first cycle. Then, it will issue a (W2,W1 = 00) op-code in the second cycle. In the first cycle, the CSU will drive (C2,C1 = 01) to perform the lookup atomic operation. In the second cycle, the CSU generates the C2 and C1 signals based on the value of $\overline{Exist}$ signal. If $\overline{Exist} = 1$, then the CSU generates 00 for (C2,C1) and performs a no-op. Otherwise, the CSU drives 10 for (C2,C1) to delete the (*key*, *value*) pair in the HT. The CSU implements the state machine which produces the values of (C2,C1) based on the values of (W2,W1) and $\overline{Exist}$. This state machine is shown in Figure 4.3.

4.3.4 Hash Table (HT)

We design each bin of our Hash Table (HT) using a basic 9-T NOR-type CAM cell [20] (Figure 4.4) to store a *key*, and a conventional 6-T SRAM cell [53] (Figure 4.5) to store a *value*. Each HT bin is implemented as a CAM in order to perform fast hash operations.

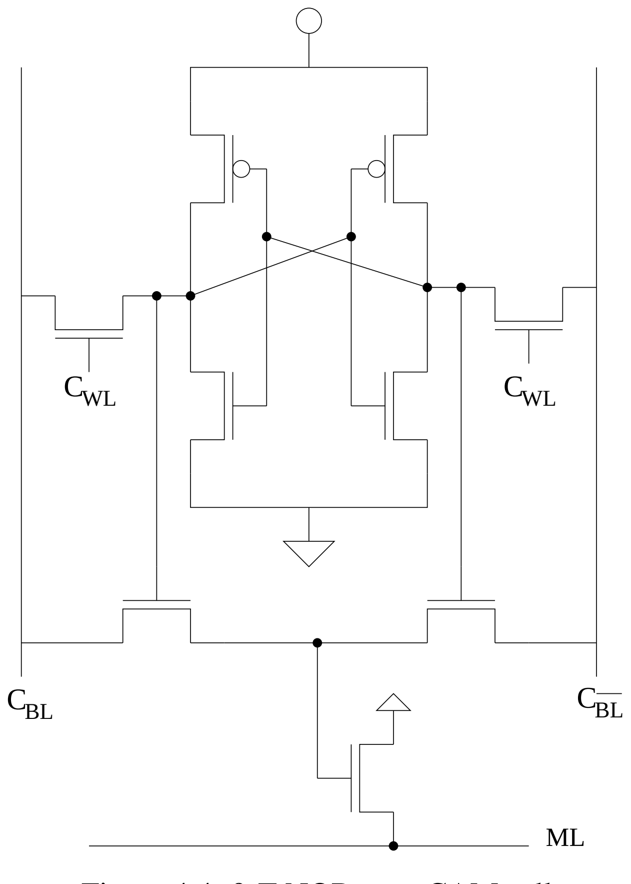


Figure 4.4: 9-T NOR-type CAM cell

The HT has 6 inputs and 2 outputs as shown in Figure 4.1. The inputs of the HT are: *key*, *Value_In*, *Bin_EN$_i$*, CLK, C2, and C1. The *key* and *Value_In* inputs are the (*key*, *value*) pair. The

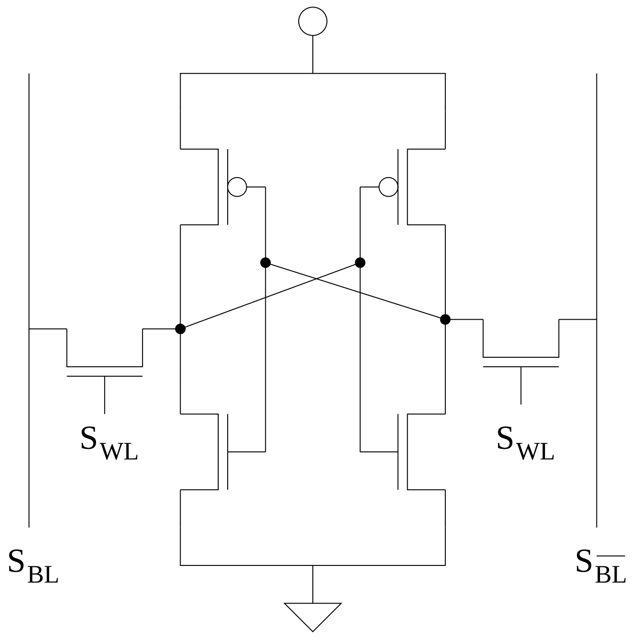


Figure 4.5: 6-T SRAM cell

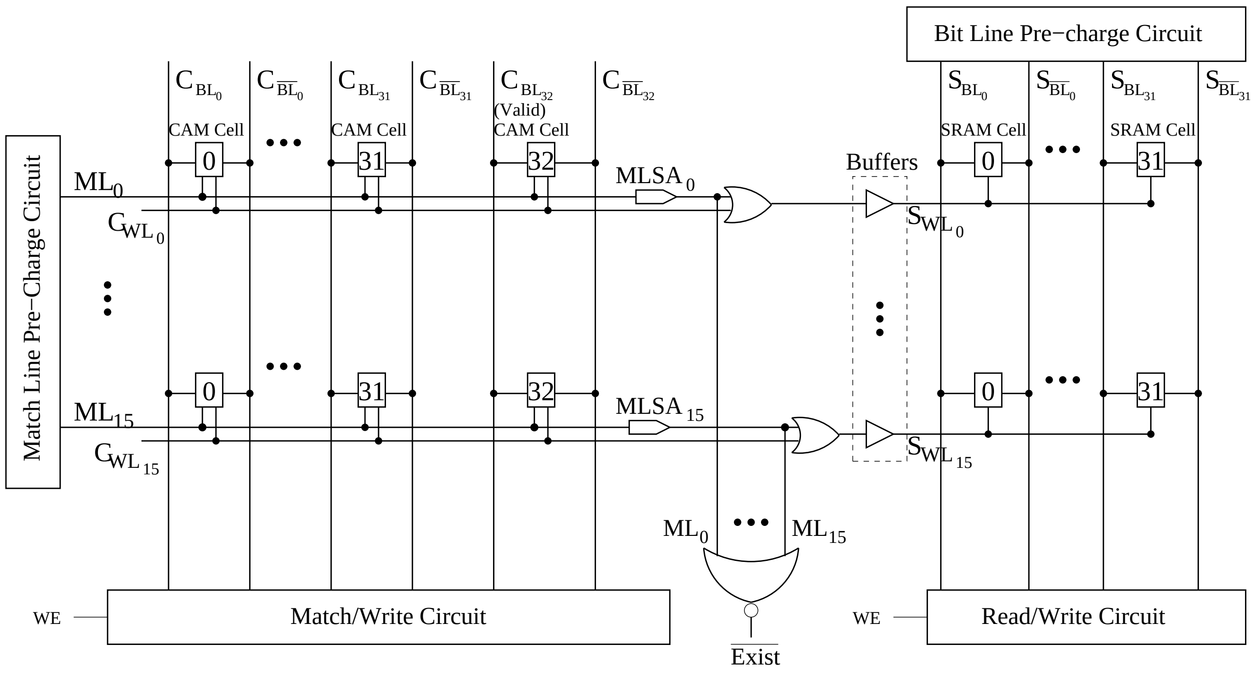


Figure 4.6: Hardware Hash Table (HT) for a Single Bin [3] (Reprinted [4])

HT stores an extra *valid* bit along with the *key*, to indicate if the corresponding (*key*, *value*) pair is valid. This bit is written to *zero* during a delete operation. During an insert operation, it is driven to a logic 1. During a lookup, this bit must be a 1, otherwise the lookup fails. The *Bin_EN$_i$* signals are the outputs of *Bin Selector* which only enables a single bin of the HT in any clock cycle. The C2 and C1 inputs are the outputs of the CSU, which indicate the atomic operation to be performed (lookup, delete or insert) in the HT. As shown in Figure 4.6, there are two major blocks of memory in a HT bin. The first block is 33 CAM cells wide (32 bit of *key* and one *valid* bit) and the second is 32 SRAM cells wide (32 bit of *value*). The number of rows m represents the number of entries in a HT bin ($m = 16$ in Figure 4.6). There is a match line pre-charge circuit to pre-charge ML_0, ..., ML_m. Also, there is a bit line pre-charge circuit to pre-charge the SRAM bit lines (S_{BL_i}, $\mathrm{S}_{\overline{BL_i}}$). There is a Match/Write circuit for CAM (since the polarity of C_{BL_i} and $\mathrm{C}_{\overline{BL_i}}$ are reversed during a write to the CAM as opposed to a lookup) and a Read/Write circuit for SRAM cells (since the S_{BL_i} and $\mathrm{S}_{\overline{BL_i}}$ lines are pre-charged during lookup and driven during a write to the SRAM). The Match/Write and Read/Write circuits are enabled by a write enable signal (WE). The WE signal is set to 1 for the delete or insert atomic operations (i.e. when (C2,C1) = 11 or 10). The *key* input is driven on the C_{BL_i} lines using the Match/Write circuit in all HT atomic operations. In the Read/Write circuit, the *Value_In* input is driven on the S_{BL_i} lines in the insert atomic operation, while the S_{BL_i} lines are driven on the *Value_Out* output in the lookup atomic operation.

During a lookup operation, all C_{WL_i} lines are driven to a *zero* value. All ML lines are pre-charged, and at most one (say ML_i) stays *high* (indicating a match). This ML_i is amplified by the sense amplifier MLSA_i, and results in the $\overline{Exist}$ line being driven to a *zero* value. If no ML_i stays *high* (a mis-match condition), then $\overline{Exist}$ ends up being driven to a 1. When ML_i is *high*, then S_{WL_i} is also driven *high*, and the *value* is read out from the SRAM.

S_{WL_i} is also set *high* if C_{WL_i} is driven *high*, in a delete or insert operation (which requires the SRAM to be updated). We discuss the delete operation next.

For a delete operation, we first perform a lookup atomic operation ((C2,C1) = 01). The ML_i which is found to match during the lookup is latched in the Q_i signal of the i^{th} D-latch in Figure 4.7.

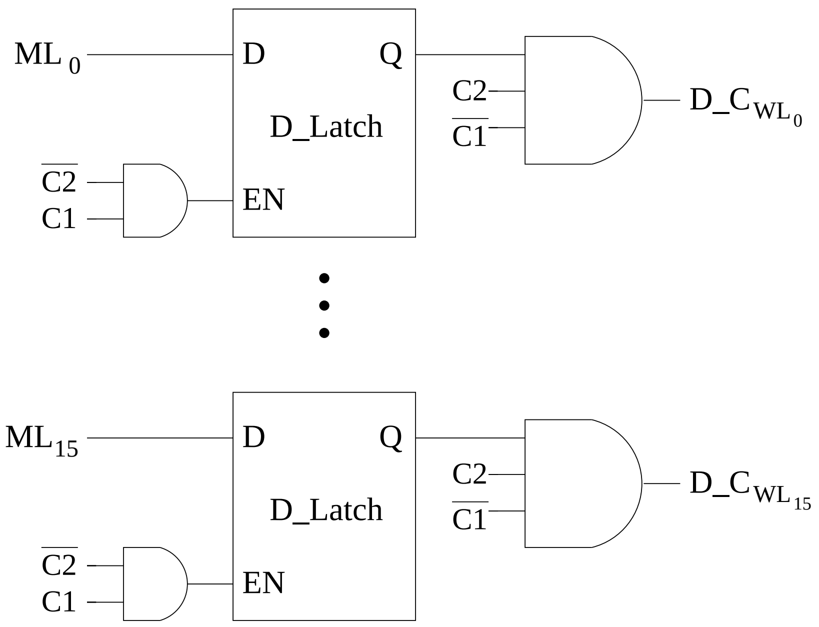


Figure 4.7: Determining WL for Deletion (Reprinted [4])

At most one Q_i signal is high after the lookup operation. During the delete atomic operation ((C2,C1) = 10), the corresponding D_C$_{WL_i}$ is driven *high*, which drives C$_{WL_i}$ *high* (see Figure 4.9). Hence the deletion is performed on the ith row of the CAM, as required.

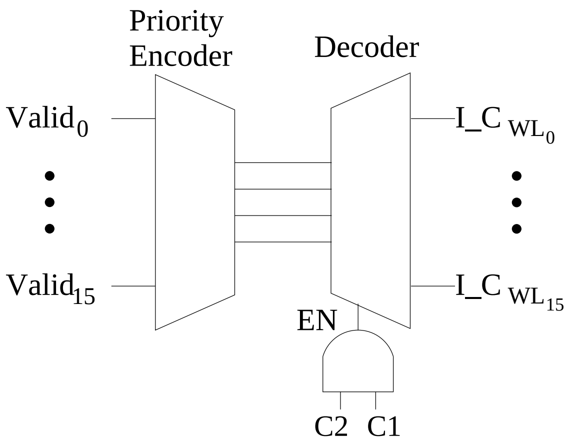


Figure 4.8: Determining WL for Insertion (Reprinted [4])

For an insert operation, we use a priority encoder to find the index of the first row which is not

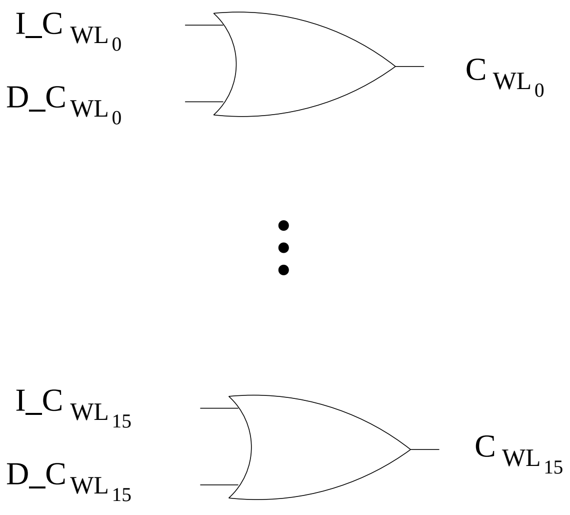


Figure 4.9: Deriving C$_{WL_i}$ (Reprinted [4])

valid. Such a row can be written into since it is invalid. This is shown in Figure 4.8. The valid bits are stored in latches as well as in the CAM cells. During the insert operation (C2,C1 = 11), we decode the index of the first invalid entry (Figure 4.8) and drive a single I_C$_{WL_i}$ line *high*. This causes C$_{WL_i}$ to be driven *high* (Figure 4.9), resulting in an insertion in the first invalid entry as desired.

In Section 4.4, we will present our experimental results of the HU, at the circuit level.

4.4 Experimental Results

In this section, we present the simulation results of the HU at the circuit level. First, we present the simulation environment used to implement the proposed design of the hash unit. Then, we discuss the procedure we used to determine that the design operates correctly. Finally, we present the circuit-level simulation results along with a discussion.

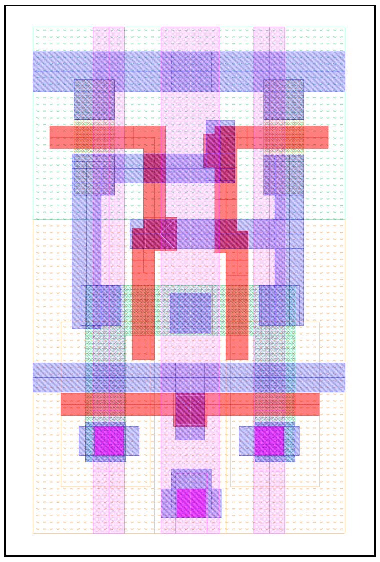
4.4.1 Simulation Environment

Since our proposed hardware-based HU engine is based on the use of a CAM per bin of the HT, we compare our design with a traditional CAM of the same total size as the HU. One can conceive of a design in which the CAM of a microprocessor can be dynamically split between the cache and the HU. In such a scenario, the power and area comparison between the HU and the traditional CAM is an important figure of merit to consider.

We compare two design implementations: the HU and a traditional CAM. In the traditional CAM, the whole memory is enabled, while the HT in the HU design enables a single bin. We simulated both designs using Synopsys HSPICE [45] and 45nm PTM [46] high-performance process model card. We used custom Perl [54] scripts to generate both designs. We synthesized the HF, the *Bin Selector*, and the CSU from Verilog and simulated them in HSPICE while the HT and traditional CAM were custom designed, and simulated in HSPICE.

The ratio of the width of the pull-down NMOS device in the SRAM (and CAM) cell to the width of the NMOS access transistor of the same cell was chosen to be 1.25 [55]. Also, we choose minimum size for the pull-up transistors in the SRAM and CAM cells [55] in order to obtain minimum size, read stability and writeability.

We also generated the layout of the CAM and SRAM cells as shown in Figure 4.10(a and b). These layouts are based on the 45nm design rules [6] and are generated using the Cadence Virtuoso [56] layout tool. We then used Synopsys Raphael [47] to extract the parasitic capacitance between wires (such as bit lines, world lines, and match lines) in the SRAM and CAM cells. We used the parasitic resistance and capacitance results in our HSPICE simulation for both the HT


(a) SRAM cell layout

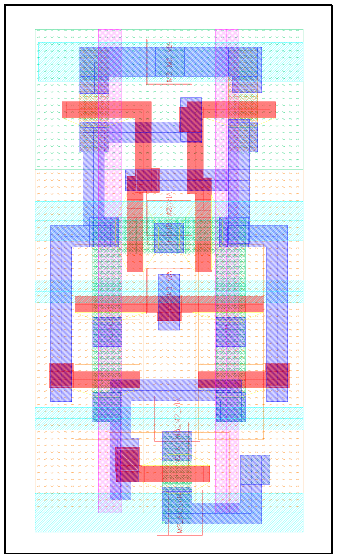

(b) CAM cell layout

Figure 4.10: SRAM and CAM cell layouts in 45nm design rules [6] (Reprinted [4])

and traditional CAM. We sized the memory (CAM and SRAM) drivers and buffers based on the number of entries in the HT bin as well as wire parasitics. In our design, we pre-charge the bit-lines of the SRAM cells and the match lines of the CAM cells to VDD.

4.4.2 Design Verification

We designed the HF, the *Bin Selector*, and the CSU in structural Verilog and then generated a HSPICE netlist using Synopsys V2S [57]. Then, we simulated and verified the logical correctness of these units using Synopsys VCS [58].

After that, we simulated the HF, *Bin Selector*, and CSU in HSPICE to verify their correctness, and determine their delay, power and area requirements.

We simulated the HT and the traditional CAM circuits with custom scripts in HSPICE to verify their correct functionality. These scripts performed lookup, delete and insert operations.

Finally, we integrated all the blocks of the HU design (HF, *Bin Selector*, CSU and HT) and verified the correctness of all operations (lookup, insert and delete) using HSPICE. The same HSPICE level verification was performed for the traditional CAM as well.

4.4.3 Results and Analysis

We simulated both the HU and the traditional CAM designs in HSPICE [45] to measure their delays, dynamic power and static power.

Entries/Bin	Operation	Pre-chg Delay (ps)		Evaluate Delay (ps)		Bin Dynamic Power (mW)
		CAM	SRAM	CAM	SRAM	
4	Insert	206.15	249.24	217.17	268.21	10.46
	Lookup			122.74	135.01	9.32
	Delete			202.73	NA	9.37
8	Insert	245.12	273.42	258.48	294.11	20.67
	Lookup			122.66	134.93	18.51
	Delete			245.90	NA	19.01
12	Insert	261.29	289.13	276.32	310.13	31.21
	Lookup			122.47	134.71	27.84
	Delete			262.81	NA	28.42
16	Insert	265.13	291.41	302.34	331.41	40.74
	Lookup			122.37	134.84	37.40
	Delete			289.27	NA	37.53

Table 4.2: Delay and Dynamic Power analysis of Insert, Delete, and Lookup operations in HT (Modified [4])

Table 4.2 reports the results as the number of entries per bin are varied (Column 1). Results are shown for each operation (lookup, insert, and delete). The dynamic power increases proportionally as the number of entries per bin increases. The evaluation delay of SRAM cells in the delete operation (Column 6) is not applicable, since the delete operation writes a logic '0' to the *valid* bit of the corresponding CAM entry *only* (if the lookup atomic operation is successful). The SRAM entry is not written to in this case.

We note that the insert operation has the highest dynamic power consumption and the longest evaluation delay, compared to the lookup and delete operations, and as such, determines the worst case delay and power of the HU. For 16 entries/bin, the clock period of an insert operation (after adding a 15% guard-band due to PVT variations) is slightly lower than 720 ps. Hence the HU can be operated at a maximum frequency of about 1.39 GHz.

HT (kB)	Bins	Entries	Area μ^2			HU Operating Speed (GHz)
			HU	Tr.CAM	HU/Tr.CAM	
8	128	16	120223.11	120117.25	1.0009×	1.39
16	256	16	240347.12	240234.50	1.0005×	1.39
24	512	12	360472.84	360351.74	1.0003×	1.45
32	512	16	480590.09	480468.99	1.0003×	1.39
48	1024	12	720830.96	720703.49	1.0002×	1.45
64	1024	16	961065.45	960937.98	1.0001×	1.39
96	2048	12	1441545.64	1441406.98	1.0001×	1.45
Avg.					1.00033×	

Table 4.3: Area and Delay analysis of the HU for different configuration of the HT (Modified [4])

In Table 4.3, we report results for different HT sizes that were used in the architectural simulations of Chapter 3. The different HUs utilize 12 or 16 entries per bin, with the number of bins varying from 128 to 2048.

We determine our worst case power and delay numbers assuming that one bin performs an insert operation (which was shown in Table 4.2 to be the slowest and most power hungry operation) and all other bins are not enabled, and consume static power.

From Table 4.3, we note that the area (Column 4) of the different HU designs is roughly proportional to the total HT size (Column 1). Also, the ratio of the HU area to the area of the traditional CAM (Column 6) is almost unity, indicating that the total overhead of the HF, *Bin Selector* and CSU blocks is very small compared to the HT area. In other words, the size of the HU (on average) is only 0.033% larger than that of the traditional CAM.

Column 7 of Table 4.3 shows the worst case operating speed of the HU (with a 15% guard-band for PVT variations included). Note that due to the fact that we size the drivers and buffers based on the HT size, these numbers are relatively constant.

HT (kB)	Bins	Entries	Total Power (mW)			Dyn Power (mW)	
			HU	Tr.CAM	Tr.CAM/HU	HF+BS	CSU
8	128	16	82	389	4.74×	0.2578	0.0242
16	256	16	136	750	5.51×	0.2963	
24	512	12	224	1141	5.09×	0.3329	
32	512	16	246	1477	6.00×	0.3329	
48	1024	12	426	2261	5.31×	0.3699	
64	1024	16	464	2921	6.30×	0.3699	
96	2048	12	831	4501	5.42×	0.4143	
Avg.					5.48×		

Table 4.4: Power analysis of the HU for different configuration of the HT (Modified [4])

In Table 4.4, we note that on average, the power consumption of the HU (Column 8) is about 5.48× lower than the traditional CAM (Column 9). This is because in the HU design, exactly one bin consumes active power, and the remaining bins are static. In the traditional CAM, however, the entire CAM consumes dynamic power.

Columns 11 and 12 report the power consumption of the HF and *Bin Selector* units (Column 11) and the CSU unit (Column 12). We note that these power numbers are significantly smaller than the corresponding HT power numbers, as expected.

4.5 Chapter Summary

In this chapter, we proposed the circuit design for a hardware hash unit (HU) for use as a new SFU in modern microprocessors. Each bin of the HU was implemented as a CAM. We verified the correctness of all hash operations at the logic as well as the circuit level. We demonstrated an average power improvement of 5.48$\times$ for our HU design compared to a traditional CAM design. We also quantify the area, delay, and power for different HT sizes. We showed that the HU can be operated at 1.39 GHz after guard-banding for PVT variations.

5. AN FPGA-BASED COPROCESSOR FOR VIRUS CHECKING APPLICATIONS [3]

In this chapter, we present our FPGA-based coprocessor for virus checking applications. We start with a background in Section 5.1. In Section 5.2, we discuss the previous work. In Section 5.3, we present our proposed approach. Then, we present our experimental results in Section 5.4. Finally, we conclude in Section 5.5.

5.1 Background

In order to speedup hashing intensive applications, we proposed a hardware hash unit (HU) at the michroarchitecture level in Chapter 3. Furthermore, a custom VLSI circuit level realization (using a 45nm fabrication process) of the HU was presented in Chapter 4. The HU in Chapter 3 and 4 showed an impressive speedup for algorithms that utilize hashing operations. This chapter focuses, instead, on an FPGA realization of a HU, acting as a coprocessor to a computer system, and communicating over a PCIe interface. In situations where it is not cost-effective to build a HU as an SFU on the microprocessor die as in Chapter 3 and 4, one can build the HU as a coprocessor to the CPU. This is what we explore in the current chapter.

FPGA-based coprocessors are a commonly deployed re-configurable hardware platform today. Many datacenters use FPGA accelerators to speedup cloud applications like search [59] or machine learning [60, 61]. In this chapter, we validate the conjecture that an FPGA-based HU would increase the performance of hashing-intensive applications for a computing platform.

Computer system security has been a concern for decades in the computing community. To check for system integrity, many virus checking applications are utilized, for different operating systems. Virus checking applications operate by computing the hash signatures (usually MD5 hash based) for all files in the computer system. These signatures are then tested against a database of signatures of all known viruses. Virus checking applications require heavy computations, thereby

[3] Part of the data (including some figures and tables) reported in this chapter is reprinted with permission from [7] "An FPGA-based Coprocessor for Hash Unit Acceleration" by A. Fairouz and S. P. Khatri, in 2017 IEEE International Conference on Computer Design (ICCD), pp. 301– 304, Nov 2017., Copyright 2017 by IEEE.

reducing the performance of the computer system. We expect that implementing an FPGA-based HU will increase the performance of such applications.

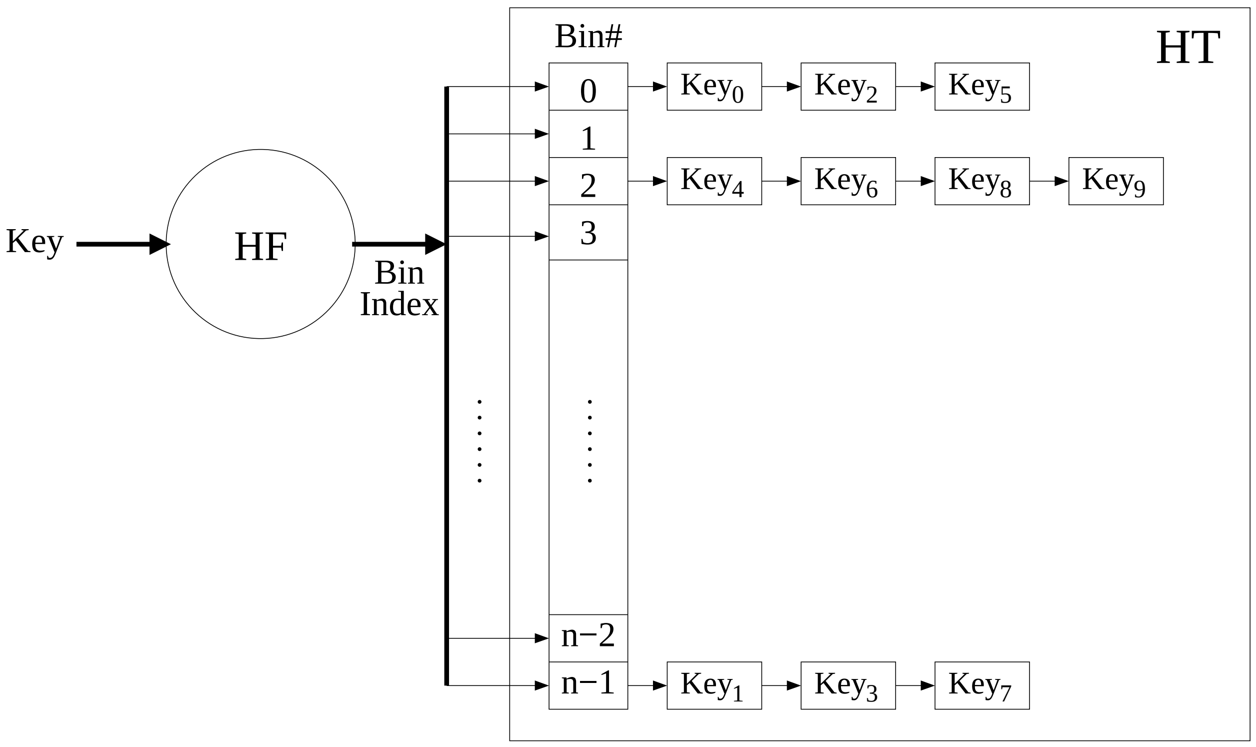


Figure 5.1: Software-based hash table implementation (Reprinted [7])

In this chapter, we implement the idea of the HU architectural model proposed in Chapter 3 in an FPGA-based HU design. We utilize a hash function of class H3 [5] and a hash table bins of CAM-based [20] implementation. Our FPGA-based HU design obtains a speedup of up to 5.37× for a hash-based virus checking application.

The key contributions of this chapter are:

- We implement an FPGA-based hardware hash unit (HU) design for virus checking application. The HU supports membership checks, using a hardware-based hash function and hash table.

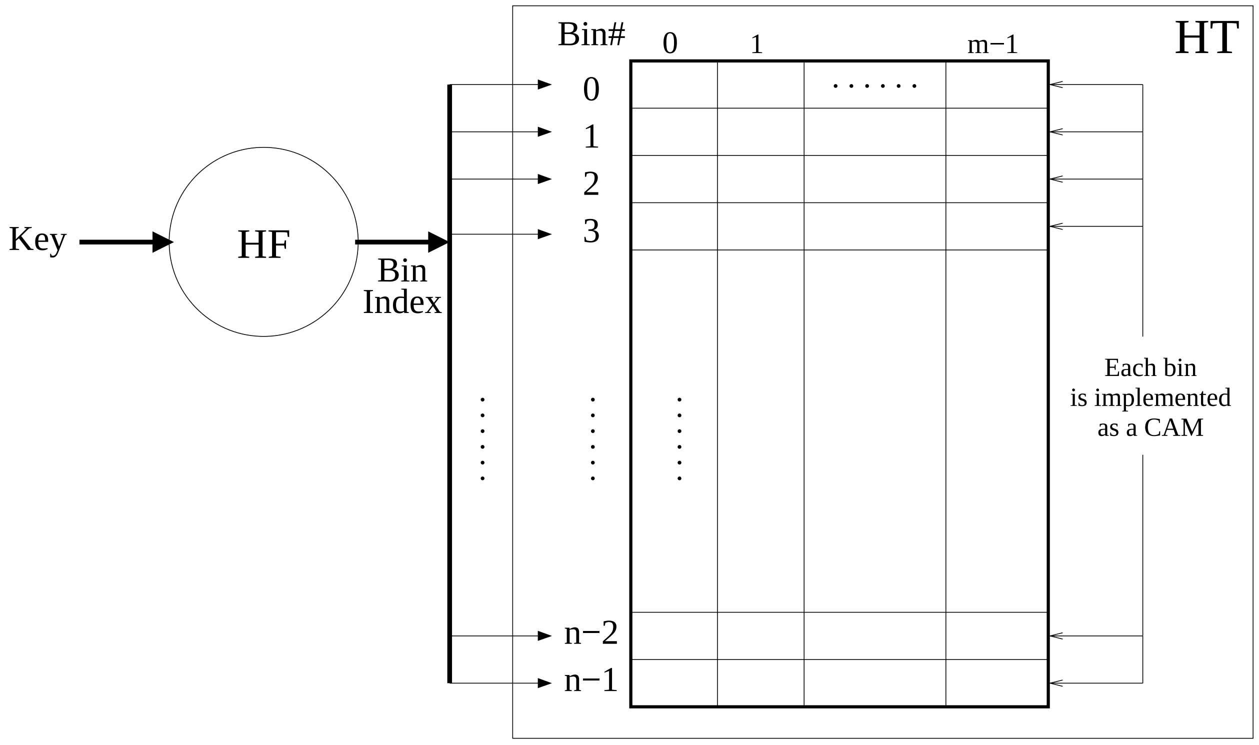


Figure 5.2: Hardware-based FPGA hash table implementation (Reprinted [7])

- Our FPGA-based HU implementation is flexible and can be used for other applications as well. We use the virus checking application as a candidate.
- Our hash table bins are implemented as CAMs on the FPGA, for $O(1)$ lookup. If the entries of a bin do not fit in the CAM, they spill over into DRAM, which stores the complete hash table contents.
- We use Xilinx ISE Design Suite [62] for our FPGA design flow, and verified the correctness of lookup, insert and delete hash operations.
- We observe a speedup of up to $5.37\times$ while varying the size of the hardware hash table, the number of lookups per burst, and the database (DB) size. We obtain our HT database from VirusShare [63].
- We quantify the scaling of the speedup of the HU when we vary the design parameters (DB size, hash table size, and number of lookups per burst).

- Our FPGA-based HU implementation supports streaming hash operations.

The rest of the chapter is organized as follows. In Section 5.2, we discuss related previous work. Section 5.3 describes our approach, and Section 5.4 presents experimental results. We summarize this chapter in Section 5.5.

5.2 Previous Work

In this chapter, we present an FPGA-based implementation of a hash function and a hash table. This FPGA-based implementation focus on streaming data applications, such as virus checking applications.

Virus checking applications has been an important part in computer systems for years. There have been many efforts to improve the performance of such applications. In [64], the authors propose hashing smaller amounts of data to reduce the runtime for virus checking. An MD5 checksum lookup scheme has been proposed by [65], to increase the virus checking performance. An automatic virus checking model has been proposed by [66] to generate virus signatures.

Unlike previous techniques, we propose a hardware-based hash unit for such applications. The hash unit consists of a hash function *and* a hash table. Some prior approaches have been described in Chapter 3. Unlike previous implementations, we implement both a hash function and a hash table in an FPGA-based coprocessor. Our FPGA-based HU is based on the architectural model proposed in Chapter 3 and the circuit level design presented in Chapter 4, while being realized in a Linux-based operating system as an FPGA-based HU coprocessor.

5.3 CPU-FPGA Hash Unit

In this section, we start with a top-level discussion of the HU using an FPGA as a coprocessor. Next, we discuss the Hash Function (HF) of class H3 [5]. Then, we discuss the FPGA-based design of the Hash Table (HT). Finally, we describe the pipeline structure of the FPGA-based HU design.

5.3.1 Hash Unit (HU) on an FPGA

The block diagram of the FPGA-based HU is shown in Figure 5.3. The HU is implemented on an FPGA, and operates as a CPU coprocessor, to speedup hash table operations (insert, delete and

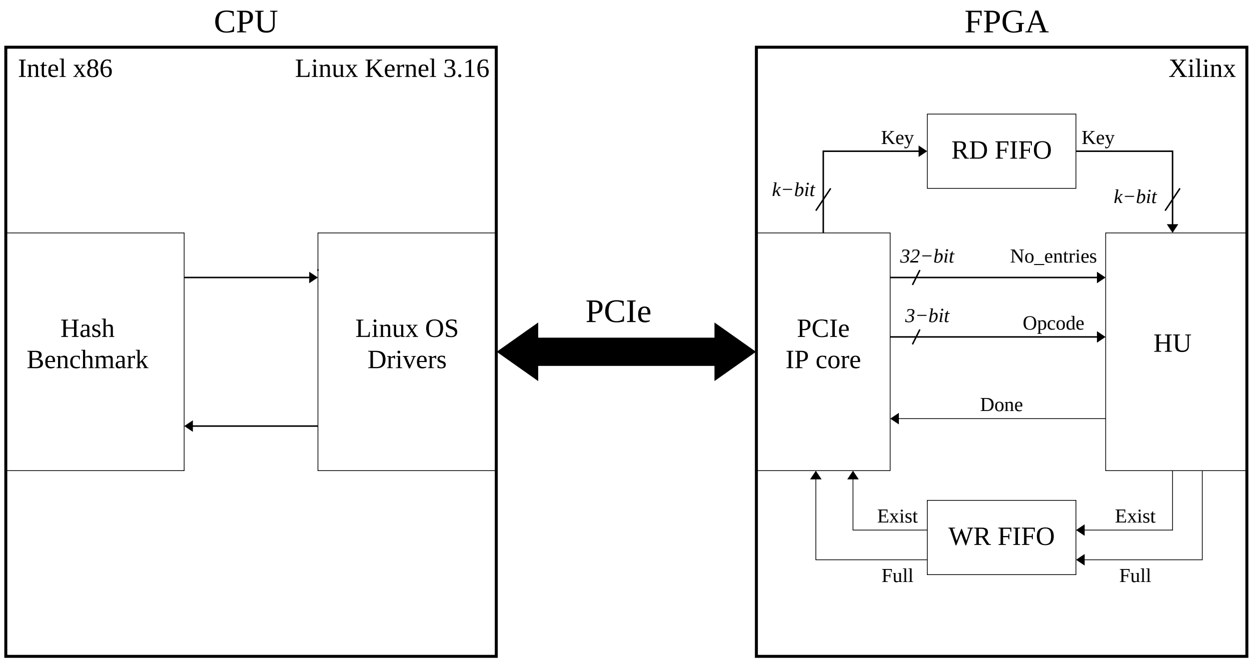


Figure 5.3: Hash Unit CPU-FPGA through PCIe implementation (Modified [7])

lookup). The HU is implemented on the FPGA and communicates with the CPU through a PCIe interface. The CPU runs a virus checking benchmark. Once the CPU reaches a hash operation subroutine call, it sends the hash operation request (in a batch) to the FPGA. The FPGA processes the batch virus check operation and sends the results back to the CPU.

The HU consists of two units: the hash table (HT) and the hash function (HF), as shown in Figure 5.2. It takes a *key* as an input and returns a result based on the hash operation requested (i.e. lookup, insert, or delete). The HU has three inputs and three outputs, as shown in Figure 5.3. Since we focus on a HU that only supports membership queries, the *value* field in not required. The inputs of the HU are:

- Key: A unique value that can be searched and stored in the HT.
- Opcode: A three bit value that indicates the hash operation to be performed in the HU, as shown in Table 5.1.
- No_entries: A 32-bit value that indicates the number of streamed (burst) hash operations to

be performed in the HU.

The outputs of the HU are:

- Done: A signal that indicates the end of streamed (burst) hash operations in the HU.
- Exist: This signal is an output from the HT. It is an outcome signal from a lookup operation. It indicates if the entry exists in the HT.
- Full: It is an output signal from the HT. It indicates that the HT bin is full. We will discuss the details of the Exist and Full signals in the HT in Section 5.3.3.

Operation	Opcode
Replace	100
Insert	011
Delete	010
Lookup	001
no-op	000

Table 5.1: Opcodes for the HT operations (Reprinted [7])

The HU performs four operations:

- Lookup: This operation is performed before insert or delete hash operations, to avoid duplicate entries during insertion, and to confirm membership before deletion. The lookup operation takes a *key* as an input, and queries its membership in the HT. First, it hashes the *key* using the HF to produce a *Bin Index*. The *Bin Index* refers to the bin in the HT where *key* will be located if it is present in the HT. After that, the *key* is searched among all bin entries in parallel. This $O(1)$ operation is possible since the HT bins are implemented as CAMs. If the *key* is found, the lookup operation will report the existence of the *key*. If the *key* is not found, the lookup operation will continue the search in the DRAM (where the complete bin data resides). If the *key* is not present in DRAM, the lookup operation returns that the *key* does not exist.

- Insert: Before an insert operation, a lookup operation will be executed first. If the *key* is found, the insert operation will be aborted, so as to avoid duplication. Otherwise, the new *key* will be inserted in the HT bin. If the HT bin is full, the *key* will be inserted to the corresponding HT bin in the DRAM, at the end of the linked list.

- Delete: Before a delete operation, a lookup operation will be performed for the *key*. This is done to make sure that the *key* indeed exists in the hash table. If the *key* does not exists, the delete operation reports that the *key* does not exist in the hash table. Otherwise, deletion is performed by setting the valid bit (associated with the entry in the HT bin) to 0.

- Replace: a replace operation occurs when there is *a HT miss and a DRAM hit*, in a HT bin. The replace operation resets all the valid bits in a HT bin to '0'. Then, it performs multiple insert operations, to insert the replaced *keys* from the DRAM into the corresponding HT bin.

5.3.2 Hash Function (HF) of class H3

A *key* is the input of the hash function (HF), and a *Bin Index* is the output of the HF, as shown in Figure 5.2. The *Bin Index* will point to the bin in the HT where the *key* will reside. Also, it represents the address of the Block RAM (BRAM) memory in the FPGA, as shown in Figure 5.4 and discussed in Section 5.3.3. The *Bin Index* has $log_2(n)$ bits, where n is the number of bins in the HT. We choose the HF to be of class H3 [5], since it is a good candidate for hardware hash operations. The HF operation is based on the AND and XOR logic, hence it has a fast, efficient implementation. The HF performs a hash operation on an input (the *key* in our design), and a Q matrix with random numbers of class H3 [5]. This Q matrix is stored in latches as shown in Figure 4.2, and is initialized during the HU initialization. The Q matrix can be changed for different applications, resulting in a flexible HF.

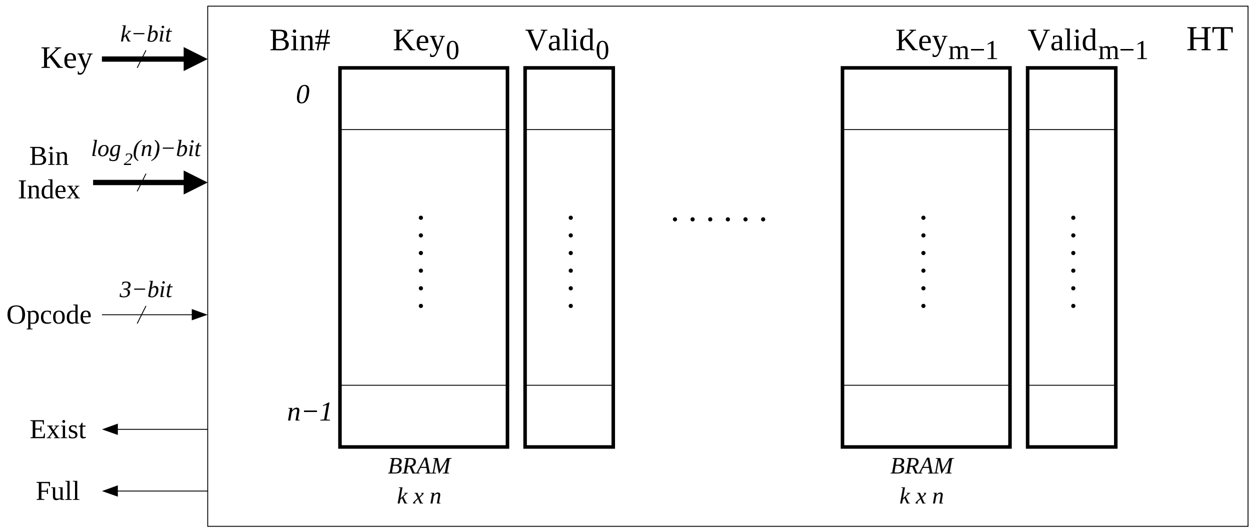


Figure 5.4: Hash Table (HT) structure in the FPGA (Modified [7])

5.3.3 Hash Table (HT)

Typically, software-based hash table (HT) designs use a linked list to store the hash table entries of each bin, as shown in Figure 5.1. We design our HT using CAM memory blocks [20] to provide fast access. We use the FPGA Block RAM (BRAM) memory to design our HT, as shown in Figure 5.4. Each bin in the HT can provide a one-cycle lookup operation, since the whole bin is stored as a CAM as can be searched in parallel. Each entry in the HT bin is stored in separate BRAM memory in the FPGA as shown in Figure 5.4. As we mentioned in Section 5.3.2, the HF generates the *Bin Index* to point to a bin in the HT. The *Bin Index* is the address for all BRAM memory blocks in Figure 5.4. We use the DRAM to extend a HT bin, in case the HT bin is full.

As shown in Figure 5.4, there are four inputs and three outputs in the HT. The inputs of the HT are:

- Key: It is the k-bit input entry to the HT. This input will be stored in one of the FPGA BRAM memory blocks in the HT.
- Opcode: It is the 3-bit opcode which indicates the hash operation to be performed, as shown in Table 5.1.

- Bin Index: It is the *$log_2(n)$* bit address of the FPGA BRAM memory blocks in the HT. This address is the output of the HF as shown in Figure 5.2.

The outputs of the HT are:

- Exist: It is the outcome of the lookup operation. It indicates whether the input *key* exists in the HT bin.

- Full: This output signal is set to *high*, in case the HT bin is full.

We use a *valid* bit associated with each entry in the HT bin, to indicate the validity of each entry in the HT bin, as shown in Figure 5.4. We store these *valid* bits in D Flip-Flops in the FPGA for fast hash operations. In the next paragraphs, we will discuss the hash table operations.

In the lookup operation (the *Opcode* is 01), the HT takes the *key* as its input. Then, the HT searches for *key* in all the FPGA BRAM memory blocks at the *Bin Index* address in parallel. This is performed by looking up all *m* entries (located at address HF(*key*) in the *m* BRAMs) in parallel. These entries (if valid) are compared in parallel, to the lookup *key*. If the *key* is found, the *Exist* signal is set to *high*. Otherwise, the lookup operation continues the search in the corresponding bin in the DRAM, if the pointer to the bin is not *null*, and the *Full* signal is *high*.

In the insert operation, the HT first performs a lookup operation. If the *key* is found, the insert operation will be aborted, to avoid duplication. Otherwise, the *key* will be added to the HT bin, at the *Bin Index* address of one of the BRAMs whose *valid* bit is '0'. The position of the inserted *key* in the bin is pre-computed using a priority encoder. The priority encoder operates in parallel with the lookup operation, prior to the insert operation. It takes all the *valid* bits of the *m* entries of the HT bin as an input, and provides the correct insert location (which is one of the BRAM memory blocks). The insertion is performed by writing the *key* to the HF(*key*) location of this BRAM block, and also writing a '1' in its *valid* bit. In case the HT bin is full, the HT sets the *Full* signal to *high*, and the insert operation inserts the *key* in the linked list of the DRAM bin.

For the delete operation, the HT first performs a lookup operation for the input *key*. This is to ensure that the *key* actually exists in the HT. If the *key* is found, the delete operation simply sets the corresponding *valid* bit to 0, to effectively clear the entry. Otherwise, if the corresponding DRAM bin pointer in not *null* and the *Full* signal is *high*, the lookup operation continues the search in the DRAM bin. If the *key* is found in the DRAM bin, the delete operation removes the entry from the linked list in the DRAM.

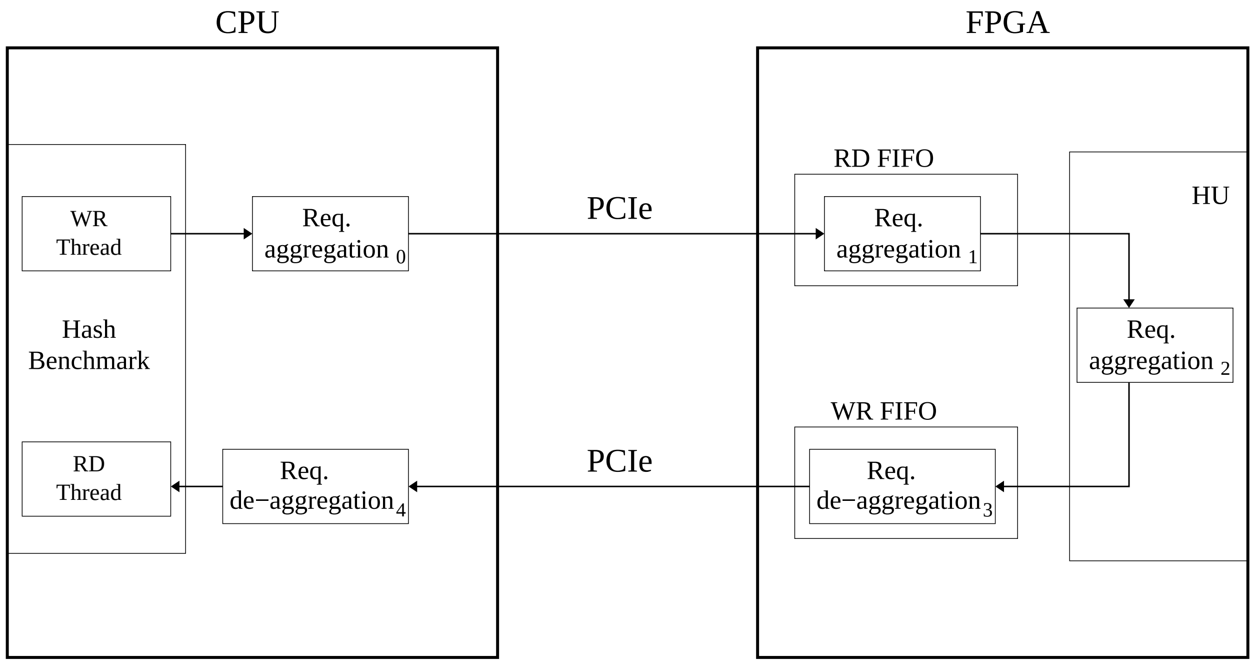


Figure 5.5: Hash operations burst structure in our CPU-FPGA implementation (Modified [7])

In case of *a HT miss and DRAM hit*, the replace operation takes place in the HT. First, it sets all the valid bits in the corresponding bin to '0' value. After that, it inserts the replaced *keys* from the DRAM to the HT.

5.3.4 Hash Unit Pipeline Structure

The HU is implemented on an FPGA to provide fast hash operation through a PCIe bus. The PCIe protocol has a latency overhead. Hence we implement our HU in a pipelined manner in the FPGA, to increase the overall performance of hash operations. The pipeline structure is shown in Figure 5.6.

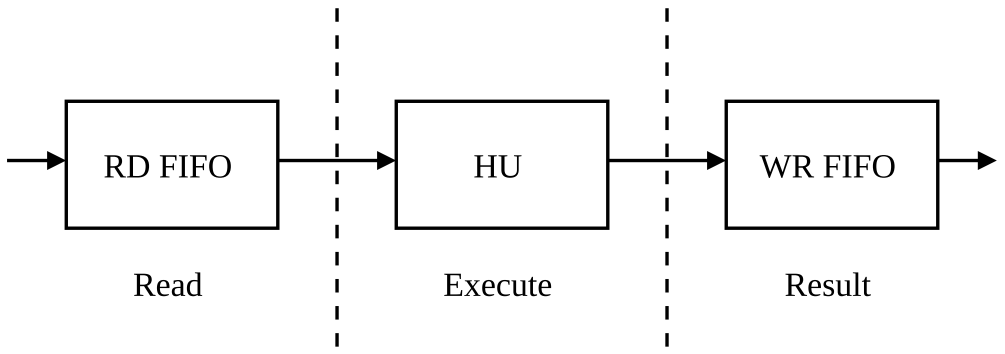


Figure 5.6: Hash operation pipeline stages in the FPGA (Reprinted [7])

The HU pipeline has three major stages:

- Read stage "Read FIFO (RD FIFO)": This stage reads the stream of hash operations (referred as *Req. aggregation* in Figure 5.5) sent from the CPU and stores them in-order in a RD FIFO as shown in Figure 5.5
- Execute stage "HU": This stage reads the hash operations from the RD FIFO and executes the hash operations.
- Result stage "Write FIFO (WR FIFO)": This stage writes the results (referred as *Req. de-aggregation* in Figure 5.5) from the Execute stage into a WR FIFO as shown in Figure 5.5. These hash operations will be send back to the CPU at the same order as they were received.

We have implemented these pipeline stages to allow data streaming, and efficient operation. The CPU streams a burst of hash operations. Each burst has multiple hash operations transmitted in each PCIe payload, to offset the PCIe protocol latency. Therefore, the pipeline structure will increase the performance of the FPGA-based coprocessor. This pipeline implementation is found to be a suitable match for virus checking applications.

As shown in Figure 5.5, the hash benchmark on the CPU runs two threads: the write thread (WR Thread) and the read thread (RD Thread).

The WR Thread sends multiple hash operations in a single burst message as shown in Figure 5.5. In this way, it utilizes the PCIe bus bandwidth effectively, resulting in an increase of the overall performance of the FPGA-based HU implementation.

The RD Thread continuously pulls the results of hash operations from the PCIe bus, as shown in Figure 5.5. Once all the results of hash operations are received, the RD Thread is terminated.

5.4 Experimental Results

In this section, we first present the simulation environment used to validate the FPGA-based coprocessor for the hash unit (HU). Then, we discuss the benchmark setup used for our design verification. Finally, we present the CPU-FPGA simulation results along with a discussion.

5.4.1 Simulation Environment

We implement our design using a CPU and an FPGA, which communicate with each other through a PCIe interface.

We implement our HU on a Xilinx Kintex-7 XC7K325T FPGA board [67] with '-1' speed grade. We use the Xilinx ISE Design Suite [62] for our FPGA design flow. We generate the PCIe IP core using Xilinx CORE Generator System [68]. We use custom Perl [54] scripts to generate Verilog modules for different HU design configurations.

We run our hash benchmarks on an x86 Intel [21] host machine (2GHz Core 2 Duo CPU with 4MB Cache and 4GB DDR2-800MHz DRAM). We use the Linux (Kernel 3.16) [69] operating system (OS) on the CPU. We use Xillybus [70] Linux drivers to read and write to the FPGA board through the PCIe bus. These drivers are included in Ubuntu [71] distributions.

5.4.2 Benchmark Setup

To construct a hash benchmark for a virus checking application, we need to model the hash lookup operations for the virus checking application. The YCSB benchmark uses a zipfian distribution. The zipfian distribution is not a good model for the virus checking application, since it has high occurrences for many viruses in our virus checking application. Therefore, we need to construct a new model (based on the zipfian distribution) for the virus checking application in a file system.

In our virus checking application model, we need to make sure that each file system (including viruses) has been checked at least once. So, a function for the number of occurrences (*NOCC*) of any file system can be expressed as follows (α and β are constants):

$$NOCC(x) = 1 + \frac{\beta}{x^{\alpha}} \tag{5.1}$$

Let L be the total number of occurrences, and n is the database size for the file system. We can solve β from Equation 5.1, since L is the integration of *NOCC(x)* function:

$$L = \int_{1}^{n} NOCC(x)dx = \int_{1}^{n} (1 + \frac{\beta}{x^{\alpha}})dx$$

Once we solve β, the final Equation of the *NOCC(x)* function will be as follows (we call it as *modified zipfian (m-Zipf) model*):

$$NOCC(x) = 1 + \frac{(L-n)(1-\alpha)}{(n^{1-\alpha}-1)}.\frac{1}{x^{\alpha}} \tag{5.2}$$

We vary α in Equation 5.2, to get a good model for the file systems in our experiments. As shown in Figure 5.7($\alpha = 0.9$), the *NOCC* has a high maximum value of 29727 for the first file index, and a value of 60 after the 1000^{th} file index. This value of α is not a good representation for the file systems, since a typical file system will not have $\sim 30,000$ copies of a file. For a value of $\alpha = 0.1$ (Figure 5.8), the *NOCC* has a high maximum value of 10 for the first file index, which is a reasonable representation of the file systems, but more than 50,000 files have 4 occurrences, which is not typical in file systems as well. As a result, we choose $\alpha = 0.5$ and multiply (and round) the first 1000 files by 0.1, to reduce the number of occurrences of the first 1000 file systems (as shown in Figure 5.9). In this case, the maximum occurrence count (for the first file) is 72 and the 1000^{th} file index has 2 occurrence, which are reasonable values for a typical file system.

We use the Intel [21] Performance Monitoring Unit (PMU) to measure the system performance. We compute the speedup of the HU performance by the ratio of the x86 time stamp counter (TSC)

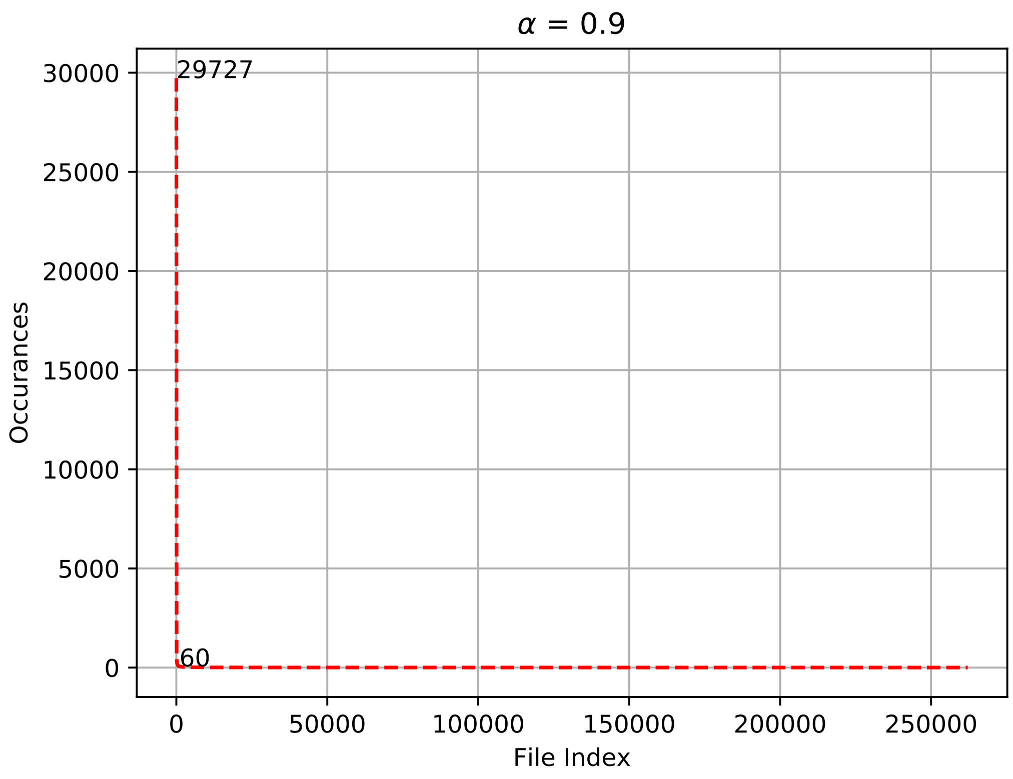


Figure 5.7: Plot of *NOCC(x)*: $L = 1M$, $n = 262144$, and $\alpha = 0.9$

when hashing is done in software, without the HU ($\mathrm{SW_{TSC}}$), versus the x86 TSC when hashing in done with the HU ($\mathrm{HW_{TSC}}$).

$$Speedup = \frac{SW_{TSC}}{HW_{TSC}} \tag{5.3}$$

For our HU simulations, we vary the following parameters:

- The number of lookups per a single burst in the PCIe bus: this is varied from 1 to 256 lookups in our experiments. Each lookup *Key* contains 128 bits (MD5 hash virus signature size).
- HT size: this is varied from 512kB to 1520kB, the maximum value supported by the board.
- Database (DB) size: this is varied from 65,536 to 262,144 MD5 hash signatures. The

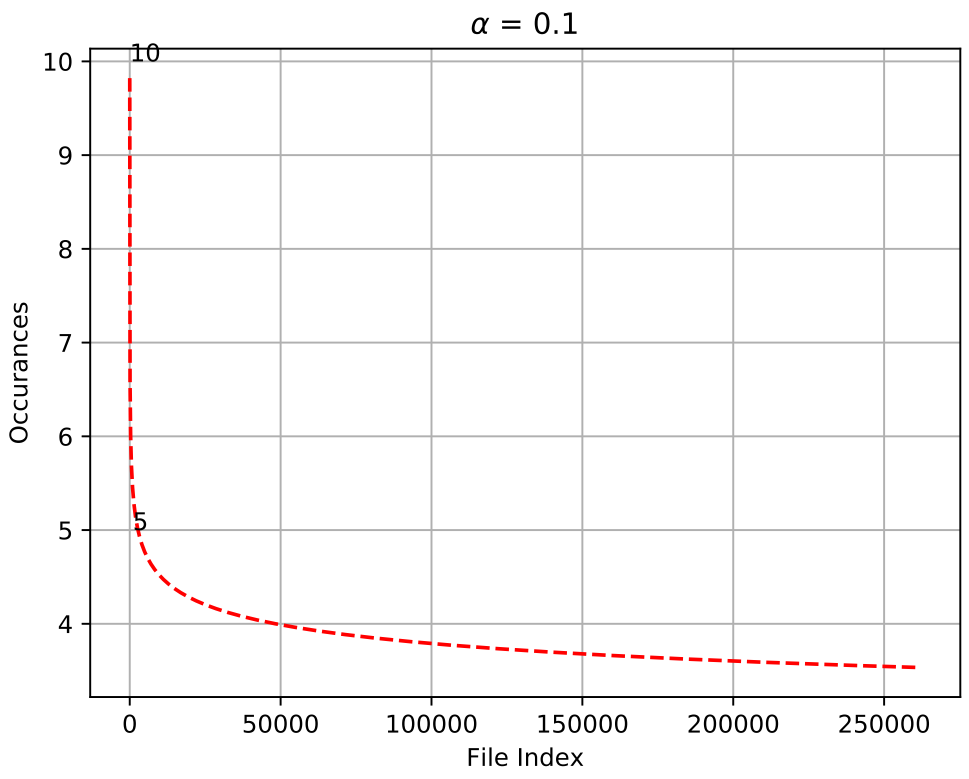


Figure 5.8: Plot of *NOCC(x)*: $L = 1M$, $n = 262144$, and $\alpha = 0.1$

MD5 hash signatures are obtained from VirusShare [63]. We used VirusShare_00147.zip (131,072 MD5 hash signatures), VirusShare_00148.zip (131,072 MD5 hash signatures), and VirusShare_00149.zip (65,536 MD5 hash signatures) files for our simulations. These are actual representative virus definition files. The length of each MD5 virus signature is 128-bit.

As mentioned in Section 5.3.4, the PCIe protocol has a latency overhead. This latency is about 8K clock cycles on our host machine (Intel 2GHz Core 2 Duo) for a single lookup operation. The latency stays constant till 32 lookup operations (i.e. 512B, since each lookup is 16B in length). After 32 lookup operations, there is a minimal increase of about 2% in the latency compared to a single lookup operation. The latency stays constant again for another 32 lookup operations. In this manner, after every 32 lookup operations, there is another small increase in the latency. Therefore, increasing the number of lookup operations per burst is expected to increase the performance.

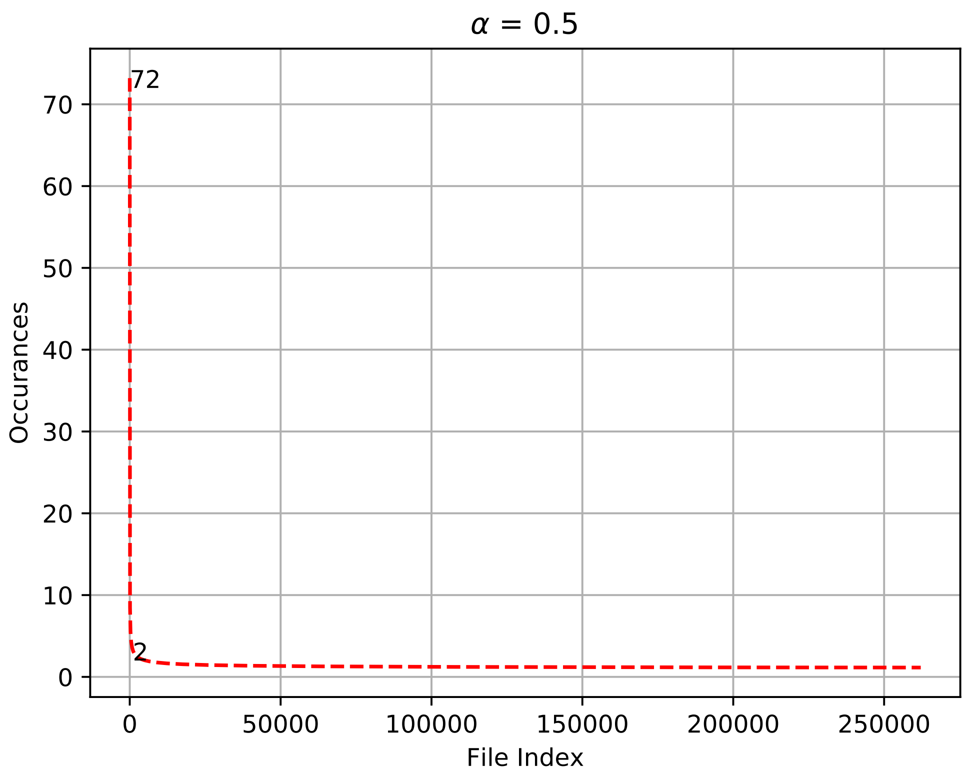


Figure 5.9: Plot of *NOCC(x)*: $L = 1M$, $n = 262144$, and $\alpha = 0.5$

Next, we present our simulation performance results along with a discussion.

5.4.3 Results and Analysis

In our results, we compute the average speedup of each experiment over 100 runs. Results are shown in Figures 5.10, 5.14, 5.15, 5.16, 5.17, and 5.18.

In the following experiments, we apply lookup hash operations for MD5 hash virus signatures, that have a 100% of virus hits (except for the last experiment, where we vary the percentage of virus hits). Therefore, we can measure the performance of the FPGA-based HU in the worst case, where the HT replacement is applied.

In Figure 5.10, the HT size is chosen to be 512kB. We vary the number of bins (n) and the number of entries per bin (m) in the HT. In this experiment, we explore the effect of different

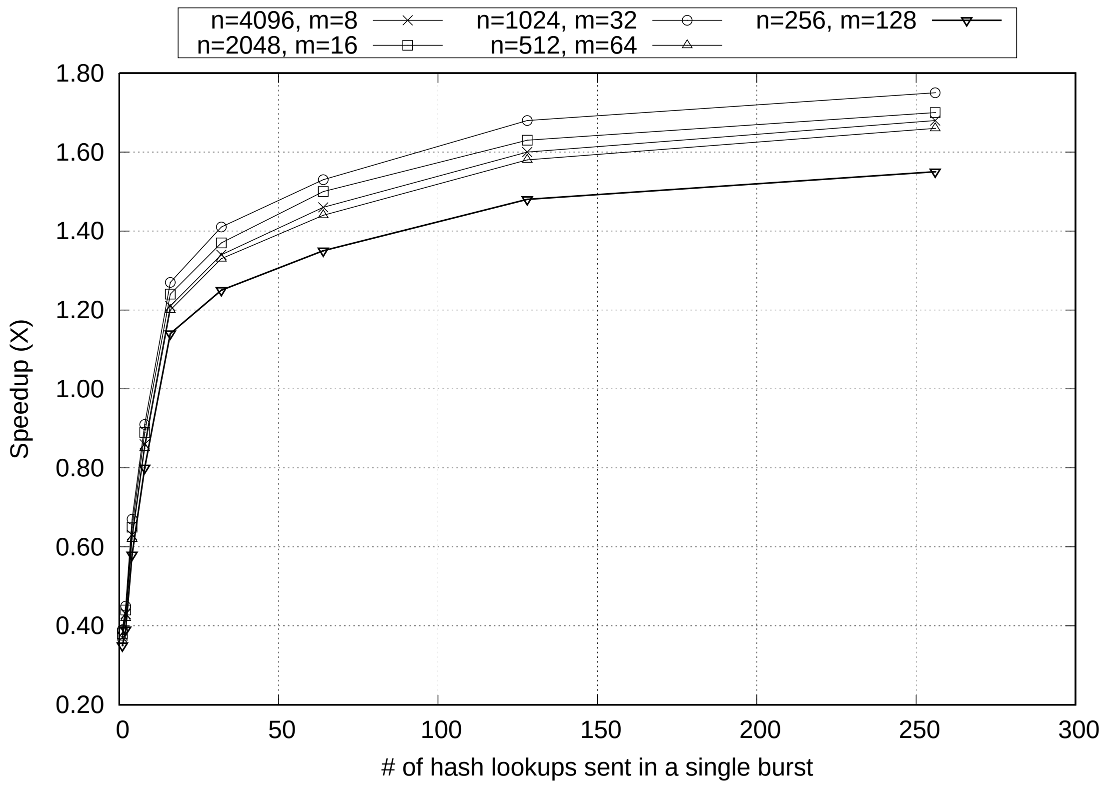


Figure 5.10: Lookup hash operations for DB = 65,536 MD5 virus signatures, the HT size is 512kB – Varying: the number of bins (n) and the number of entries/bin (m) in the HT. (Modified [7])

combinations of n and m for the same HT size. The HT has more DRAM accesses when $n < 1024$, because collisions increase. Also, the HT utilizes less number of BRAMs in the FPGA, which optimizes the locality of the BRAMs. When $n > 1024$, the i^{th} key of all bin in the HT (key_i) will be stored into two or more BRAMs in the FPGA, which will increase the routing delay in the FPGA. As a result, it will increase the hash lookup delay. Therefore, the *sweet* spot is when $n = 1024$. In general, the performance does not have a strong dependency on n and m. For larger values of HT size, some of the (n,m) values caused the FPGA to run out of LUT resources. In the following experiments, we use $n = 1024$ for our HT.

From the previous experiment, we notice that the performance of the HU with larger number of bins (n) is slightly better than the performance of the HU with smaller number of bins. Bad

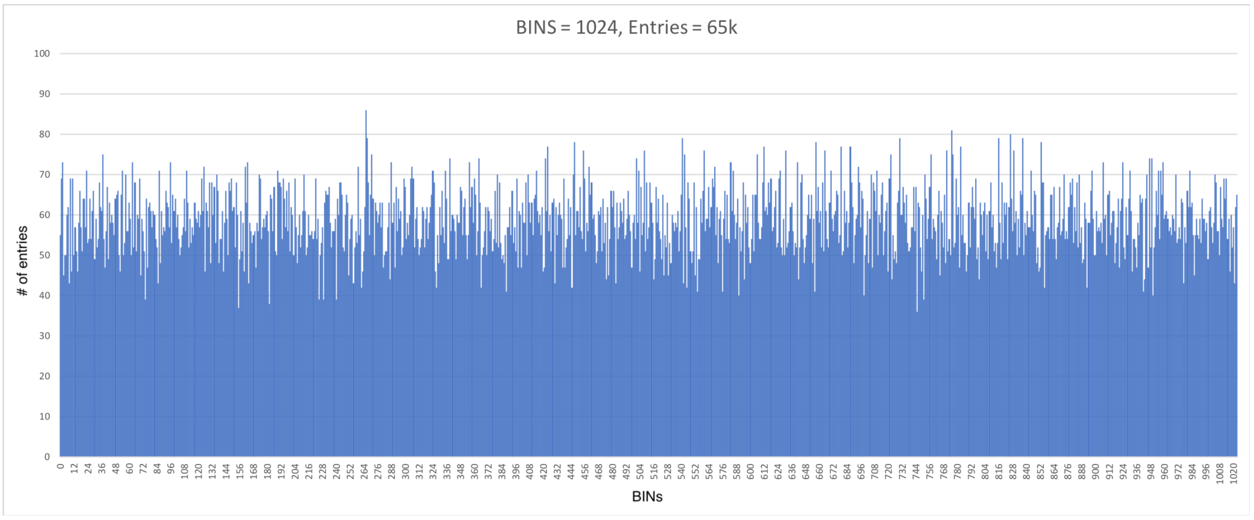


Figure 5.11: Hash table (HT) occupancy for DB=65k entries and HT=512kB ($n = 1024$ bins and $m = 32$ entries).

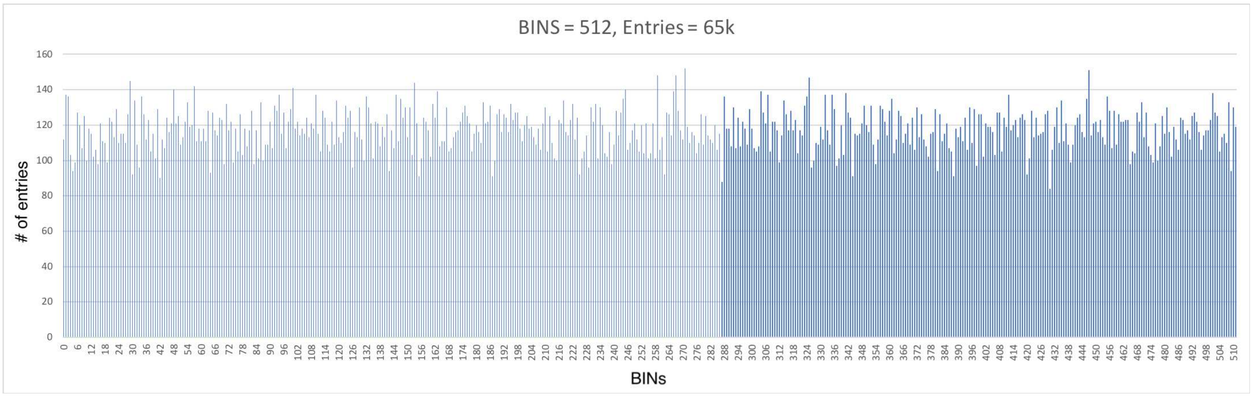


Figure 5.12: Hash table (HT) occupancy for DB=65k entries and HT=512kB ($n = 512$ bins and $m = 64$ entries).

occupancy[4] of the hash entries between different bins in the hardware HT can be a reason for that. Therefore, we analyze the occupancy of the hardware HT bins for different number of the HT bins of the same DB size of 65k entries. Figures 5.11, 5.12, and 5.13 report bin occupancy. The x-axis represents bin number and the y-axis represents the number of entries in each bin. As

[4]We define occupancy as the number of entries in each hash table bin.

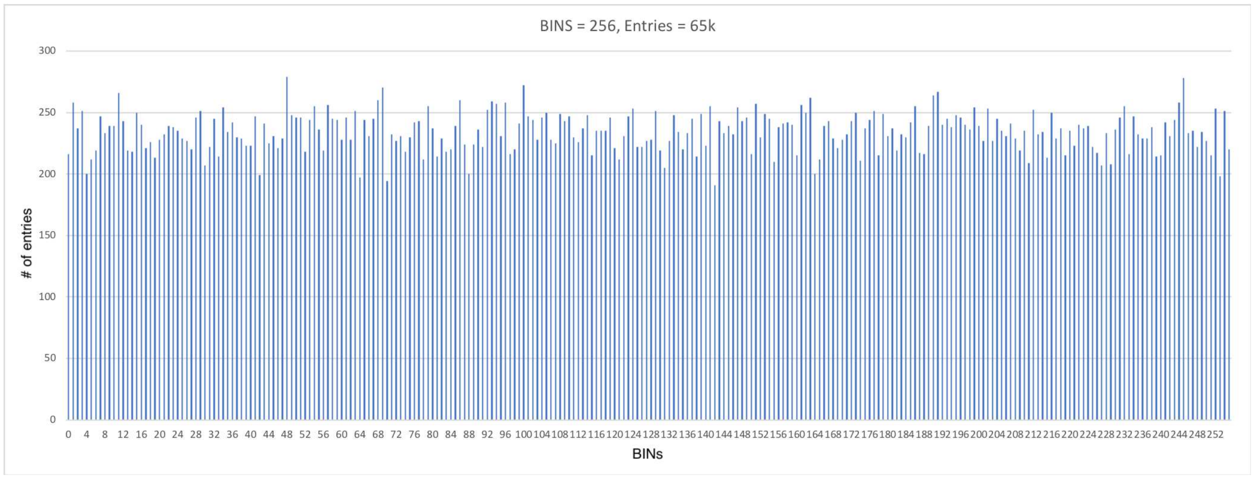


Figure 5.13: Hash table (HT) occupancy for DB=65k entries and HT=512kB ($n = 256$ bins and $m = 128$ entries).

shown in Figures (5.11, 5.12, and 5.13), the occupancy of each bin in the hardware HT is 100% (the hardware HT is full). Therefore, the occupancy each HT bin will be 100% for larger DB sizes as well. As a result, the HF used in our HU is a good fit for hash operations.

In Figure 5.14, we use a DB size of 65,536 of MD5 hash virus signatures. For a HT size of 1520kB and 256 lookups in a single burst, we demonstrate a maximum speedup of 5.37×. Note that a speedup of ∼5× is obtained even for 64 lookups per burst. In the case when HT sizes are 1520kB, 1280kB, and 1024kB, the DB entries almost entirely fit in the HT. Hence the speedup is high. For a HT size of 512kB and 768kB, there are more DRAM accesses, and hence the speedup is reduced. In the 768kB size, we still demonstrate a healthy maximum speedup of 92% over the software-based hashing. We notice that the performance of our FPGA-based HU implementation for a single hash lookup operation (burst size is 1) is worse than the software-based implementation, (for HT sizes of 512kB, 768kB and 1024kB) as expected. This is due to the latency overhead of the PCIe protocol. Therefore, implementing a pipelined structure with burst lookups achieves better performance for hash operations.

For the DB size of 131,072 of MD5 hash virus signatures (as shown in Figure 5.15), the DB size is larger than all the HT sizes. For the HT size of 1520kB, the maximum speedup is 2.42×,

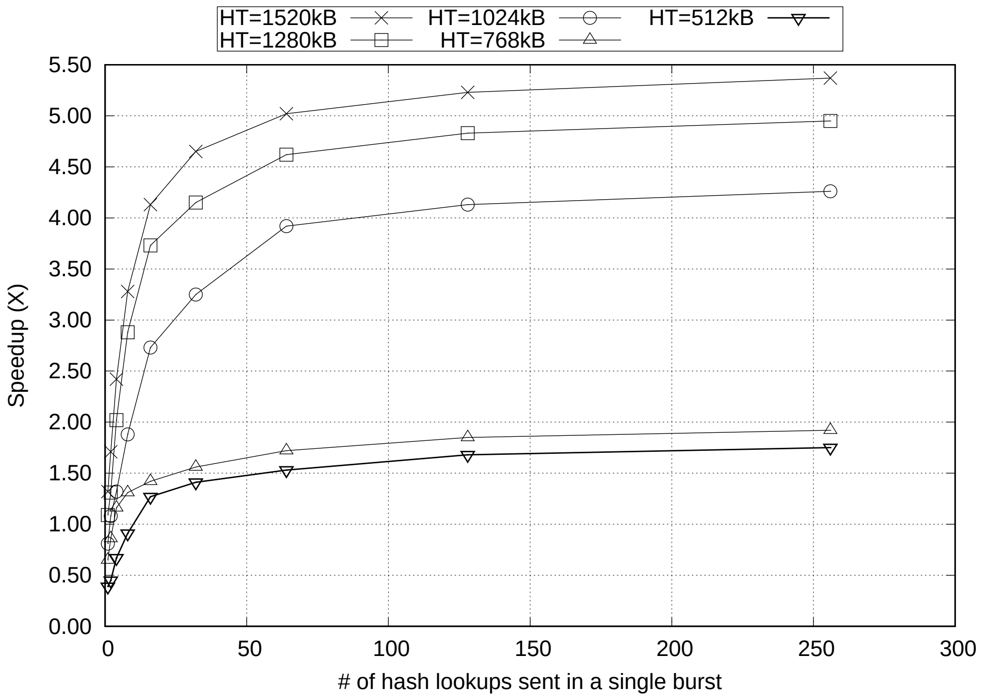


Figure 5.14: Lookup hash operations for MD5 hash virus signatures – Varying burst and HT sizes. DB = 65,536 MD5 virus signatures (Modified [7])

while the maximum speedup is 79% for the HT size of 768kB. This shows that our scheme scales well even when the DB cannot fit in the HT.

As shown in Figure 5.16, we demonstrate a maximum speedup of 89% for a HT size of 1520kB (and a DB size of 196,608 MD5 hash virus signatures). This speedup is achieved for 256 lookup hash operations per burst. For the HT size of 768kB, a maximum speedup of 36% is obtained. When the DB size is higher, most of the MD5 hash virus signatures will be stored in the extended DRAM bins, reducing the speedup. However, the speedups are still healthy, and are achieved for modest (32 or higher) burst sizes.

In Figure 5.17, we use a DB size of 260,144 MD5 hash virus signatures. We achieve a maximum speedup of 54% when we transmit 256 hash lookup operations in a single burst, with a HT of size 1520kB. For a HT size of 768kB, the maximum speedup is 31%. For this HT size, the DB size is 5.33× larger than the HT size. This demonstrate that even for large number of DB entries, the FPGA-based HU implementation can achieve good speedups compared to the software-based

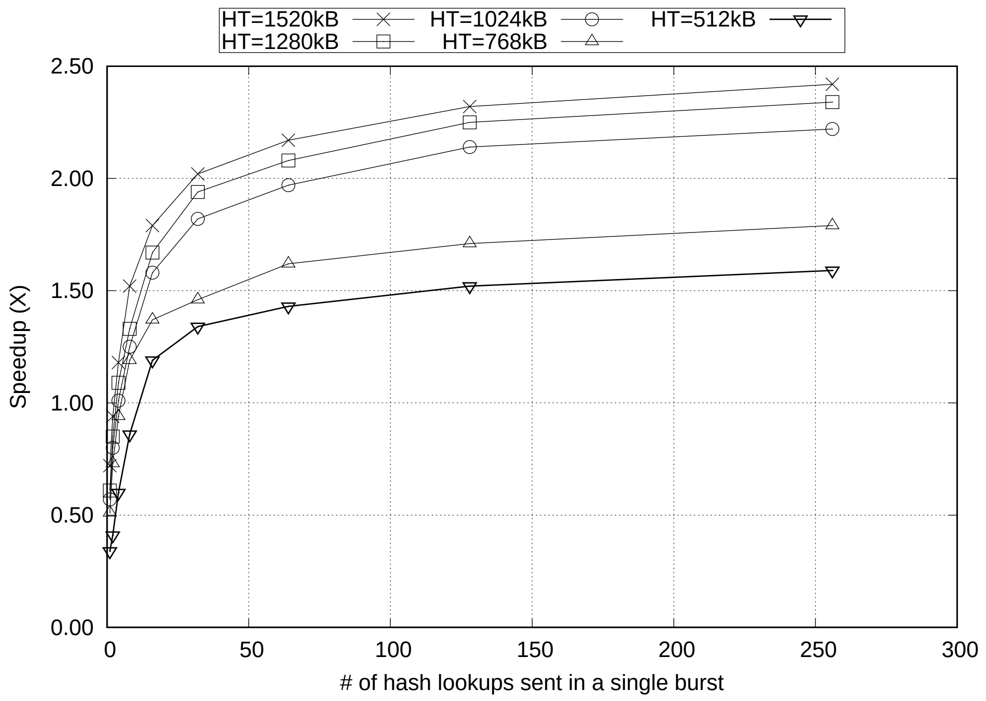


Figure 5.15: Lookup hash operations for MD5 hash virus signatures – Varying burst and HT sizes. DB = 131,072 MD5 virus signatures (Modified [7])

hashing implementation. Using more advanced FPGA boards with larger BRAM can further improve our results.

In previous experiments, we have applied lookup hash operations for MD5 hash virus signatures, that have a 100% of virus hits. In the following experiment, we vary the percentage of virus hits in the hash lookup operations. In Figure 5.18, we use a DB size of 260,144 MD5 hash virus signatures, and a HT size of 1520kB. We achieve a maximum speedup of 2.01×, when the virus hit is 1%. Also, the speedup is more than 30% when the hash lookups in a single burst is 64.

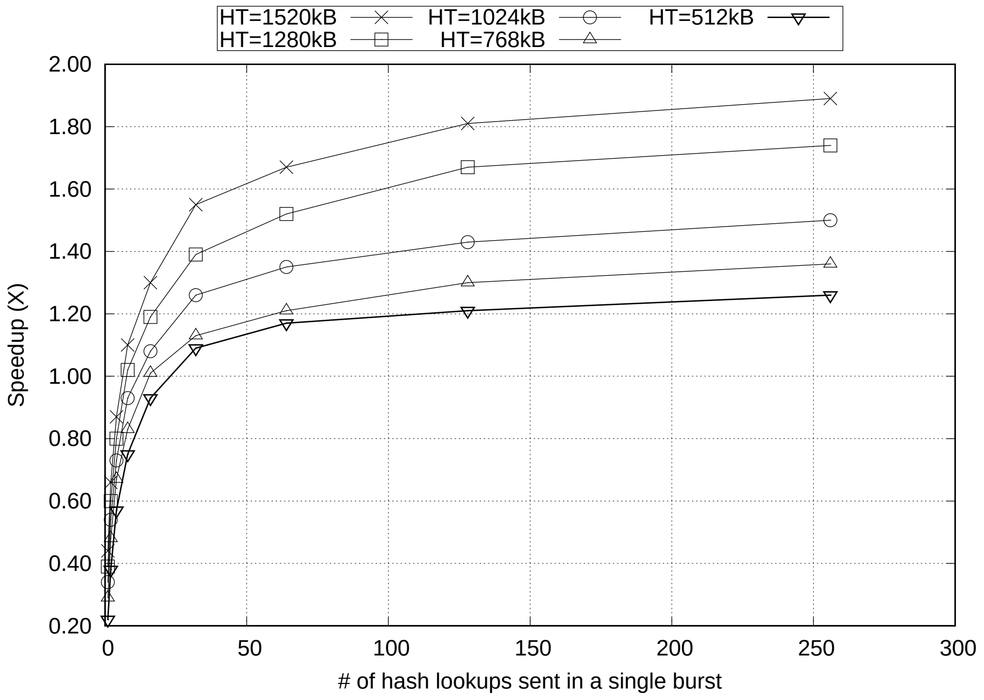


Figure 5.16: Lookup hash operations for MD5 hash virus signatures – Varying burst and HT sizes. DB = 196,608 MD5 virus signatures (Modified [7])

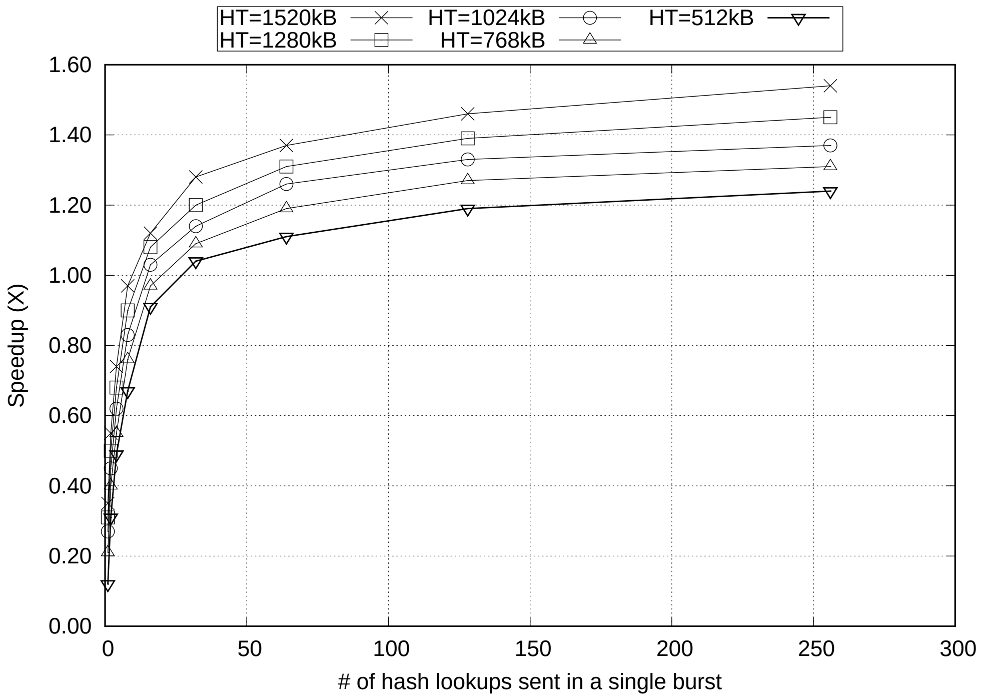


Figure 5.17: Lookup hash operations for MD5 hash virus signatures – Varying burst and HT sizes. DB = 262,144 MD5 virus signatures (Modified [7])

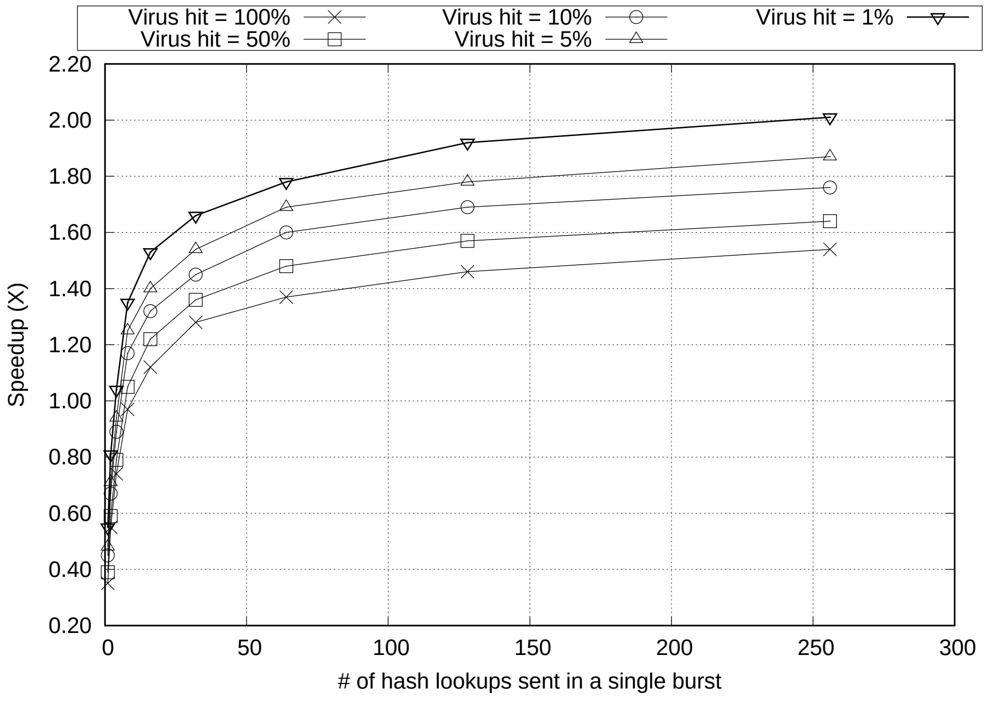


Figure 5.18: Lookup hash operations for MD5 hash virus signatures – Varying the percentage of virus hits. DB = 262,144 MD5 virus signatures. HT size = 1520kB.

5.5 Chapter Summary

In this chapter, we present an FPGA-based hardware hash unit (HU) design for hashing operations. We implement each bin of the hash table (HT) as a CAM. We have verified the correctness of all hash operations in the real-time system design. We demonstrate a speedup of up to 5.37$\times$ for the FPGA-based HU implementation over the software-based hashing implementation, in the context of a virus checking application. Higher speedups are expected with more advanced FPGA boards.

6. PAU: A PROGRAMMABLE ARITHMETIC UNIT FOR USE IN MODERN MICROPROCESSORS

In this chapter, we present a programmable arithmetic unit (PAU) for use in modern microprocessors. We start with a background in Section 6.1. Then, in Section 6.2, we discuss the previous work. In Section 6.3, we present our proposed PAU design. After that, we present our experimental results in Section 6.4. We conclude in Section 6.5.

6.1 Background

Modern microprocessors are designed for general purpose applications. Today's applications require complex arithmetic computations such as signal processing, image processing, scientific computation, along with data-intensive applications like networking, cloud computing and web-based search. Arithmetic applications are implemented in high-level languages, and run in software. Arithmetic applications require different kind of mathematical computations. The performance of arithmetic applications in modern microprocessors are limited by memory accesses, instruction execution latencies and data dependencies. As a result, new techniques need to be explored to enhance the speed of arithmetic computations in modern microprocessors.

Arithmetic applications in today's applications comprise a lot of operations. These operations require a high memory utilization and a high number of CPU cycles in modern microprocessors. Many special function units (SFUs) in modern microprocessors are used to speedup common algorithms or operations for applications such as memory management, integer operations, floating point operations and vector operations. In order to run applications containing different arithmetic computations, modern microprocessors compile these applications and run their instructions. As we demonstrate in the chapter, implementing a flexible arithmetic unit in modern microprocessors can improve the performance of the arithmetic applications significantly. We propose a programmable arithmetic unit that includes multiple intellectual property blocks (IPs), a programmable control logic and a fast Network-on-Chip (NoC) data fabric. The proposed programmable arithmetic unit can yield a significant speedup for a series of arithmetic-intensive applications.

As an example, multimedia content are heavily used in the internet and storage space [72]. Therefore, efficient image and video compression is required, using formats such as Joint Photographic Experts Group (JPEG). JPEG is widely used in lossy image compression [73], and employs a discrete cosine transform (DCT) engine. The DCT is an arithmetic-heavy computation that decomposes an image into different frequency components. Typically, the DCT computation is done in software, which reduces the performance of JPEG compression and increases the memory utilization.

Modern microprocessors do not have a flexible programmable arithmetic unit. In this chapter, we propose a novel programmable arithmetic unit (PAU) as a new SFU for use in modern microprocessors. The programmable arithmetic unit consists of three major blocks: arithmetic tiles (IP blocks), an FPGA controller (control logic) and a fast ring-based network-on-chip (NoC) data fabric [8]. The IP blocks in the PAU can be adders, subtractors, multipliers, comparators, etc. The PAU can have one or more of the same tile. The FPGA controller implements programmable control logic. It allows different arithmetic application to be embedded in the PAU. We build the reconfigurable logic of the FPGA controller using a LUT-based design, like a traditional FPGA. The FPGA controller and the tiles communicate via a fast ring NoC data fabric [8]. We design our ring NoC data fabric to run at a speed of up to 20GHz, that is about 20$\times$ the FPGA controller and tiles speeds. This speed of the ring NoC provides a high bandwidth compared to a state of the art mesh-based NoC [8]. We test the PAU with three applications and measure their delay, area and power. We compare the performance of the PAU with a software-based implementation. We demonstrate a significant speedup of up to 832.3$\times$ using the PAU over the software-based implementation. We compare the area of the PAU with a 32kB of L1D cache of the same technology node. Also, we calculate the increase of power consumption of the PAU compared to an average power of an Intel [21] i7 processor.

The key contributions of this chapter are:

- Design a programmable arithmetic unit (PAU) for use in modern microprocessors. The PAU is flexible and can be programmed for different arithmetic applications. The PAU enables different arithmetic applications to be embedded in the modern microprocessors. To the best of our knowledge, this has not been undertaken to date.

- We implement three arithmetic application examples in the PAU and measure their delay, area and power. Also, we compare the performance of the PAU with a software-based implementation. We demonstrate a significant speedup of the PAU of up to 832.3$\times$ over the software-based implementation, with a minimal power increase and an acceptable area increase, while varying the configuration of the number of tiles, the FPGA controller and the ring NoC size in the PAU.

- We simulate our fast ring NoC data fabric in Synopsys HSPICE [45] in a 16nm technology, using PTM [46] high-performance model card. RC parasitics of the ring NoC are modeled as well, and extracted using Synopsys Raphael [47]. We design the arithmetic tiles (IP blocks) of the PAU using a 45nm technology library [6] in Synopsys DC [74] RTL synthesizer and scale the delay, area, and power to the 16nm technology.

- We design our FPGA controller (control logic) in Xilinx Vivado [75] using Kintex UltraScale+ FPGA boards (using a 16nm technology).

- We compare the area of the PAU with the area of a 32kB L1D cache in 16nm, and the power of the PAU to the average power of the Intel i7-5600U processor.

The rest of the chapter is organized as follows. In Section 6.2, we discuss related previous work. Section 6.3 describes our PAU design approach. Then, we present our experimental results in Section 6.4. We summarize the chapter in Section 6.5.

6.2 Previous Work

A survey paper on coarse grained reconfigurable architectures (CGRAs) has been presented in [76]. In [76], the authors focused on architectures that have either (or both) temporal or spatial granularity. They considered architectures that have granularity in spatial reconfiguration at fixed functional unit or above, and architectures that have granularity in temporal reconfiguration at region level or above. In the CGRA study, there is a list of reconfigurable architectures. The reconfigurable architectures most related to our work (a programmable arithmetic unit, PAU) in the CGRA study are the works presented in [77, 78, 79]. In the Colt [77] architecture, homogeneous functional units (FUs) connected in a mesh network that perform integer operations. The Colt architecture provides a simple reconfigurable control flow for applications that require complex looping structures and conditional execution. The authors of [78] presented a specialized reconfigurable architecture for mobile wireless protocols, which makes it application specific. In their architecture, there are multiple reconfigurable processing units (RPUs) grouped in clusters of four. The communication between clusters is configurable using SRAM-based switches. In [79], the authors proposed a polymorphic pipeline array (PPA) that has groups of four processing elements (PEs). Each group is called a core. The PEs in a core share a cache, a loop-buffer and a register file. In each PE, there is an arithmetic unit (as an FU) for integer operations. The FUs communicate through a mesh network. The PEs are connected via n columns of shared memory-bus. Each column memory-bus is configurable, and used to reduce the load latency for applications that do not need core sharing.

Unlike [78], the PAU is general, and not application specific. In contrast to [77], the PAU allows complex control and flexible operations in the tiles. Finally, the PAU uses extremely low-latency interconnect unlike [79]. The PAU is programmed for different arithmetic applications. The PAU uses a fast ring-based NoC to connect different tiles (IP blocks) with an FPGA controller.

There is a reconfigurable arithmetic processor (RAP) proposed by [80], that can be used for multiple-instructions/multiple-data (MIMD) computations. RAP uses multiple add and multiply units that communicate via a crossbar switch. RAP uses a serial input implementation to the crossbar switch, which reduces the data bandwidth. A reconfigurable modular arithmetic FPGA implementation has been proposed by [81] for public-key cryptosystems, by changing connections between carry-save adders. In [82], a dynamically reconfigurable multi-operation arithmetic unit is proposed, basing on multiply-add-fused unit for matrix algorithms.

In contrast to [80] and [82], the PAU uses different type of tiles (IP blocks) such as adders, subtractors, multipliers, comparators. The PAU is programmable using an FPGA-like control logic (FPGA controller), using a mesh-like network instead of the cross-bar of [81]. The FPGA controller and the tiles in the PAU communicate through a fast ring NoC data fabric that provides a high data throughput than a state of the art mesh NoC [8] or a cross-bar switch.

6.3 The PAU Design Approach

In this section, we discuss our proposed programmable arithmetic unit (PAU) for use in modern microprocessors. First, we start with an overview of the PAU. After that, we discuss the PAU components: the tiles, the FPGA controller and the ring NoC data fabric. Then, we discuss the PAU design flexibility. Finally, we discuss the arithmetic applications used to benchmark the PAU.

6.3.1 Overview: PAU

Our programmable arithmetic unit (PAU) is designed to accelerate arithmetic application that use intensive arithmetic operations in a modern microprocessor. We embed the PAU in modern microprocessors as a new SFU, as shown in Figure 6.1. The PAU consists of three major blocks – the tiles (IP blocks), the FPGA controller (we refer to it as F_C) and the ring NoC data fabric ($Ring$), as shown in Figure 6.2. We refer to a tile as T_i. There are n total tiles in the PAU. The figure shows that the F_C and the tiles communicate via the $Ring$ NoC (which can be can be alternatively replaced by a $m \times m$ ring-based mesh NoC). The tiles in the PAU can be adders, subtractors, multipliers, comparators, or any complex IP block. We can have multiple tiles of the same type in the PAU. The tiles in the PAU can operate simultaneously. The F_C is the control logic in the PAU. We use a LUT-based design like a traditional FPGA, to construct our F_C. The F_C can be programmed for different arithmetic applications. The $Ring$ NoC [8] is used to connect the F_C with the tiles in the PAU. Our $Ring$ NoC runs at a high speed, providing a much higher bandwidth than the state of the art mesh NoC. We will discuss in details the tile, the F_C and the $Ring$ NoC in Section 6.3.2, Section 6.3.3 and Section 6.3.4, respectively.

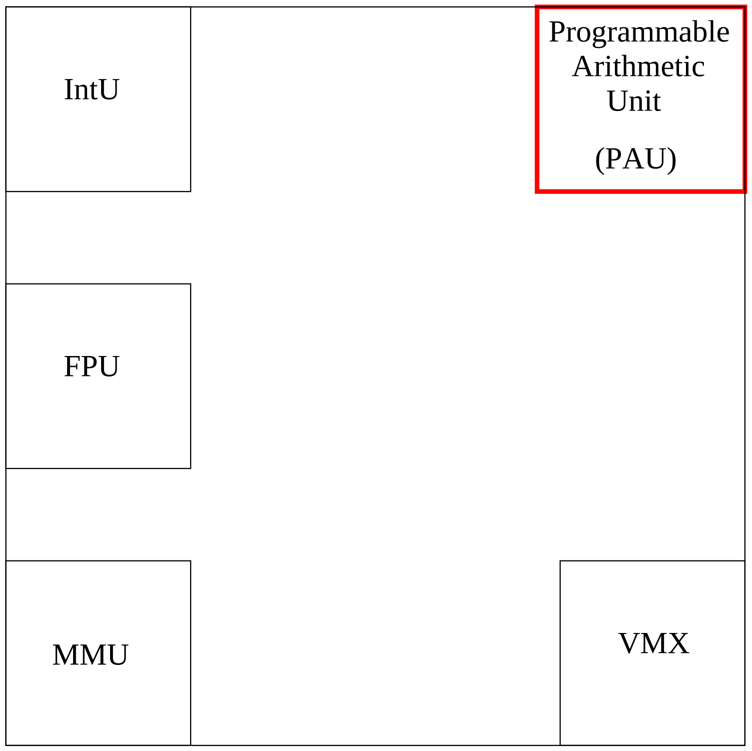


Figure 6.1: CPU units with the PAU

6.3.2 Tiles

The tiles T_i in the PAU (as shown in Figure 6.2) are custom-designed IP blocks designed for efficiency. These tiles can be any common arithmetic IP block (such as adders, subtractors, multipliers, comparators). We can have one or more instance of the same tile T_i in the PAU. For example, in the PAU, we can have 5 multipliers (T_1 through T_5) and 7 adders (T_6 through T_{12}) for a total of $n = 12$ tiles. Each tile T_i in the PAU can perform an operation in the same clock cycle as any other tile, to increase the performance of an arithmetic application. The order of operations of the tiles is specified by the F_C in the PAU, to implement a general arithmetic application. In an arithmetic application, the number of available tiles (resources) in the PAU determines the number of cycles required to produce a final result for an arithmetic computation.

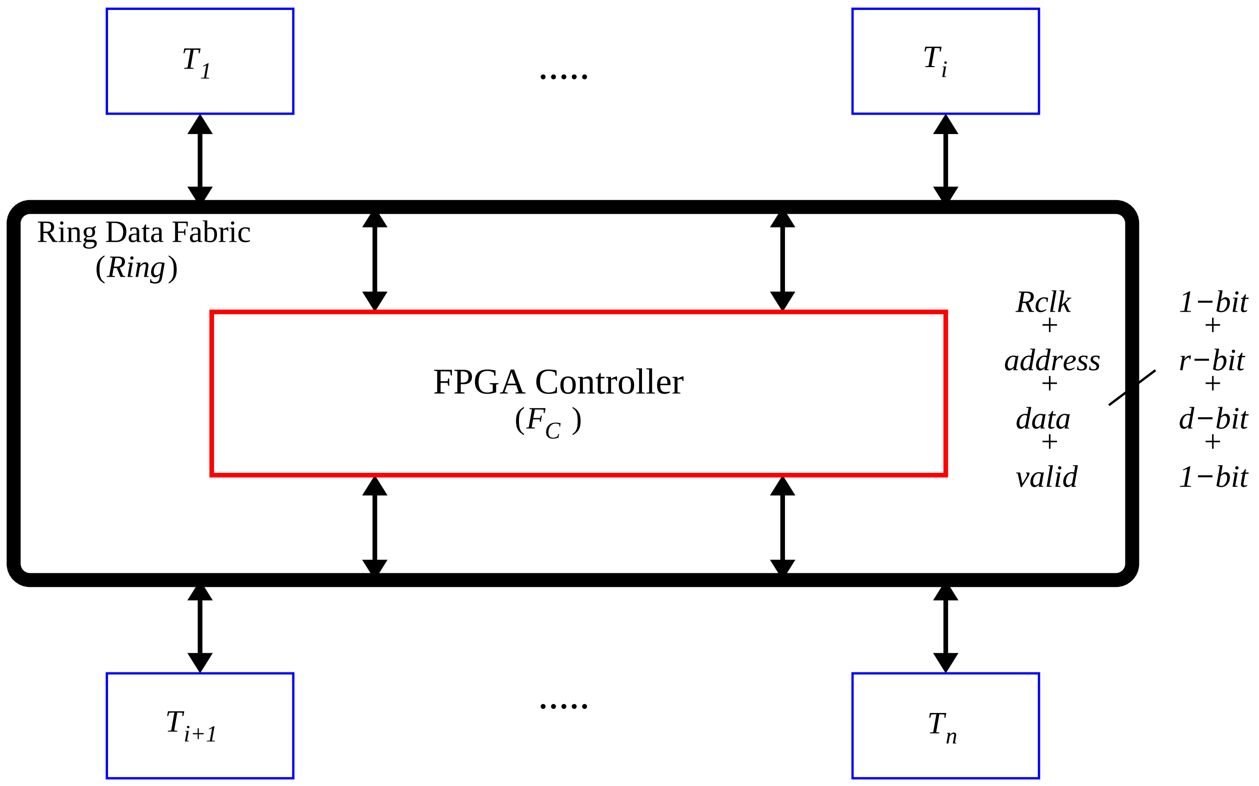


Figure 6.2: The PAU General Architecture showing Single Ring

6.3.3 The FPGA Controller (F_C)

We design our FPGA controller (F_C) using a LUT-like traditional FPGA fabric. Hence the F_C is reconfigurable and can be programmed for different arithmetic applications. The reconfigurability feature in the F_C enables different arithmetic applications to be embedded in the PAU. The F_C is the state-machine, which sequences the operations of the tiles in the PAU, for any arithmetic application. The F_C can assign at most n tiles during any cycle, for any arithmetic application. The F_C also controls the data flow between the tiles in the PAU, via the fast $Ring$ NoC data fabric. If the number of hops between two tiles is h, we assume it takes h cycles to route data between the two tiles. Usually a larger number of LUTs in the F_C allows the PAU to perform more complex arithmetic computations that span a large number of cycles.

6.3.4 The Ring NoC Data Fabric ($Ring$)

Our fast ring NoC data fabric in the PAU, is based on the idea presented in [8]. We use a two dimensional $m \times m$ mesh-based NoC to communicate between the F_C and the tiles in the PAU (as shown in Figure 6.3 for $m = 4$). Each ring in our mesh NoC has a clock ($Rclk$), destination $address$ (*r-bit*), $data$ (*d-bit*) and one $valid$ wires (bits). The $Rclk$ operates at a speed of up to 20GHz, which is much faster than the speed of the tiles in the PAU. In the NoC, there are three types of stations: *insertion/extraction* stations (*IES*, represented as $\times$ in Figure 6.3), *junction* stations (*JS*, represented as ● in Figure 6.3) and *repeater* stations (*RPT*, represented as ○ in Figure 6.3). Each *IES* in the NoC is connected to a tile T_i. The *IES*$_k$ station (marked as $\times$ in Figure 6.3) inserts or extract *data* from the NoC to a tile located at the $(i, j)^{th}$ location in the mesh NoC. The *JS* station (marked as ● in Figure 6.3) transfers *data* from a *vertical* NoC segment to a *horizontal* NoC segment (or vice versa). The *RPT* station (marked as ○ in Figure 6.3) is a buffer that forwards a data on *vertical* rings. It ensures that horizontal and vertical data segments have the same length.

The rings in our NoC are unidirectional [8]. Therefore, without loss of generality, we assume in each ring that the $data$, the $address$ and the $valid$ bits flows in a counter-clockwise manner. The distance between two adjacent stations (both the vertical and horizontal directions of the *IES*) in the NoC is fixed, as shown in Figure 6.3.

6.3.4.1 The Ring Clock ($Rclk$)

Our ring-based NoC is based on the idea presented in [8]. We run our ring clock ($Rclk$) at a speed of up to 20GHz. The data flows in each ring of the NoC source-synchronously using a ring-based standing wave resonant oscillator (SWO) [8, 9], as shown in Figure 6.4. The ring clock is based on two parallel wires, that cross at the end by mobius crossing, to ensure that we get the same phase of the clock signal at any point in the ring. In [9], the authors use a single inverter to sustain the oscillation the ring, as shown in Figure 6.4. In our work, as mentioned in [8], we utilize an odd number of inverter pairs (3 in particular). We extract a full amplitude clock signal at any desired point in the ring using a clock recovery circuit, except for the *Dead-Zone* (where the

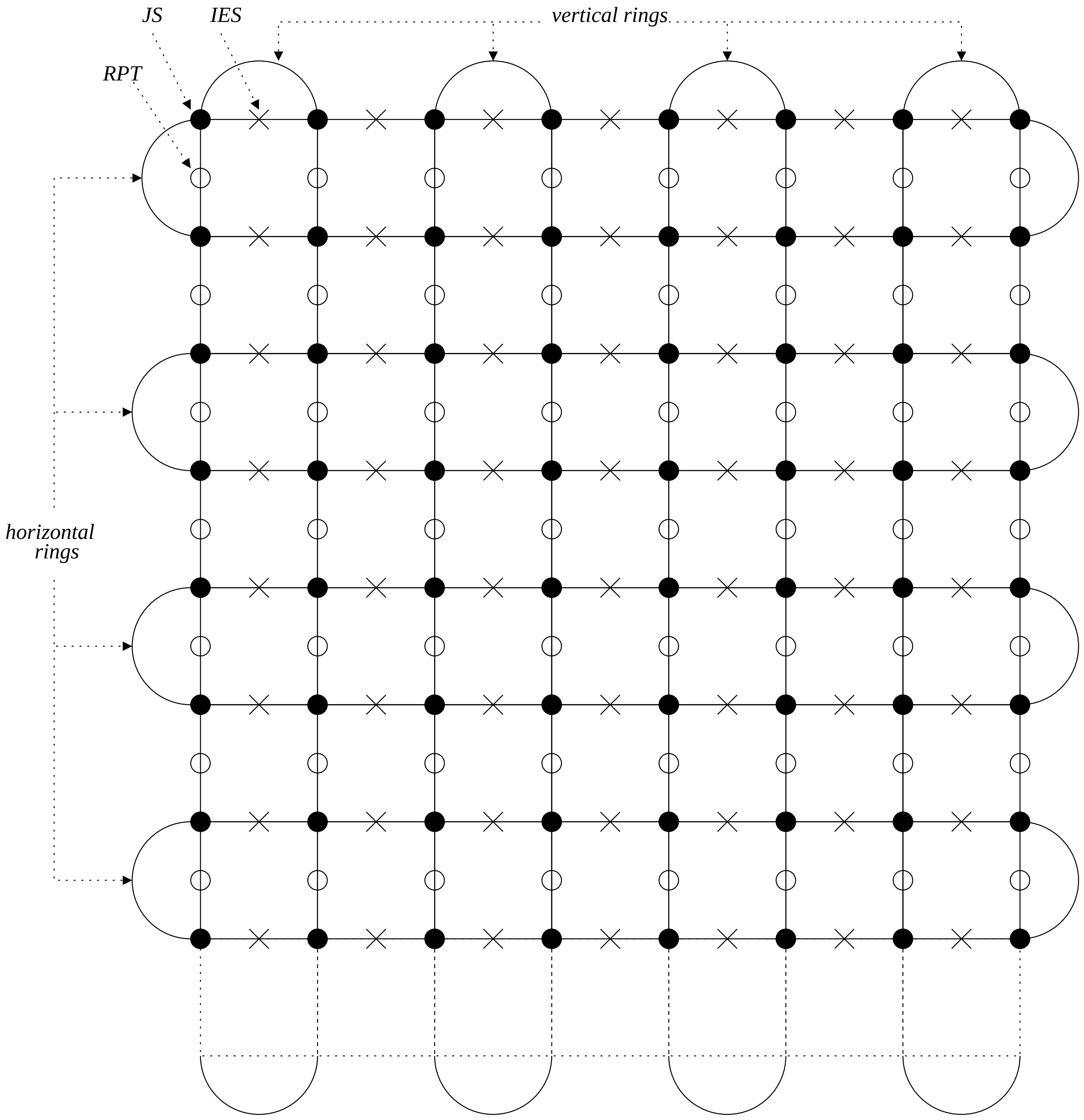


Figure 6.3: A 4×4 Ring-based NoC Architecture [8] in the PAU

amplitude of the ring signals is very small), as shown in Figure 6.4. The extracted full amplitude clock signal operates at a speed of up to 20GHz. We use the extracted clock signal to run our *IES*, *JS* and *RPT* stations in the NoC.

Next, we will discuss the *IES*, the *JS* and the *RPT* stations in the NoC, which were first de-

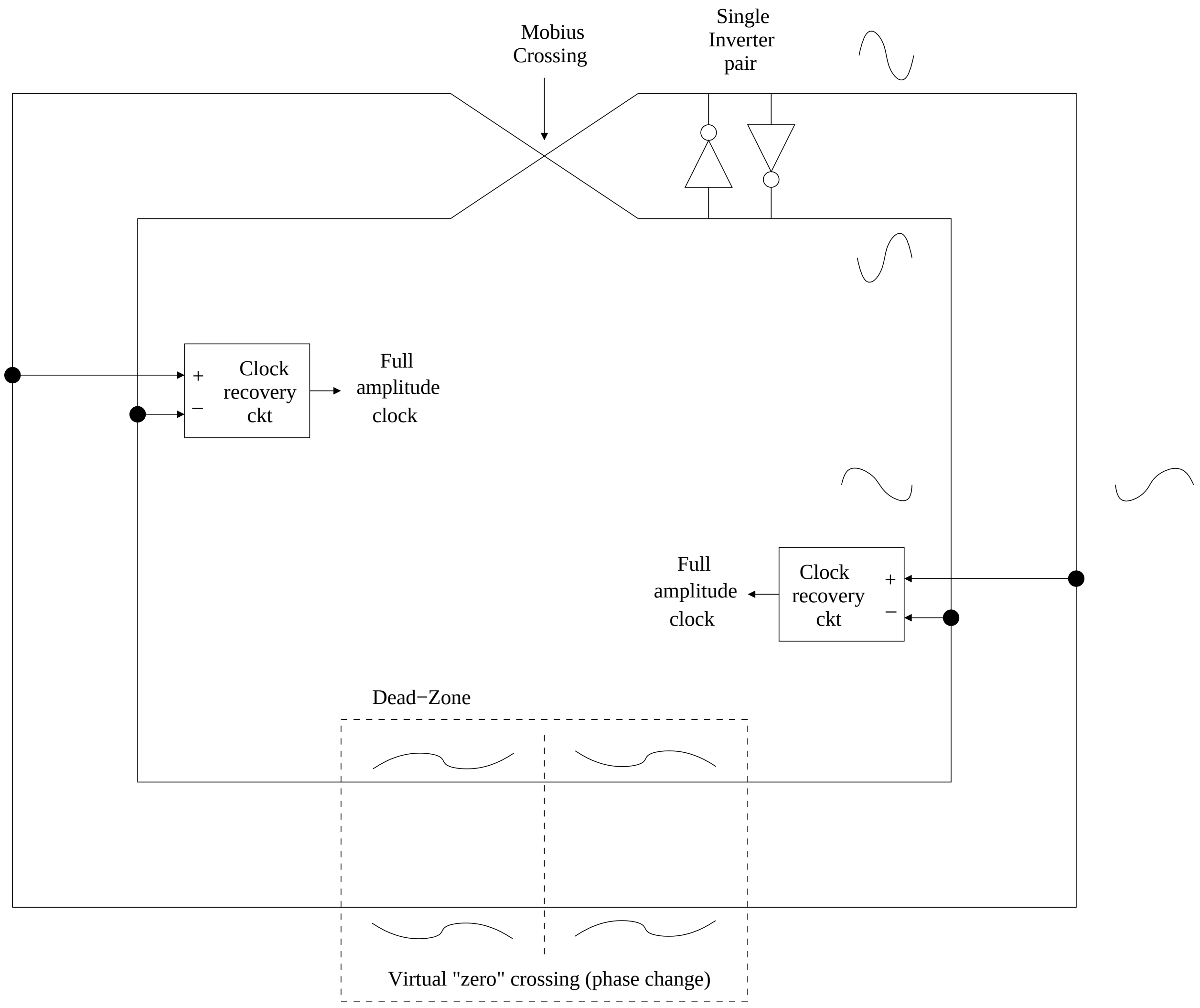


Figure 6.4: Single wave ring clock [8, 9]

scribed in [8].

6.3.4.2 The Insertion/Extraction Station (IES)

Each insertion/extraction station (*IES*) in the NoC in connected to some tile T_i. The *IES* (marked as $\times$ in Figure 6.3) extracts data from the NoC and delivers it to a tile T_i (if the data is valid and has the right address of the tile T_i). Also, the *IES* can insert data from a tile T_i into the NoC (when an empty slot is available). Figure 6.6 shows the circuit structure of the *IES* in the NoC in the PAU. The *address* (*r-bit*) signal consists the horizontal address of the *IES*, as well as its vertical address. Figure 6.5 shows the cross-section of the *address* wires modeled in the NoC.

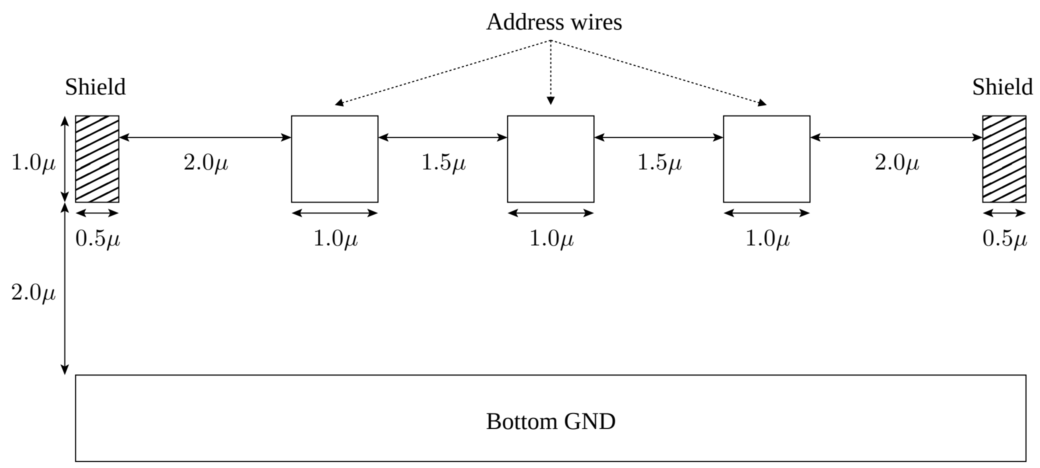


Figure 6.5: Address wires cross-section

In *IES$_i$*, there are three input signals coming from the mesh NoC (as shown in Figure 6.6): the *IEaddress$_{in}$*, the *data$_{in}$* and the *valid$_{in}$*. All these input signals are latched in latches that are operated by the $Ring$ clock ($Rclk$). *IES$_i$* compares the *IEaddress$_{in}$* with its own address *IEaddress$_i$*. If there is a match and the data is valid (*valid$_{in}$* is *high*), the *data$_{in}$* is inserted into the asynchronous fifo (*Din FIFO*), then the tile T_i will consume the data from the *Din FIFO*. Otherwise, the input signals will be forwarded to the output signals (the *IEaddress$_{out}$*, the *data$_{out}$* and the *valid$_{out}$*) on the $Ring$ NoC to the next station. In the other case, if a tile T_i has a data to be send to another tile T_j in the $Ring$ NoC, then the tile T_i writes its output data (*dataNext*) into the asynchronous fifo (*Dout FIFO*) and the next target tile T_j address (*addressNext*) into the asynchronous fifo (*Aout FIFO*), to be forwarded to the output signals of the *IES$_i$* station (the *IEaddress$_{out}$*, the *data$_{out}$* and the *valid$_{out}$*). The operation of the *IES* in the PAU is substantially the same as in [8].

6.3.4.3 The Junction Station (JS)

The *junction* station (*JS*, marked as ● in Figure 6.3) is the junction point between a *horizontal* ring and a *vertical* ring, that forwards the *address*, the *data* and the *valid* bits from a *horizontal* ring

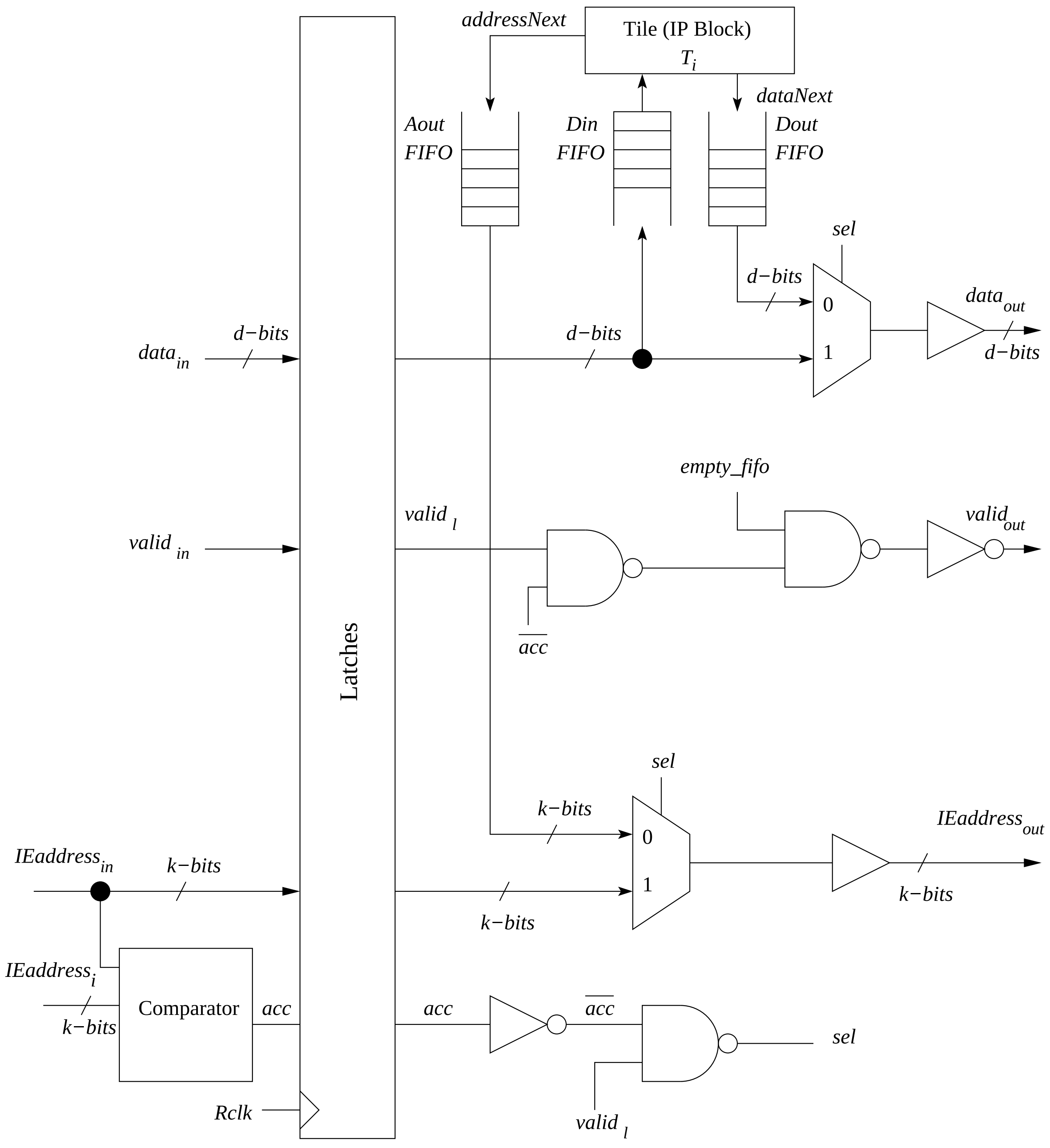


Figure 6.6: Insertion/Extraction Station (IES) in the NoC in the PAU

to a *vertical* ring (or vice versa) in the $Ring$ NoC. As shown in Figure 6.7, the *JS* compares the input vertical address (which is included in the packet header) with the current horizontal address. If there is a match ($Vsel_{ring}$ *is high*), the *JS* inserts the *data* bits (the *address* and the *valid* bits as well)

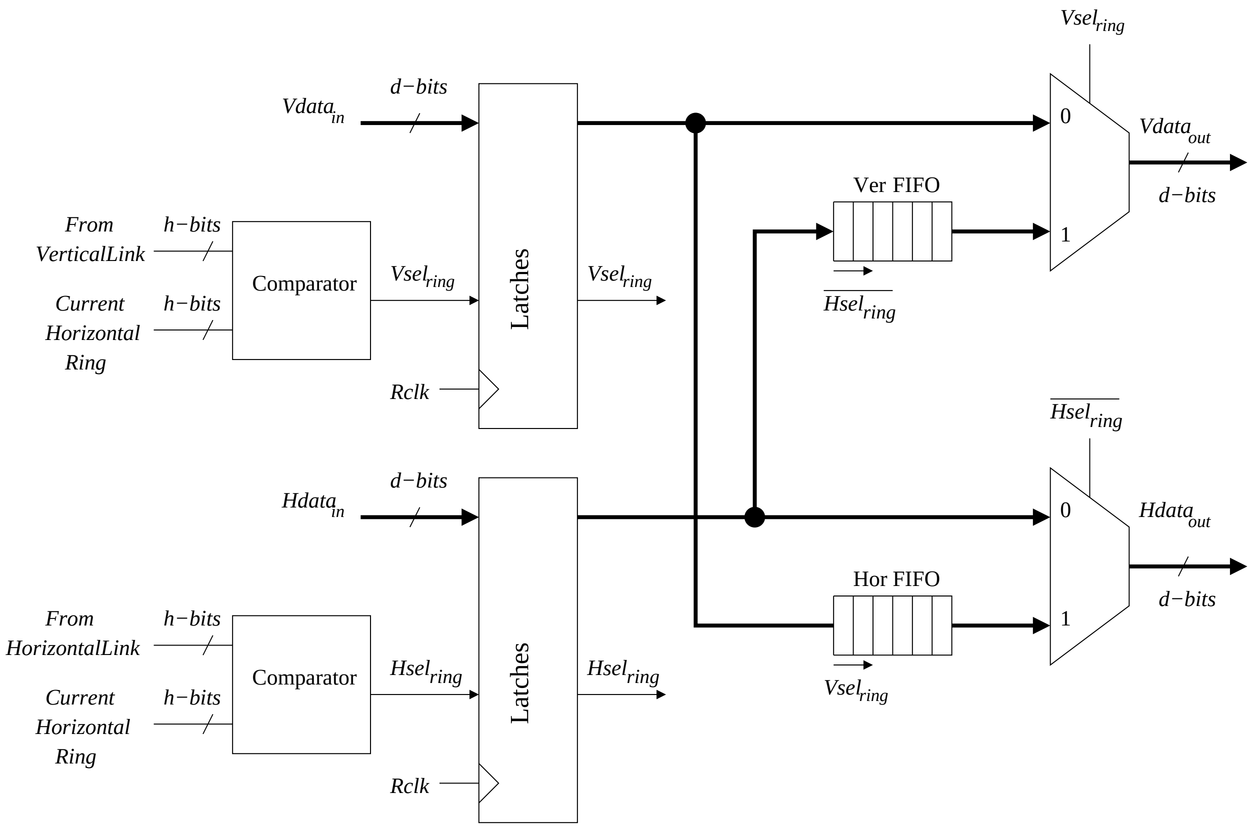


Figure 6.7: Junction Station (JS) in the NoC in the PAU

into the synchronous fifo (*Hor FIFO*), which the *JS* transfer the *data* to the current horizontal ring in the $Ring$ NoC. Otherwise, the data will stay in the current vertical ring (*$Vdata_{out}$*). In the other hand, the *JS* compares the input horizontal address with the current horizontal address. If there is not a match (*$Hsel_{ring}$ is low*), the *JS* inserts the *data* bits (the *address* and the *valid* bits as well) into the synchronous fifo (*Ver FIFO*), which the *JS* transfers the *data* bits to the current vertical ring in the $Ring$ NoC. Otherwise, the data will stay in the current horizontal ring (*$Hdata_{out}$*). The operation of the *JS* in the PAU is substantially the same as in [8].

6.3.4.4 The Repeater (RPT) Station

The *repeater* station (*RPT*, marked as ○ in Figure 6.3) forwards the data on the vertical rings in the $Ring$ NoC using buffers, as shown in Figure 6.8. The RPTs in the $Ring$ NoC are used to have a fixed distance between each adjacent stations in *vertical* rings.

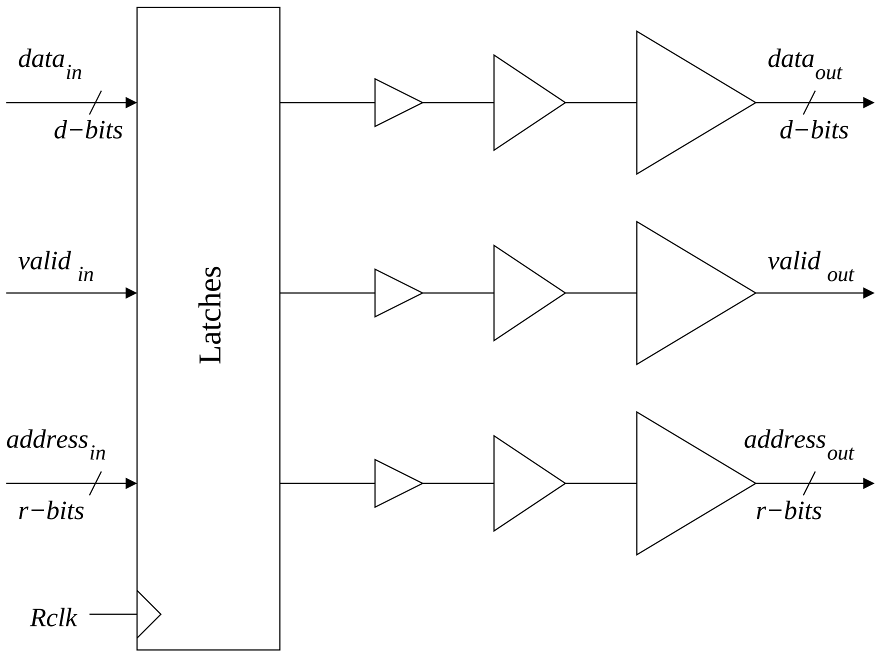


Figure 6.8: Repeater (RPT) Station in the NoC in the PAU

6.3.5 The PAU Design Flexibility

The control logic F_C in the PAU is realized by a reconfigurable FPGA like structure. This structure utilizes LUTs similar to an FPGA to allow programmability for different arithmetic applications in modern microprocessors. The reconfigurable feature of F_C provides the PAU the flexibility to embed different arithmetic application in modern microprocessors. For each arithmetic application, the F_C can be programmed to assign the required tiles, and the order of each tile in the arithmetic application computation order. In addition, the control logic F_C is utilized as a means for overall control of the PAU design.

6.3.6 Arithmetic Applications used in the PAU

We use three arithmetic application examples in the PAU, to compare it with a software-based implementation: the finite impulse response (FIR) filter [83], the discrete cosine transform (DCT) [73] and the Viterbi decoder [84]. The FIR filter (Figure 6.9) operates on discrete-time signals, whose impulse response is of finite duration [83]. In Figure 6.9, the input to the FIR filter

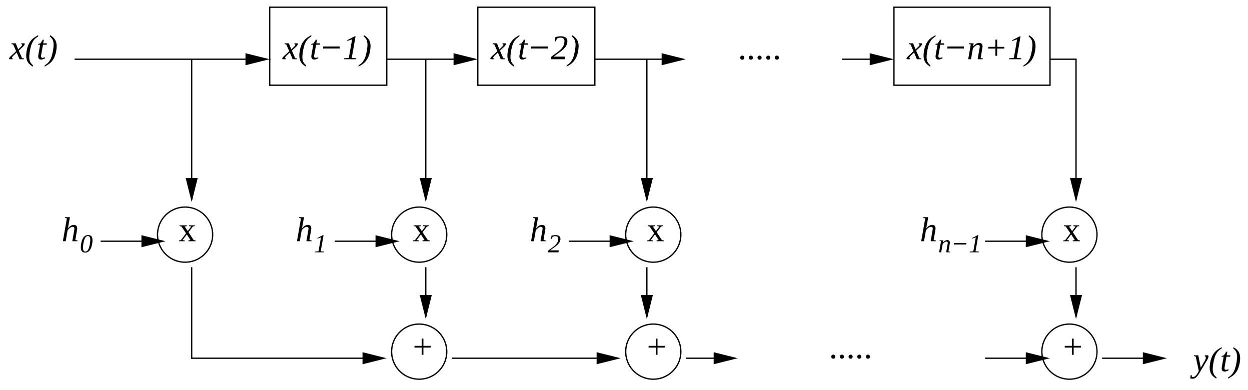


Figure 6.9: A discrete form of the FIR filter in the time domain.

at a discrete time is $x(t)$, and $y(t)$ is the output at a discrete time as well, where $[h_0 : h_{n-1}]$ are constants. The output $y(t)$ of the FIR filter can be calculated in the following formula:

$$y(t) = \sum_{0}^{n-1} h_i \times x(t-i) \tag{6.1}$$

The FIR filter arithmetic operations consists of multiple addition and multiplication operations.

The DCT is used in signal and image processing, and widely used in JPEG compression [73], which is lossy. For example, as shown in Figure 6.10, a DCT is applied on an image that is first converted into 16×16 macroblocks. Each macroblock consists of 16×16 pixels (each pixel is 8-bit). In each macroblock, an 8×8 pixels block is converted to a frequency domain using the DCT (8-point DCT in Figure 6.10). Each converted point ($D(i,j)$) in the frequency domain is calculated by the following formula [73] ($n = 8$ in Figure 6.10):

$$D(i,j) = \frac{1}{\sqrt{2n}} C(i)C(j) \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} cos(\frac{(2x+1)i\pi}{2n}) cos(\frac{(2y+1)j\pi}{2n}) \tag{6.2}$$

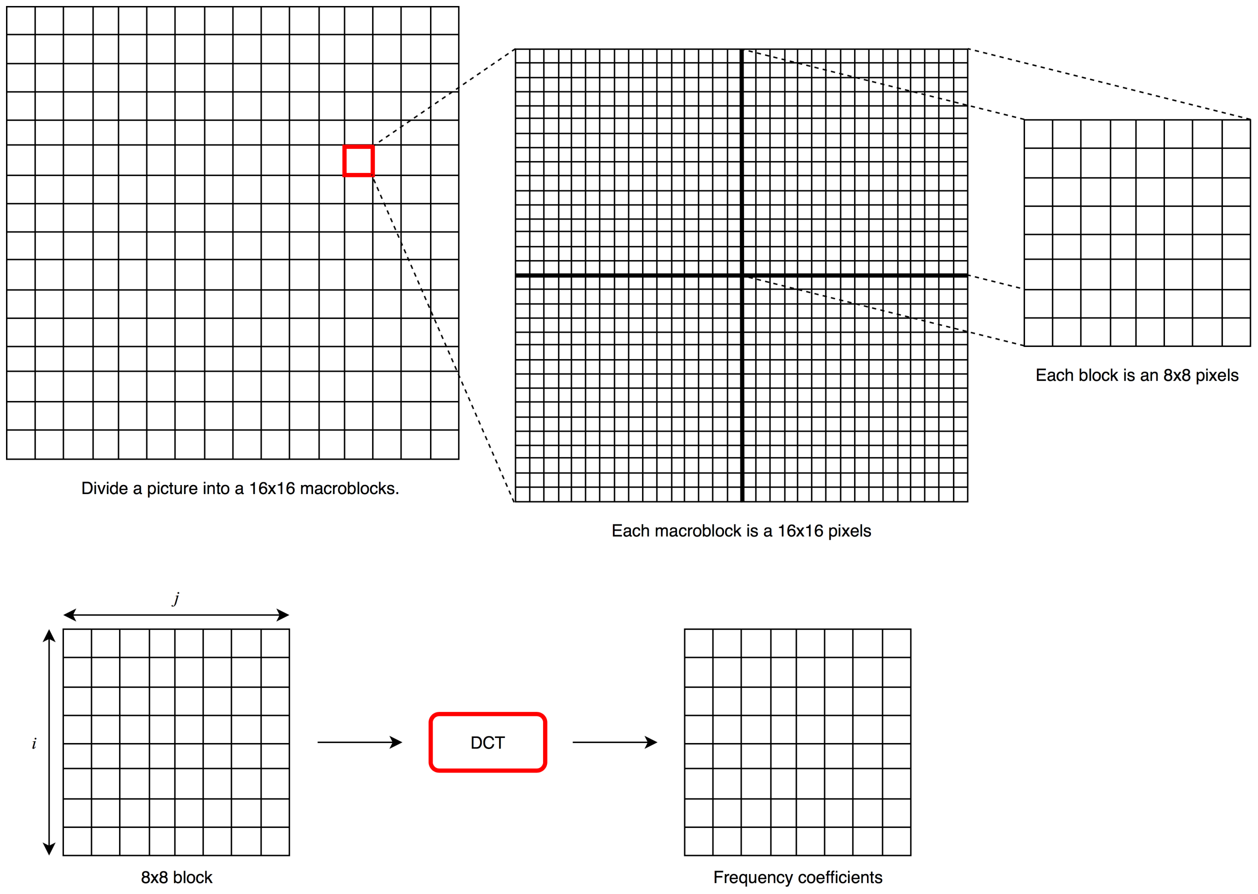

Divide a picture into a 16x16 macroblocks.
Each macroblock is a 16x16 pixels
Each block is an 8x8 pixels
j
i
8x8 block
DCT
Frequency coefficients

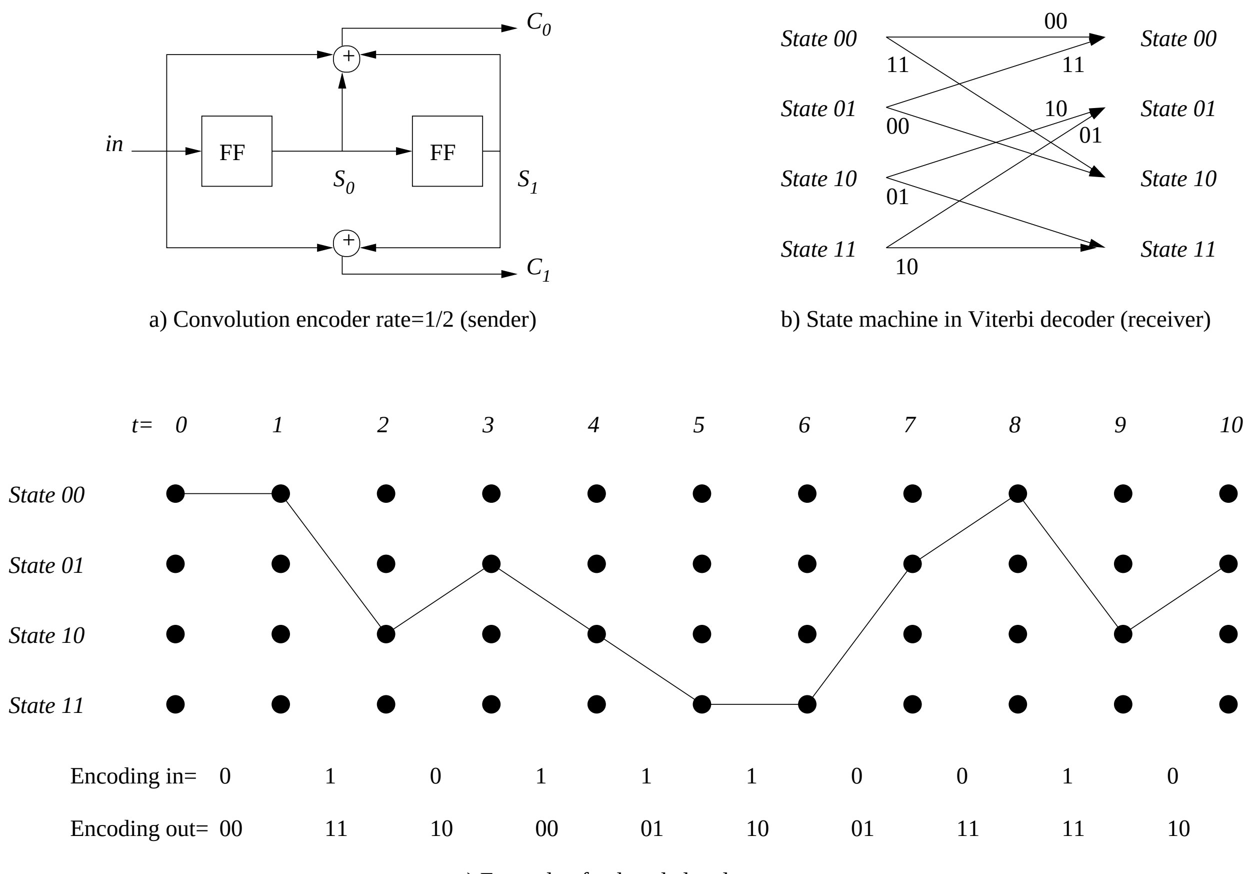
Figure 6.10: An example of an 8-point DCT used in JPEG.

where the scaling function $C(k)$ is:

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0 \\ 1 & \text{if } k > 0 \end{cases} \tag{6.3}$$

The DCT arithmetic operations consists of multiple addition, subtraction and multiplication operations. The cosine functions in the DCT can be pre-calculated and used as coefficients.

The Viterbi decoder used for decoding a bitstream that has been encoded using convolution code or trellis code [84]. The Viterbi decoder is widely used in wireless communication and hard drives (HDDs) error-correction codes (ECC). An example of the Viterbi decoder is shown in Fig-


in
FF
FF
S_0
S_1
C_0
C_1
+
+
State 00
State 01
State 10
State 11
00
11
11
00
10
01
01
10
t= 0 1 2 3 4 5 6 7 8 9 10
Encoding in= 0 1 0 1 1 1 0 0 1 0
Encoding out= 00 11 10 00 01 10 01 11 11 10

a) Convolution encoder rate=1/2 (sender)

b) State machine in Viterbi decoder (receiver)

c) Example of a decoded code

Figure 6.11: An example of a convolution encoder with a rate of $\frac{1}{2}$ and the Viterbi decoding trellis.

ure 6.11. In this example, the convolution encoder in the sender side of a two-way communication (Figure 6.11(a)) has an input (in) and two outputs (C_0 and C_1) and hence a rate (r) of $\frac{1}{2}$. The constrained length (l) is 3 in this example. The state machine of the Viterbi decoder for the receiver is shown in Figure 6.11(b). An encoded example by the Viterbi decoder is shown in Figure 6.11(c). The Viterbi decoder arithmetic operations consists of multiple addition, subtraction, multiplication and comparison operations.

In the next section, we will present the experimental results of the previous arithmetic application examples in the PAU.

6.4 Experimental Results

In this section, we present our experimental results of the PAU, using three arithmetic application examples: the FIR filter, the DCT and the Viterbi decoder. First, we discuss our PAU design flow in Section 6.4.1. Then, we discuss our simulation environment in Section 6.4.2. Finally, we present our simulation results along with a discussion in Section 6.4.3.

6.4.1 The PAU Design Flow

Each block in the PAU has a different design flow (as shown in Figure 6.12). The first is the tile design flow, the second is the FPGA controller (F_C) design flow and the third is the fast ring NoC data fabric design flow. In the tiles design flow, we describe the design process for each tile that will be connected to the PAU architecture, as determined by the arithmetic applications requirements. Then, we write the hardware description language (HDL) in Verilog for each tile T_i. We can have multiple instances of the same tile T_i in the PAU. As shown in the top part of Figure 6.12, we synthesize the Verilog code of each tile T_i using Synopsys Design Compiler (DC) [74] using cells from the the Nangate standard cell library [6] in the 45nm technology node. We extract the delay, the area and the power for each tile T_i. We next scale the delay, area and power from the 45nm to the 16nm technology node. To do this, we measure the delay, area and power of a 32-bit adder with random input vectors using Synopsys HSPICE [45] and PTM [46] high-performance process model card for the 16nm and 45nm technologies. After that, we calculate the scaling ratios of the delay, the area and the power, for the transitions from the 45nm to the 16nm technology nodes. Finally, we use these scaling ratios to extract the delay (D_{T_i}), the area (A_{T_i}) and the power (P_{T_i}) of each tile T_i in the 16nm technology node.

In the FPGA controller (F_C) design flow, the first step is a generation of the application in a language such as SystemC. Then, as shown in the middle part of Figure 6.12, we use Cadence Stratus HLS [85], high-level synthesis (HLS) tool to perform operation sequencing, scheduling and control logic generation for the PAU control logic. The HLS tool provides behavioral Verilog code of the application. After that, we use custom Perl [54] scripts to extract the control logic from

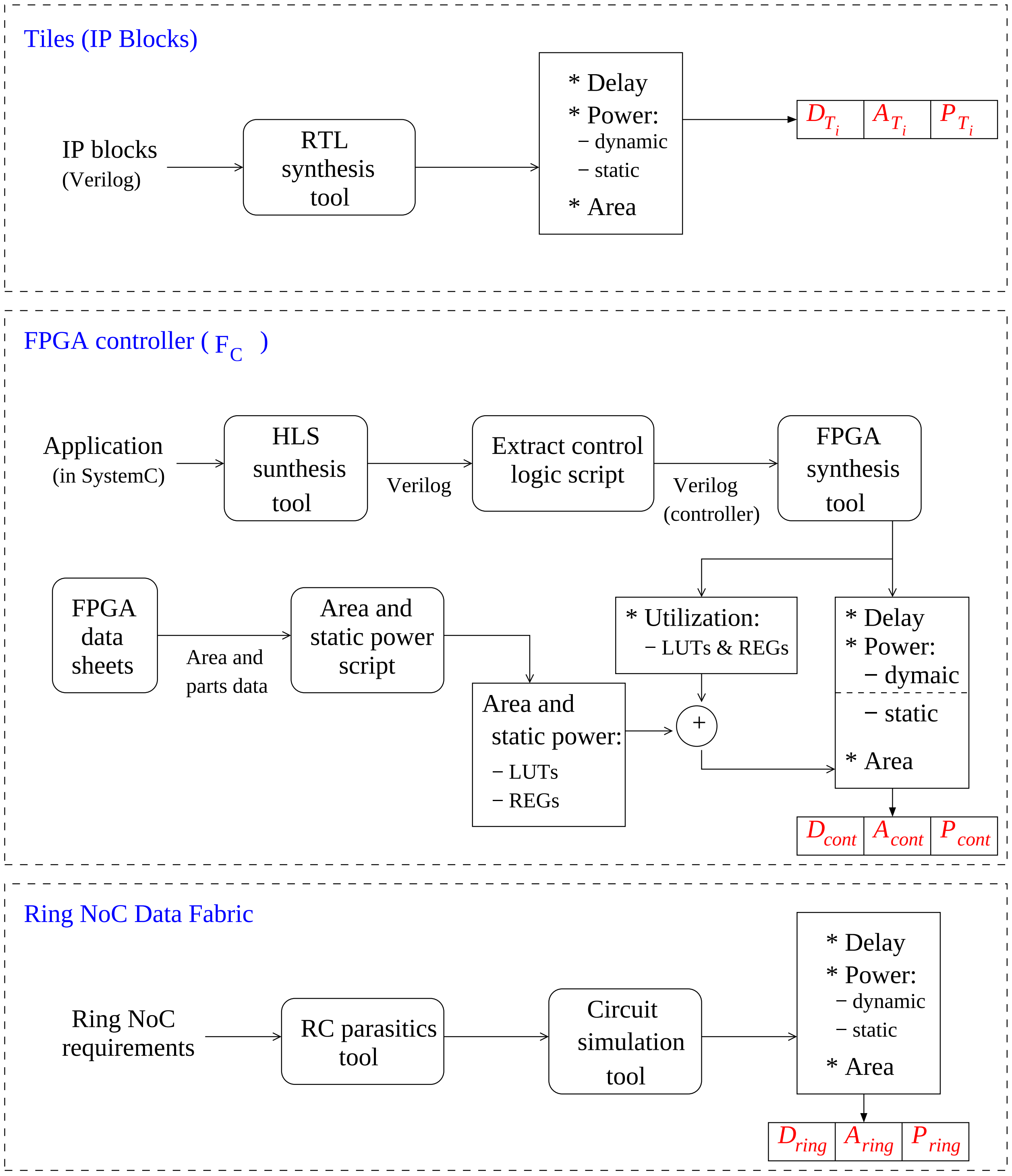


Figure 6.12: The PAU Design Flow

the Verilog code of the application (we refer to it as *controller* in Figure 6.12). We use the Xilinx Vivado [75], FPGA synthesis tool to generate a bitstream that configures the FPGA controller (F_C) to implement the control logic indicated by the HLS engine. From the FPGA synthesis tool, we extract the delay and the dynamic power of the F_C, and the utilization of the number of LUTs and the number of registers (REGs) in the F_C. Moreover, we the Xilinx Kintex UltraScale+ [86], FPGA device data sheet and a custom Python [87] script to extract the approximate area and static power of a single LUT. We obtain the area and the static power of a single register from a 45nm Nangate standard cell library [6], and scale them to a 16nm technology node. Finally, we extract the delay (D_{cont}), the area (A_{cont}) and the power (P_{cont}) of the F_C.

The bottom part of Figure 6.12 describes the fast NoC data fabric design flow. The first step is to identify the number of components (tiles and the F_C) that are going to utilize the NoC in the PAU. Then, we model the wires of the NoC using Synopsys Raphael [47], to extract the RC parasitics. After that, we utilize the RC parasitics in the circuit simulation of the NoC using Synopsys HSPICE [45], with a 16nm PTM [46] high-performance process model card. Finally, we extract the delay (D_{ring}), the area (A_{ring}) and the power (P_{ring}) of the $Ring$ NoC. For a data transfer from ring location (x_1, y_1) to (x_2, y_2), the ring is assumed to take $D_{ring} = D_{link} \times (|x_2 - x_1| + |y_2 - y_1|)$, where D_{link} is the delay to traverse one link in the NoC.

We calculate the final delay, area and power of the PAU as follows:

$$\begin{aligned} Power(PAU) &= P_{cont} + P_{ring} + \sum_{i=1}^{n} P_{T_i} \\ Area(PAU) &= A_{cont} + A_{ring} + \sum_{i=1}^{n} A_{T_i} \\ Delay(PAU) &= D_{cont} + D_{ring} + max(D_{T_1}, ..., D_{T_n}) \end{aligned} \tag{6.4}$$

Next, we will discuss the simulation environment of the arithmetic applications (the FIR filter, the DCT and the Viterbi decoder) used in the PAU.

6.4.2 Simulation Environment

We implement three arithmetic applications in the PAU (the FIR filter, the DCT and the Viterbi decoder) and compare the performance of the PAU with a software-based implementation written in the C language and compiled and run on an Intel i7 3.6GHz (with 32GB DDR3-1600MHz). We use the Intel [21] Performance Monitoring Unit (PMU) to measure the system performance. For the software-based implantation, we compute the number of cycles of each arithmetic application using the Intel x86 time stamp counter (TSC). We refer to this as SW_{cycles}. For the PAU implementation, we use the delay measured for each PAU arithmetic application in Section 6.4 in the performance comparison (we refer to it as PAU_{cycles}), to estimate the speedup using the PAU over the software-based version. The *speedup* of the PAU in each arithmetic application is calculated as follows:

$$Speedup = \frac{SW_{cycles}}{PAU_{cycles}} \tag{6.5}$$

We compare the *power* increase of the PAU to the average power of the Intel i7-5600 processor $(15W)$. We compare the *area* of the PAU with the area of a 32kB L1D cache in 16nm $(\sim 60740.6\mu^2)$. We estimate the area of the 32kB L1D cache in 16nm by scaling the area of the 32kB L1D cache in 45nm (Chapter 4, Table 4.3) to the 16nm technology node.

In the PAU, we vary the tiles configurations, by varying the number of cycles to produce a single result in each PAU arithmetic application. The number of cycles is varied from 1 to 16 cycles to generate different configurations of the PAU. Furthermore, we vary the configuration of each arithmetic application (the FIR filter, the DCT and the Viterbi decoder) used in the PAU. In the FIR filter, we vary the number of taps from 64 to 128, and the number of input bits from 8 to 64 bits. In the DCT, we present the 4-point and 8-point DCT, and we vary the number of parallel inputs from 1 to 4. In the Viterbi decoder, we vary the input/output rate (r) from $\frac{1}{2}$ to $\frac{9}{10}$, and the constraint length (l) from 4 to 6.

Next, we present our simulation performance results along with a discussion.

6.4.3 Results and Analysis

In this section, we present and analyze our experimental results. For the tiles (IP blocks) in the PAU, we refer to an adder as (*A*), a multiplier as (*M*), a subtractor as (*S*) and a comparator as (*C*). The experimental results of the arithmetic applications in the PAU will be in the following order: the FIR filter, the DCT and the Viterbi decoder.

6.4.3.1 The FIR filter in the PAU

The FIR filter has been implemented in a graphical processing unit (GPU) to speedup its arithmetic operations [88, 89]. The GPU achieves a maximum speedup of up to 4× compared to a software-based implementation [88], while the PAU demonstrates a speedup of up to 666× compared to a software-based implementation, as we will present in this Section. In addition, traditional GPUs have an overhead PCIe latency for the data transfer between the GPU and the DRAM. As a result, we can use the PAU to speedup the FIR filter arithmetic operations. We use the PAU to implement the FIR filter arithmetic application with a 100K inputs. We vary the number of taps in the FIR filter from 64 to 128, and the number of input bits from 8 to 64 bits. We vary the tiles configuration of the PAU from 4M-6A to 128M-127A. When we increase the number of tiles used in the PAU, the delay to produce a single result in the PAU decreases. The *speedup* of the FIR filter in the PAU is shown in Figure 6.13. We demonstrate a significant speedup of the FIR filter in the PAU of up to 666.8× (tiles of 128M-127A in the FIR filter of 128 taps and 32 bits), with an *area* increase of 115.8% (as shown in Figure 6.14) and a *power* increase of 0.68% (as shown in Figure 6.15). The minimum speedup achieved is 21.4× (tiles of 4M-6A in the FIR filter of 64 taps and 16 bits), with an area increase of 42.2% and a power increase of 1.43%. As shown in Figure 6.14, the area increase of the FIR filter in the PAU ranges from 13.8% to 545.3%. The power increase of the FIR filter in the PAU ranges from 0.24% to 7.56%, as shown in Figure 6.15. In Tables 6.1, 6.2 and 6.3, we show the area breakdown of different parts in the PAU for the largest area size of the 64, 96 and 128 taps of the FIR filter. The area of the F_C in the PAU increases as the number of tiles decreases, as expected, because the control logic is larger for a lower number

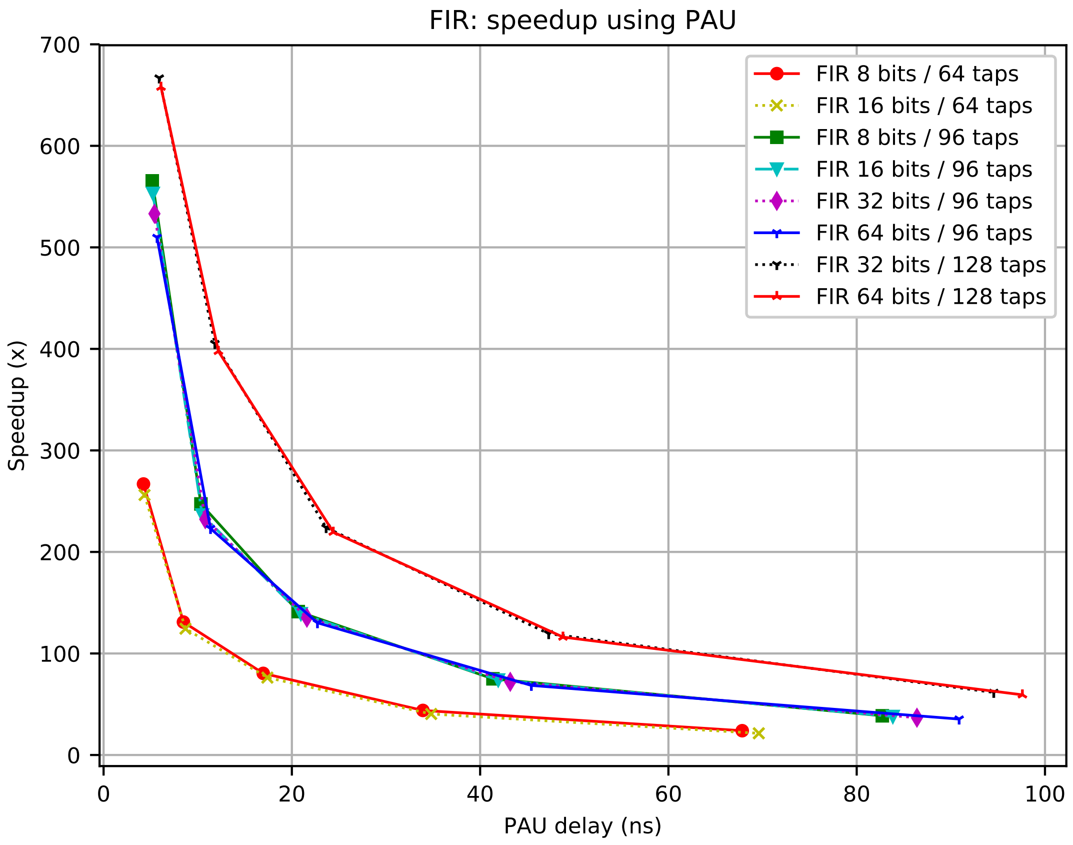


Figure 6.13: FIR filter – the PAU speedup compared to a software-based implementation.

of resources to perform the computation for the FIR filter arithmetic application.

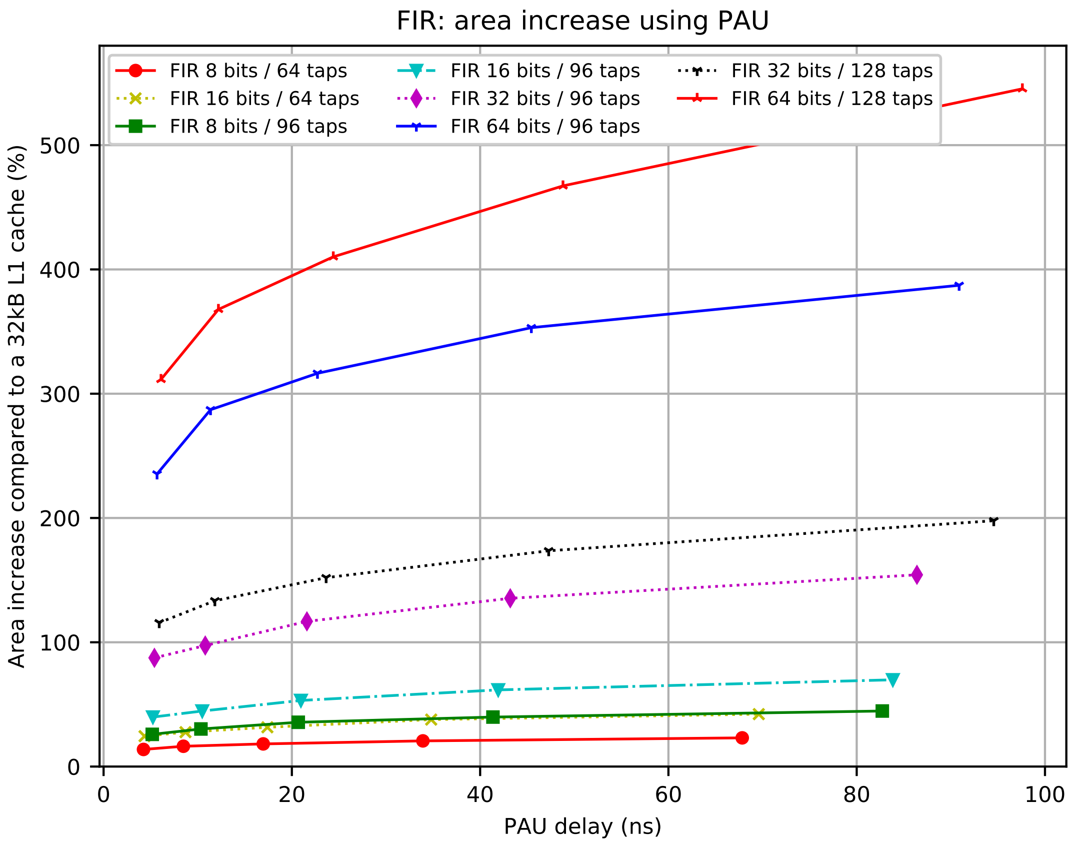


Figure 6.14: FIR filter – the PAU area increase compared to a 32kB L1D cache.

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
64M-63A	2789.1	18.79	10943.5	73.73	1109.4	7.47	14842.0
32M-33A	1936.7	11.52	5507.7	32.76	9365.7	55.71	16810.1
16M-18A	1239.3	6.46	2789.8	14.54	15159	79	19188.1
8M-11A	696.9	3.04	1442.8	6.28	20820.3	90.68	22960.0
4M-6A	309.6	1.21	733.4	2.86	24573.9	95.93	25616.9

Table 6.1: Area Breakdown of the PAU for the FIR filter of 64 Taps / 16 Bits (A: adders, M: multipliers)

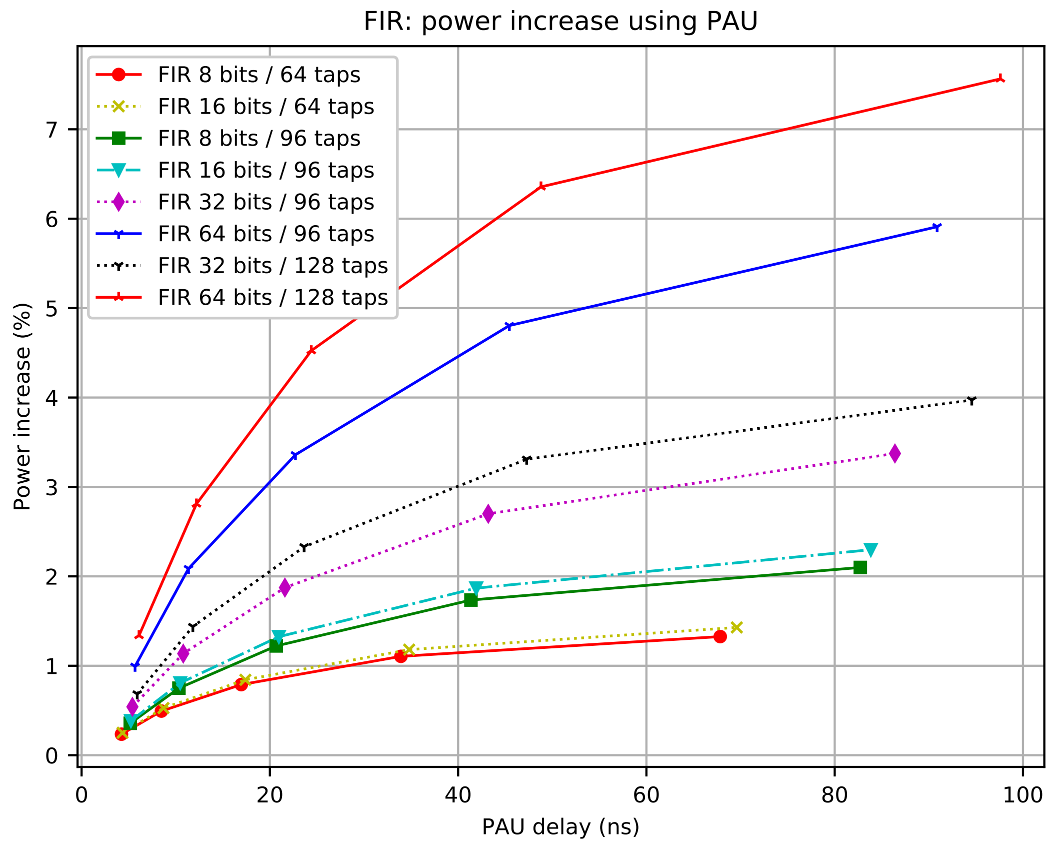


Figure 6.15: FIR filter –
the PAU power increase compared to Intel i7-5600U average power.

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
96M-95A	4958.9	3.47	132734.2	92.82	5305.5	3.71	142998.7
48M-50A	2789.1	1.60	66573.7	38.17	105036.0	60.23	174398.7
24M-28A	1239.3	0.64	33534.7	17.45	157424.7	81.91	192198.7
12M-13A	696.9	0.32	16684.7	7.78	197116.4	91.90	214498.1
6M-8A	696.9	0.30	8466.3	3.60	226034.8	96.10	235198.1

Table 6.2: Area Breakdown of the PAU for the FIR filter of 96 Taps / 64 Bits (A: adders, M: multipliers)

Tiles configuration	$Ring$		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
128M-127A	6477.3	3.42	177006.5	93.56	5714.7	3.02	189198.5
64M-66A	2789.1	1.25	88709.8	39.7	131933.3	59.05	223432.2
32M-34A	1936.7	0.78	44437.5	17.84	202747.2	81.38	249121.4
16M-17A	1239.3	0.44	22218.8	7.83	260339.7	91.73	283797.8
8M-11A	696.9	0.21	11315.9	3.42	319228.7	96.37	331241.5

Table 6.3: Area Breakdown of the PAU for the FIR filter of 128 Taps / 64 Bits (A: adders, M: multipliers)

6.4.3.2 The DCT in the PAU

The DCT has been implemented in a GPU to speedup its arithmetic operations [72, 90]. The GPU achieves a maximum speedup of up to 21× compared to a software-based implementation [72], while the PAU demonstrates a speedup of up to 696× compared to a software-based implementation, as we will present in this Section. As a result, we can use the PAU to speedup the DCT arithmetic operations. We use the PAU to implement the DCT arithmetic application in a 32×32 pixels input. We implement the 4-point and the 8-point DCT, and we vary the number of parallel inputs in the DCT. We vary the tiles configuration of the PAU from 1M-2A-1S to 24M-50A-14S. When we increase the number of tiles used in the PAU, the delay to produce a single result in the PAU decreases. The *speedup* of DCT in the PAU is shown in Figure 6.16. We demonstrate a significant speedup of the DCT in the PAU of up to 696.5× (tiles of 24M-50A-14S in the 8-point DCT of 4 parallel input), with an *area* increase of 50.1% (as shown in Figure 6.17) and a *power* increase of 0.59% (as shown in Figure 6.18). The minimum speedup achieved is 36.1× (tiles of 2M-4A-2S in the 4-point DCT of 1 parallel input), with an area increase of 66.2% and a power increase of 2.86%. As shown in Figure 6.17, the area increase of the DCT in the PAU ranges from 10.8% to 95.6%. The power increase of the DCT in the PAU ranges from 0.18% to 3.84%, as shown in Figure 6.18. In Tables 6.4 and 6.5, we show the area breakdown of different parts in

Tiles configuration	$Ring$		Tiles		F_C		Total (μ^2)
	μ^2	%	μ^2	%	μ^2	%	
24M-50A-14S	1936.7	6.37	9233.06	30.35	19252.3	63.28	30422.1
14M-26A-8S	1239.3	3.26	3744.34	9.86	32982.4	86.87	37966.0
7M-14A-6S	696.9	1.54	1908.15	4.22	42658.9	94.24	45264.0
4M-8A-4S	309.6	0.6	954.08	1.84	50478.3	97.56	51742.0
2M-4A-2S	309.6	0.53	477.04	0.82	57269.4	98.65	58056.0

Table 6.4: Area Breakdown of the PAU for the 8-point DCT with 4 parallel inputs (A: adders, M: multipliers, S: subtractors)

the PAU for the largest area size of the 4-point and 8-point DCT.

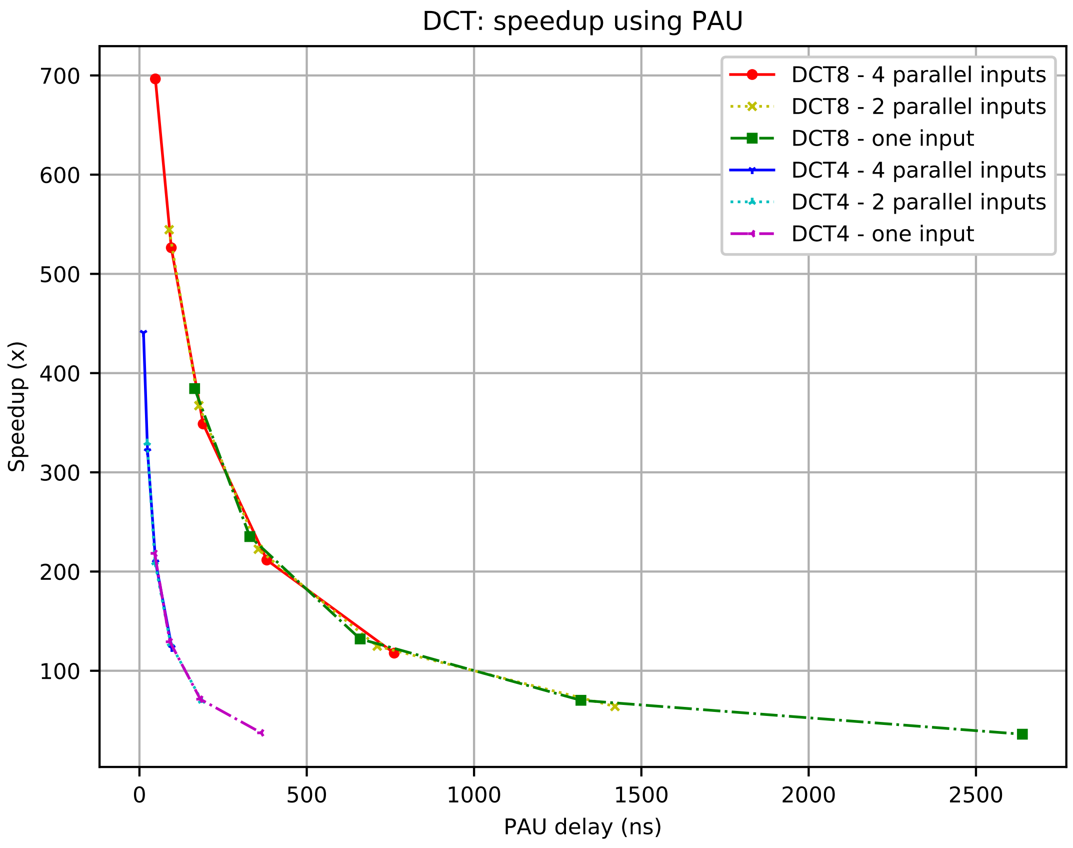


Figure 6.16: DCT – the PAU speedup compared to a software-based implementation.

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
6M-12A-6S	696.9	7.08	2388.9	24.26	6762.2	68.67	9848.0
4M-6A-4S	696.9	6.2	1520.6	13.54	9016.5	80.26	11234.0
2M-3A-2S	309.6	2.29	760.3	5.61	12472.1	92.1	13542.0
1M-2A-1S	309.6	2	398.1	2.58	14753.3	95.42	15461.0

Table 6.5: Area Breakdown of the PAU for the 4-point DCT with 4 parallel inputs (A: adders, M: multipliers, S: subtractors)

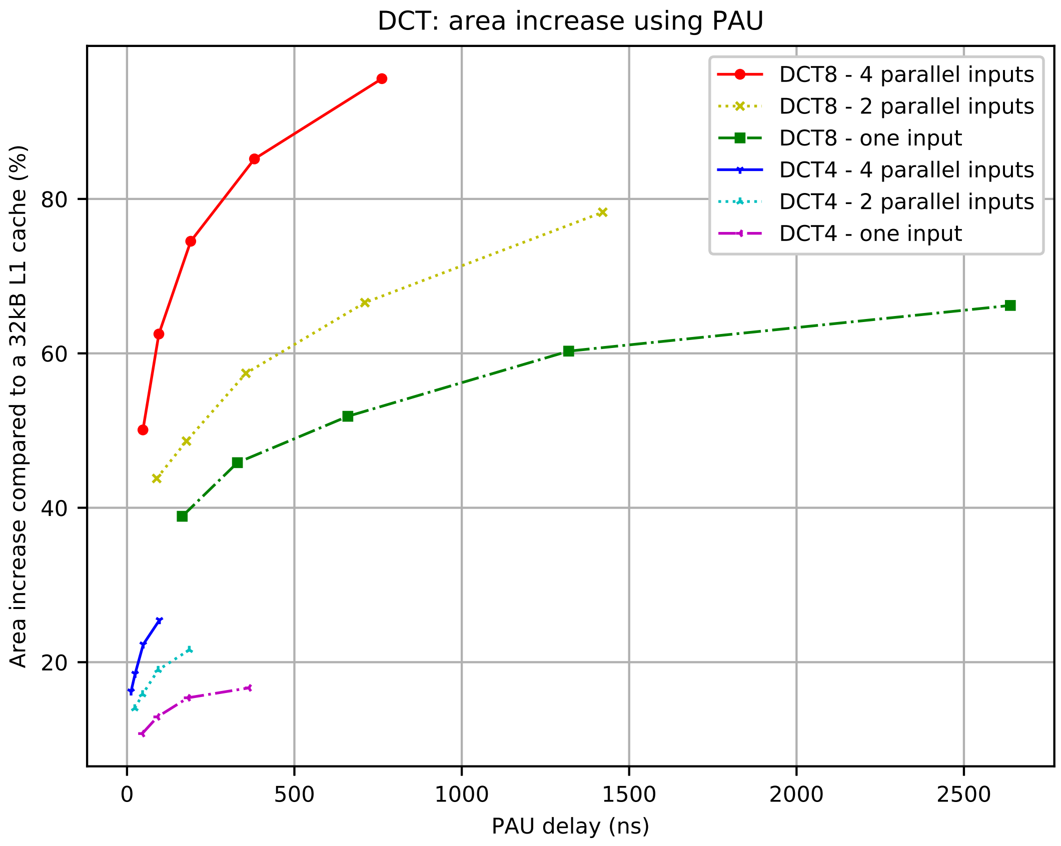


Figure 6.17: DCT – the PAU area increase compared to a 32kB L1D cache.

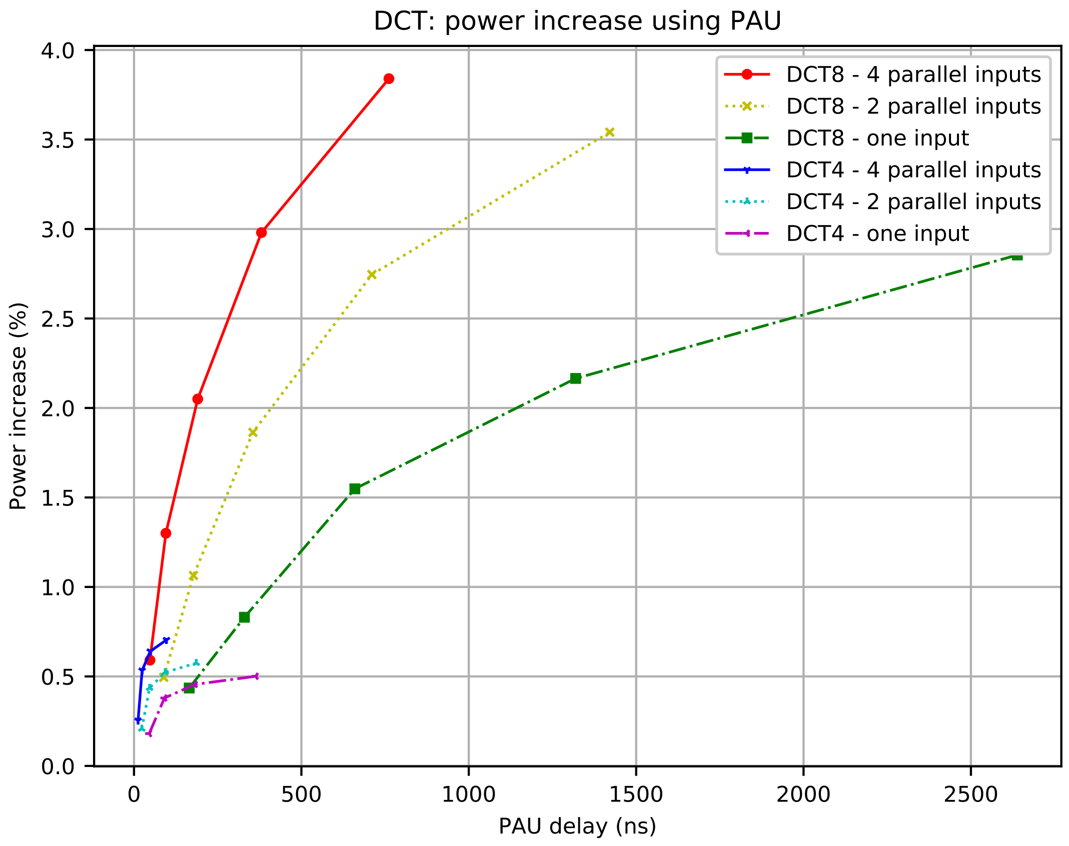


Figure 6.18: DCT – the PAU power increase compared to Intel i7-5600U average power.

6.4.3.3 *The Viterbi Decoder in the PAU*

The Viterbi decoder has been implemented in a GPU to speedup its arithmetic operations [91, 92]. The GPU achieves a maximum speedup of up to 145× compared to a software-based implementation [91], while the PAU demonstrates a speedup of up to 832.3× compared to a software-based implementation, as we will present in this Section. As a result, we can use the PAU to speedup the Viterbi decoder arithmetic operations. We use the PAU to implement the Viterbi decoder arithmetic application to decode a 1152 bytes message. We vary the number of input/output rate (r) in the Viterbi decoder from $\frac{1}{2}$ to $\frac{9}{10}$, and the constrained length (l) from 4 to 6. We vary the tiles configuration of the PAU from 1M-2A-3S-1C to 38M-55A-83S-29C. When we increase the number of tiles used in the PAU, the delay to produce a single result in the PAU decreases. The *speedup* of the Viterbi decoder in the PAU is shown in Figure 6.19. We observe a significant speedup of the Viterbi decoder in the PAU of up to 832.3× (tiles of 28M-45A-71S-24C in the Viterbi decoder of $r = \frac{7}{8}$ and $l = 6$), with an *area* increase of 54.8% (as shown in Figure 6.20) and a *power* increase of 0.99% (as shown in Figure 6.21). The minimum speedup achieved is 3.3× (tiles of 1M-2A-3S-1C in the Viterbi decoder of $r = \frac{1}{2}$ and $l = 4$), with an area increase of 91% and a power increase of 2.25%. As shown in Figure 6.20, the area increase of the Viterbi decoder in the PAU ranges from 47.4% to 104.1%. The power increase of the Viterbi decoder in the PAU ranges from 0.69% to 3.21%, as shown in Figure 6.21. In Tables 6.6, 6.7, 6.8 and 6.9, we show

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
28M-45A-71S-24C	3796.5	11.51	4267.5	12.93	24933.4	75.56	32997.4
14M-24A-36S-12C	1239.3	3.22	1293.4	3.36	35945.5	93.42	38478.2
8M-12A-18S-6C	696.9	1.52	666.0	1.45	44508.5	97.03	45871.4
4M-6A-10S-3C	696.9	1.39	333.0	0.66	49126.3	97.95	50156.2
2M-3A-5S-2C	309.6	0.52	166.5	0.28	58995.5	99.20	59471.6

Table 6.6: Area Breakdown of the PAU for the Viterbi decoder of $r = \frac{1}{2}$ and $l = 6$ (A: adders, M: multipliers, S: subtractors, C: comparators)

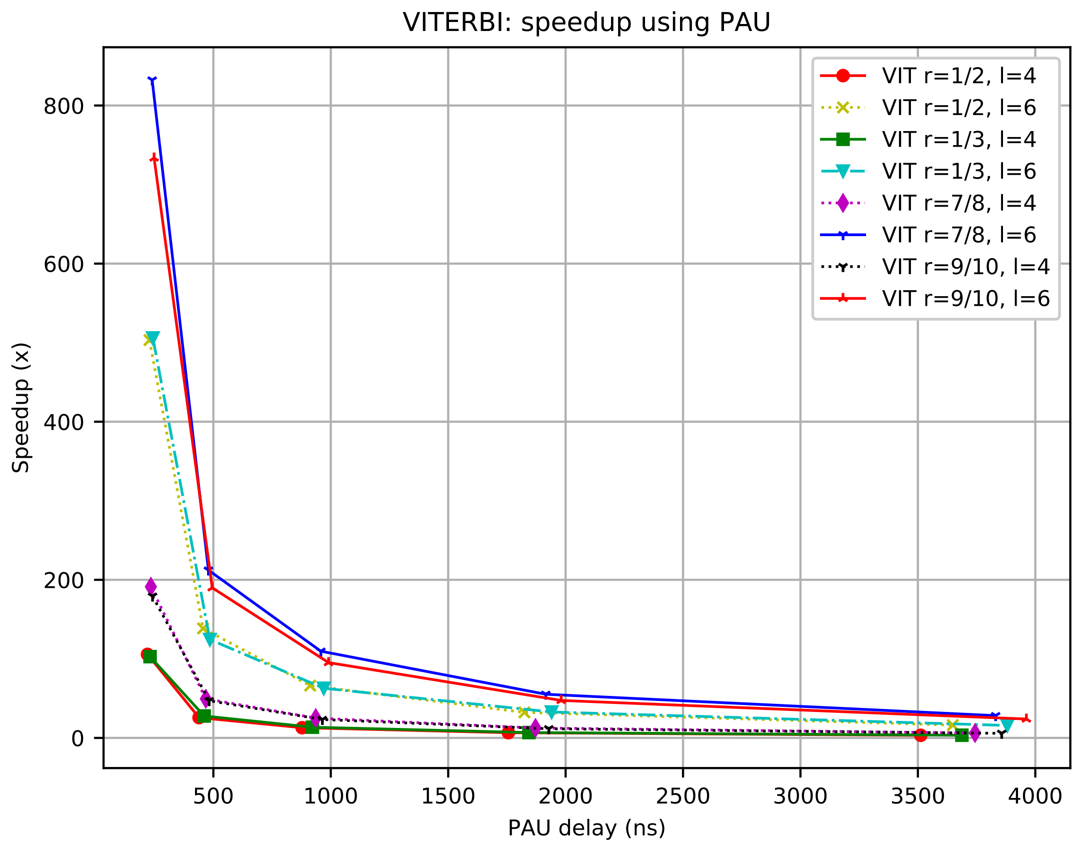


Figure 6.19: Viterbi decoder – the PAU speedup compared to a software-based implementation.

the area breakdown of different parts in the PAU for the largest area size of the $r = \frac{1}{2}$, $r = \frac{1}{3}$, $r = \frac{7}{8}$ and $r = \frac{9}{10}$ in the Viterbi decoder.

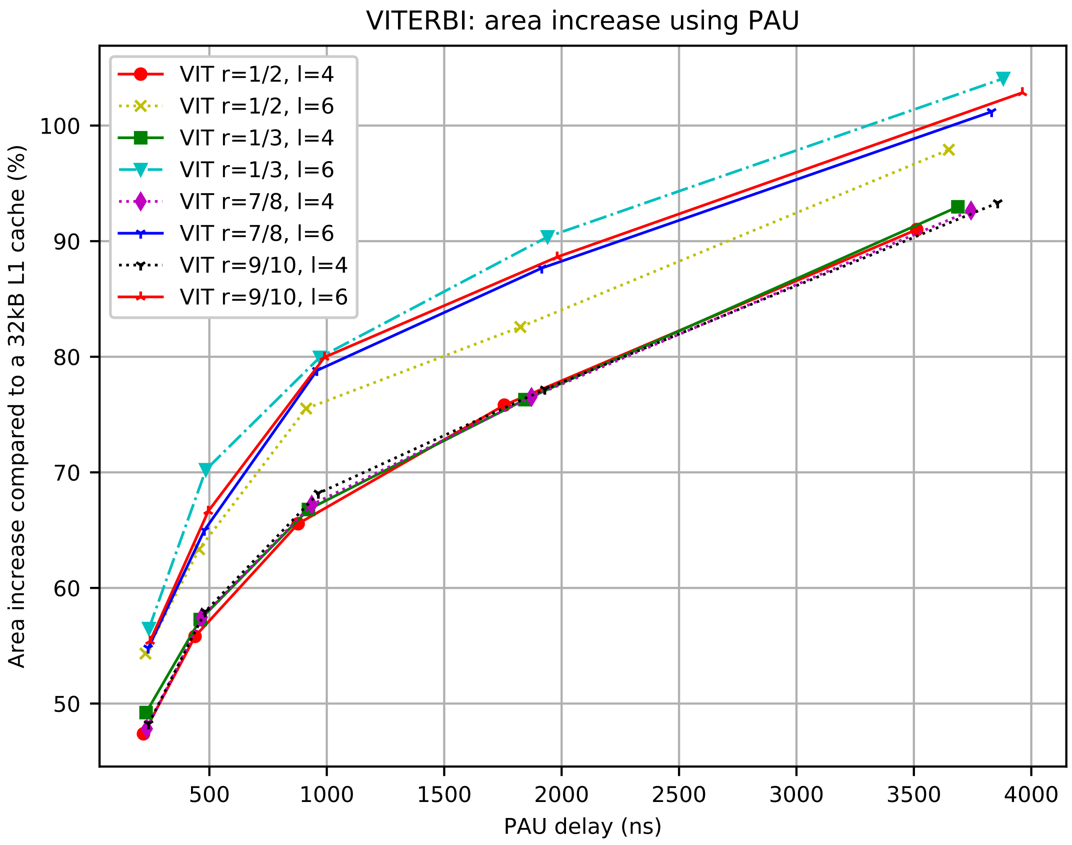


Figure 6.20: Viterbi decoder – the PAU area increase compared to a 32kB L1D cache.

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
38M-55A-83S-29C	4958.9	14.45	5470.9	15.95	23879.1	69.6	34308.9
20M-28A-42S-16C	2789.1	6.54	2845.6	6.67	37022.4	86.79	42657.1
10M-14A-21S-8C	1239.3	2.55	1422.8	2.93	45898.9	94.52	48561.0
6M-8A-12S-4C	696.9	1.27	832.3	1.52	53364.8	97.21	54894.0
3M-4A-6S-2C	696.9	1.1	416.1	0.66	62100.9	98.24	63213.9

Table 6.7: Area Breakdown of the PAU for the Viterbi decoder of $r = \frac{1}{3}$ and $l = 6$ (A: adders, M: multipliers, S: subtractors, C: comparators)

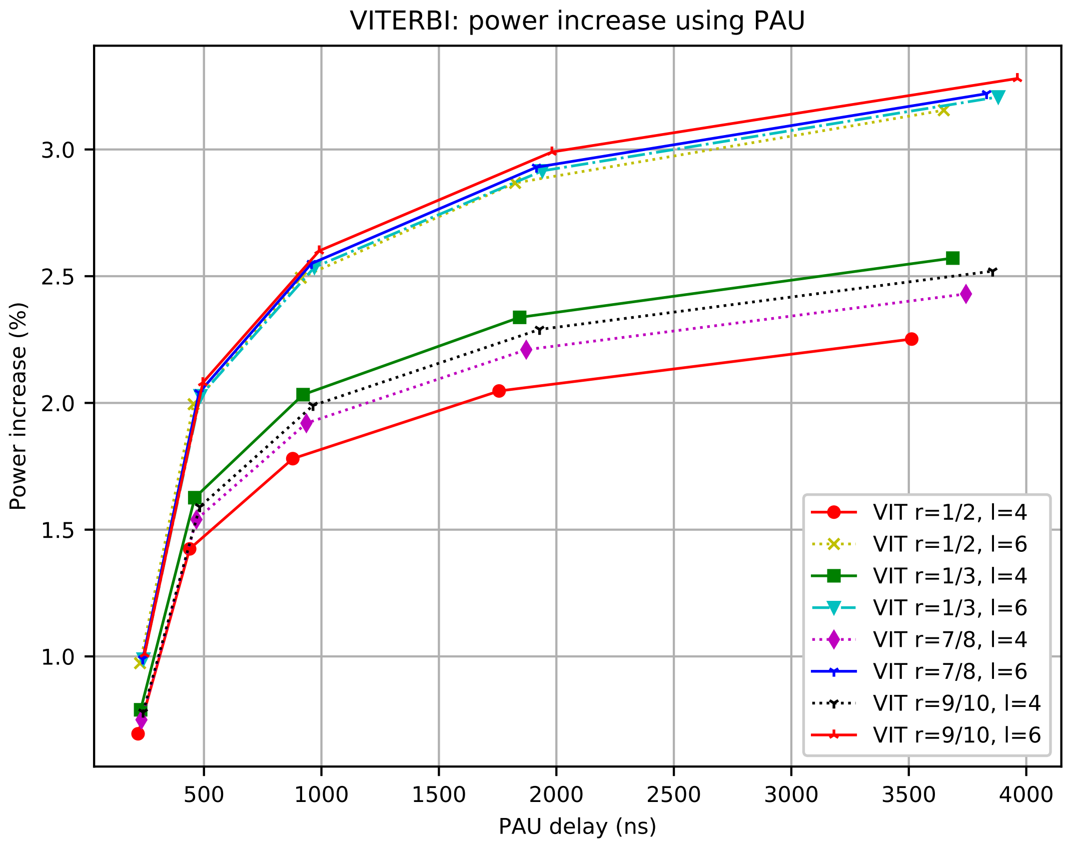


Figure 6.21: Viterbi decoder – the PAU power increase compared to Intel i7-5600U average power.

Tiles configuration	*Ring*		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
28M-45A-71S-24C	3796.5	11.41	4267.5	12.82	25218.4	75.77	33282.4
14M-24A-36S-12C	1239.3	3.14	1293.4	3.28	36945.4	93.58	39478.1
8M-12A-18S-6C	696.9	1.46	666.0	1.39	46508.3	97.15	47871.2
4M-6A-10S-3C	696.9	1.31	333.0	0.63	52221.9	98.07	53251.8
2M-3A-5S-2C	309.6	0.50	166.5	0.27	60995.2	99.23	61471.3

Table 6.8: Area Breakdown of the PAU for the Viterbi decoder of $r = \frac{7}{8}$ and $l = 6$ (A: adders, M: multipliers, S: subtractors, C: comparators)

Tiles configuration	$Ring$		Tiles		F_C		Total
	μ^2	%	μ^2	%	μ^2	%	(μ^2)
28M-45A-71S-24C	3796.5	11.28	4267.5	12.68	25584.1	76.03	33648.1
14M-24A-36S-12C	1239.3	3.06	1293.4	3.20	37945.4	93.74	40478.1
8M-12A-18S-6C	696.9	1.43	666.0	1.37	47208.3	97.19	48571.2
4M-6A-10S-3C	696.9	1.29	333.0	0.62	52807.0	98.09	53836.9
2M-3A-5S-2C	309.6	0.50	166.5	0.27	61995.2	99.24	62471.3

Table 6.9: Area Breakdown of the PAU for the Viterbi decoder of $r = \frac{9}{10}$ and $l = 6$ (A: adders, M: multipliers, S: subtractors, C: comparators)

6.5 Chapter Summary

In this chapter, we proposed a programmable arithmetic unit (PAU) for use in modern microprocessors, as a new SFU. The PAU can be programmed for a specific arithmetic algorithm and used for different arithmetic applications. The tiles (IP blocks) and the FPGA controller in the PAU communicate via a high speed ring NoC data fabric. We designed three arithmetic applications (the FIR filter, the DCT and the Viterbi decoder) in the PAU, and compared their speedup with a software-based implementation. Also, we compared their area with a 32kB L1D cache, and their power with the Intel i7-5600U average power. The PAU achieves a significant speedup of up to 832.3$\times$ with a minimal power increase and an acceptable area increase.

7. FUTURE WORK

The work presented in this thesis is the first work to discuss the use of a hardware hash unit and a programmable arithmetic unit for use in modern microprocessors. In this chapter, we discuss some ideas that can be implemented in the future to further improve the hardware hash unit and the programmable arithmetic unit design approaches presented in this thesis.

7.1 Coherent Hash Tables

In Chapter 3, we have presented a hardware hash unit that has CAM-based bins in the hash table. The hash table has been studied in a single core setting. Multi-core systems can have a significant gains using hash tables (with CAM-based bins). Multiple hash tables can increase the performance of hash applications and non-hash applications as well significantly, as shown in Chapter 3 for a single core system. Therefore, we need to study coherent hash tables to be used in multi-core microprocessors.

7.2 Flash-based CAM Cells in Hash Tables

In Chapter 4, we have presented our hardware hash unit at the circuit level, using the 9-T CAM CMOS-based cell [20], to implement our hash table. An area efficient hash table can be implemented using floating-gate (flash) transistors to construct flash-based CAM cells [93]. The flash-based CAM design can have a lower area of up to 8$\times$ and a lower power of up to 1.64$\times$ compared to the CMOS-based CAM design [93]. The delay of flash-based CAM design is 2.5$\times$ more than the CMOS-based CAM design, but the link speed of the flash-based CAM design is about 4$\times$ faster than the CMOS-based CAM design routers [93]. As a result, we can have a more dense hash tables using flash-based CAM cells with a better link speed and a lower power consumption compared to the CMOS-based CAM cells.

7.3 The HU and the PAU using Coherent Memory CPU-FPGA System

Recently, most leading companies in the technology industry [21, 94, 95] have been focusing in the heterogeneous CPU-FPGA coherent memory systems. The CPU-FPGA memory coherent systems can increase the performance of most of today's applications such as cloud computing. Also, they provide a better hardware flexibility and reduce the hardware implementation costs. Therefore, we can implement our HU and PAU in the CPU-FPGA memory coherent systems, to speedup hashing and arithmetic applications.

8. THESIS SUMMARY AND CONCLUSIONS

Today's applications demand high performance computation for applications such as cloud computing, web-based search engines, network applications, signal and medial processing and social media tasks. These software applications perform an extensive use of hashing and arithmetic operations in their computation. In this thesis, we presented a hardware hash unit (HU) and a programmable arithmetic unit (PAU) for use in modern microprocessors as new Special Function Units (SFUs). We proposed the HU at the microarchitecture level, the circuit level and an FPGA-based coprocessor for virus checking applications. Also, we proposed the PAU at the microarchitecture level and the circuit level.

For the HU at the microarchitecture level and the circuit level, we embedded the HU in the modern microprocessor's execution pipeline. Each bin in the hash table of the HU is stored in a CAM structure. We design our HU to be shared by multiple hash applications, and still demonstrate significant speedup. The HU reduces the L1D cache misses for the overall system. We have implemented and verified the operation of the HU at the microarchitecture level and the circuit level. We showed that the HU achieves a speedup of up to 15$\times$ over the software-based hash implementation with a reduced cache miss rate. We demonstrated an average power improvement of 5.48$\times$ for our HU design compared to a traditional CAM design. We varied the HT sizes and reported their area, delay, and power. Our HU can be run at 1.39 GHz after guard-banding for PVT variations. Also, we have implemented an FPGA-based coprocessor for virus checking applications, based on the idea of our HU. We implement each bin of the hash table (HT) as a CAM, using the BRAMs in the FPGA. We have verified the correctness of all hash operations in the real-time system design. We achieved a speedup of up to 5.37$\times$ for the FPGA-based HU implementation over the software-based hashing implementation, in the context of a virus checking application. We can achieve a higher speedup using more advanced FPGA boards.

We embedded the PAU as a new SFU (at the microarchitecture and circuit levels) for use in modern microprocessors. The PAU can be programmed for a variety of arithmetic applications. We use a high speed ring NoC data fabric in the PAU to transfer data from the control logic and different IP blocks. The PAU can have multiple IP blocks of the same type. The control logic in the PAU is constructed using a LUT design like a traditional FPGA, and can be reconfigured for different arithmetic applications. We have applied three arithmetic application examples in the PAU at 16nm technology node, and compared their performance with software-based implementations. The PAU achieved speedups ranges from 3.3$\times$ to 832.3$\times$ compared to a software-based implementation. The PAU demonstrated minimal power increase ranging from 0.18% to 7.56% compared to the Intel i7-5600U processor. The PAU has an area increase ranging from 10.7% to 545.3% compared to a 32kB L1D cache. In a Viterbi decoder, the PAU design with $r = \frac{7}{8}$ and $l = 6$, achieved a speedup of 832.3$\times$, an area increase of 54.8% and a power increase of 0.99%.

Today's applications require new SFUs in modern microprocessors to meet the high demand in their performance computations. Using our HU and PAU designs in modern microprocessors can increase the performance significantly, as we demonstrated in this thesis, with a minimal power increase and an acceptable area increase in the die area.

REFERENCES

[1] Our World in Data, “https://ourworldindata.org/technological-progress.”

[2] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[3] A. Fairouz, M. Abusultan and S. P. Khatri, “A Novel Hardware Hash Unit Design for Modern Microprocessors,” in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pp. 412–415, Oct 2016.

[4] A. Fairouz, M. Abusultan and S. P. Khatri, “Circuit Level Design of a Hardware Hash Unit for Use in Modern Microprocessors,” in *Proceedings of the on Great Lakes Symposium on VLSI (GLSVLSI) 2017*, pp. 101–106, ACM, May 2017.

[5] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, “Efficient Hardware Hashing Functions for High Performance Computers,” *IEEE Transactions on Computers*, vol. 46, pp. 1378–1381, Dec 1997.

[6] NCSU EDA, “http://www.eda.ncsu.edu/wiki/FreePDK45:Contents.”

[7] A. Fairouz and S. P. Khatri, “An FPGA-based Coprocessor for Hash Unit Acceleration,” in *2017 IEEE International Conference on Computer Design (ICCD)*, pp. 301–304, Nov 2017.

[8] A. Mandal, S. P. Khatri, and R. N. Mahapatra, “A Fast, Source-Synchronous Ring-based Network-on-Chip Design,” in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1489–1494, March 2012.

[9] V. H. Cordero and S. P. Khatri, “Clock Distribution Scheme using Coplanar Transmission Lines,” in *2008 Design, Automation and Test in Europe*, pp. 985–990, March 2008.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[11] G. E. Moore, “Cramming more Components onto Integrated Circuits,” *IEEE Solid-State Circuits Society Newsletter*, vol. 11, pp. 33–35, Sep. 2006.

[12] A. Fairouz, M. Abusultan, A. Elshennawy and S. P. Khatri, “Comparing Leakage Reduction Techniques for an Asynchronous Network-on-Chip Router,” *Journal of Low Power Electronics (JOLPE)*, vol. 14, pp. 414–427, Sept 2018.

[13] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS ’67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[15] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-programmable Gate Arrays*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[16] S. M. Trimberger, *Field-Programmable Gate Array Technology*. Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[17] A. Sangiovanni-Vincentelli, “The Tides of EDA,” *IEEE Des. Test*, vol. 20, pp. 59–75, Nov. 2003.

[18] I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 26, pp. 203–215, Feb. 2007.

[19] J. Carter and M. N. Wegman, “Universal classes of hash functions,” *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143 – 154, 1979.

[20] K. Pagiamtzis and A. Sheikholeslami, “Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, pp. 712–727, March 2006.

[21] Intel Corporation, “http://www.intel.com.”

[22] GEM5 Simulator: A Modular Platform for Computer Architecture Research, "http://www.gem5.org."

[23] F. Yamaguchi and H. Nishi, "Hardware-based Hash Functions for Network Applications," in *2013 19th IEEE International Conference on Networks (ICON)*, pp. 1–6, Dec 2013.

[24] N. Hua, E. Norige, S. Kumar, and B. Lynch, "Non-crypto Hardware Hash Functions for High Performance Networking ASICs," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pp. 156–166, Oct 2011.

[25] B. Baldwin, A. Byrne, L. Lu, M. Hamilton, N. Hanley, M. O'Neill, and W. P. Marnane, "A Hardware Wrapper for the SHA-3 Hash Algorithms," in *Signals and Systems Conference (ISSC 2010), IET Irish*, pp. 1–6, June 2010.

[26] A. Satoh, "ASIC Hardware Implementations for 512-bit Hash Function Whirlpool," in *2008 IEEE International Symposium on Circuits and Systems*, pp. 2917–2920, May 2008.

[27] R. Dobai and J. Korenek, "Evolution of Non-Cryptographic Hash Function Pairs for FPGA-based Network Applications," in *Computational Intelligence, 2015 IEEE Symposium Series on*, pp. 1214–1219, Dec 2015.

[28] Y. k. Lai and G. T. Byrd, "Stream-based Implementation of Hash Functions for Multi-Gigabit Message Authentication Codes," in *2006 Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'06)*, pp. 150–155, Dec 2006.

[29] L. Ioannou, H. E. Michail, and A. G. Voyiatzis, "High Performance Pipelined FPGA Implementation of the SHA-3 Hash Algorithm," in *2015 4th Mediterranean Conference on Embedded Computing (MECO)*, pp. 68–71, June 2015.

[30] D. Pao, X. Wang, and Z. Lu, "Design of a Near-minimal Dynamic Perfect Hash Function on Embedded Device," in *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, pp. 457–462, Jan 2013.

[31] D. Tong, S. Zhou, and V. K. Prasanna, “High-Throughput Online Hash Table on FPGA,” in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 105–112, May 2015.

[32] M. Hanna, S. Demetriades, S. Cho, and R. Melhem, “Progressive Hashing for Packet Processing Using Set Associative Memory,” in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS ’09, (New York, NY, USA), pp. 153–162, ACM, 2009.

[33] Y. Li, “Non-collision Hash Scheme Using Bloom Filter and CAM,” in *Web Mining and Web-based Application, 2009. WMWA ’09. Second Pacific-Asia Conference on*, pp. 55–58, June 2009.

[34] T. Kohonen, “Content-Addressable Memories,” *2nd ed. New York: Springer-Verlag*, 1987.

[35] L. Chisvin and R. J. Duckworth, “Content-addressable and Associative Memory: Alternatives to the Ubiquitous RAM,” *IEEE Computer*, vol. 22, pp. 51–64, July 1989.

[36] M. V. Ramakrishna and G. A. Portice, “Perfect hashing functions for hardware applications,” in *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pp. 464–470, Apr 1991.

[37] S. C. Liu, F. A. Wu, and J. B. Kuo, “A Novel Low-voltage Content-addressable-memory (CAM) Cell with a Fast Tag-compare Capability using Partially Depleted (PD) SOI CMOS Dynamic-threshold (DTMOS) Techniques,” *IEEE Journal of Solid-State Circuits*, vol. 36, pp. 712–716, Apr 2001.

[38] E. Shen and J. B. Kuo, “0.8 V CMOS Content-addressable-memory (CAM) Cell Circuit with a Fast Tag-compare Capability using Bulk PMOS Dynamic-threshold (BP-DTMOS) Technique Based on Standard CMOS Technology for Low-voltage VLSI Systems,” in *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, vol. 4, pp. IV–583–IV–586 vol.4, 2002.

[39] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[40] D. Interactive, "Memcached." https://memcached.org.

[41] The PARSEC Benchmark Suite, "http://parsec.cs.princeton.edu."

[42] P. C. K. Lin, A. Mandal, and S. P. Khatri, "Boolean Satisfiability using Noise Based Logic," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1256–1257, June 2012.

[43] Advanced Micro Devices, Inc. (AMD), "http://www.amd.com/en-gb/products/server."

[44] International Business Machines Corp. (IBM), "http://www-03.ibm.com/systems/power/hardware/."

[45] Synopsys HSPICE, "https://www.synopsys.com/verification/ams-verification/circuit-simulation/hspice.html."

[46] Predictive Technology Model, "http://ptm.asu.edu."

[47] Synopsys Raphael, "https://www.synopsys.com/silicon/tcad/interconnect-simulation/raphael.html."

[48] Y. Du, G. He, and D. Yu, "Efficient Hashing Technique Based on Bloom Filter for High-Speed Network," in *2016 8th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, vol. 01, pp. 58–63, Aug 2016.

[49] F. Kahri, H. Mestiri, B. Bouallegue, and M. Machhout, "Efficient FPGA Hardware Implementation of Secure Hash Function SHA-256/Blake-256," in *2015 IEEE 12th International Multi-Conference on Systems, Signals Devices (SSD15)*, pp. 1–5, March 2015.

[50] Y. Vizilter, V. Gorbatsevich, A. Vorotnikov, and N. Kostromov, "Real-Time Face Identification via CNN and Boosted Hashing Forest," in *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 146–154, June 2016.

[51] J. Lin and O. Morére and J. Petta and V. Chandrasekhar and A. Veillard, “Tiny Descriptors for Image Retrieval with Unsupervised Triplet Hashing,” in *2016 Data Compression Conference (DCC)*, pp. 397–406, March 2016.

[52] B. Salami, O. Arcas-Abella, and N. Sonmez, “HATCH: Hash Table Caching in Hardware for Efficient Relational Join on FPGA,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 163–163, May 2015.

[53] D. K. Shedge and V. Agey, “Different Types of SRAM Chips for Power Reduction: A Survey,” in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 974–979, March 2016.

[54] Perl, “http://www.perl.org.”

[55] M. I. Rahman, T. Bashar, and S. Biswas, “Performance evaluation and read stability enhancement of SRAM bit-cell in 16nm CMOS,” in *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, pp. 713–718, May 2016.

[56] Cadence Design Systems, Inc, “Virtuoso Layout Suite.” http://www.cadence.com.

[57] “Synopsys Verilog to Spice (V2S).” www.synopsys.com.

[58] “Synopsys VCS.” https://www.synopsys.com/verification/simulation/vcs.html.

[59] Large-Scale Reconfigurable Computing in a Microsoft Datacenter, “https://www.microsoft.com/en-us/research/wp-content/uploads/2014/06/HC26.12.520-Recon-Fabric-Pulnam-Microsoft-Catapult.pdf.”

[60] Q. Yanghua, N. Kapre, H. Ng, and K. Teo, “Improving Classification Accuracy of a Machine Learning Approach for FPGA Timing Closure,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 80–83, May 2016.

[61] M. Hadi, R. T. Widodo, and A. H. Alasiry, "Implementation of FPGA Technology as Human-Machine Interface to Z80 Microprocessor Learning Module," in *2016 International Electronics Symposium (IES)*, pp. 180–184, Sept 2016.

[62] Xilinx ISE Design Suite, "http://www.xilinx.com/products/design-tools/ise-design-suite.html."

[63] VirusShare, "http://www.virusshare.com."

[64] C. Hudel and M. Shehab, "Optimizing Search for Malware by Hashing Smaller Amounts of Data," in *World Congress on Internet Security (WorldCIS-2013)*, pp. 112–117, Dec 2013.

[65] N. F. Huang, C. N. Kao, and R. T. Liu, "A Novel Software-based MD5 Checksum Lookup Scheme for Anti-virus Systems," in *2011 7th International Wireless Communications and Mobile Computing Conference*, pp. 207–212, July 2011.

[66] L. Wu and Y. Zhang, "Automatic Detection Model of Malware Signature for Anti-virus Cloud Computing," in *2011 10th IEEE/ACIS International Conference on Computer and Information Science*, pp. 73–75, May 2011.

[67] Xilinx NetFPGA-1G-CML Kintex-7 FPGA Development Board, "http://www.xilinx.com/products/boards-and-kits/1-4le3gu.html."

[68] Xilinx CORE Generator System, "http://www.xilinx.com/products/design-tools/coregen.html."

[69] Linux Kernel Organization, Inc., "http://www.kernel.org."

[70] Xillybus Ltd., "http://www.xillybus.com."

[71] Canonical Ltd., "http://www.ubuntu.com."

[72] M. K. A. Shatnawi and H. A. Shatnawi, "A Performance Model of Fast 2D-DCT Parallel JPEG Encoding using CUDA GPU and SMP-Architecture," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, Sep. 2014.

[73] A. B. Watson, "Image Compression Using the Discrete Cosine Transform," *Mathematica Journal*, vol. 4, pp. 81–88, 1994.

[74] Synopsys Design Compiler (DC): RTL Synthesis, "https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html."

[75] Xilinx Vivado Design Suite, "https://www.xilinx.com/products/design-tools/vivado.html."

[76] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse Grained Reconfigurable Architectures in the Past 25 Years: Overview and Classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 235–244, July 2016.

[77] R. A. Bittner, P. M. Athanas, and M. D. Musgrove, "Colt: An Experiment in Wormhole Run-Time Reconfiguration," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 2914, 08 1998.

[78] A. Alsolaim, J. Becker, M. Glesner, and J. Starzyk, "Architecture and Application of a Dynamically Reconfigurable Hardware Array for Future Mobile Communication Systems," in *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*, pp. 205–214, April 2000.

[79] H. Park, Y. Park, and S. Mahlke, "Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 370–380, Dec 2009.

[80] S. Fiske and W. J. Dally, "The Reconfigurable Arithmetic Processor," in *[1988] The 15th Annual International Symposium on Computer Architecture. Conference Proceedings*, pp. 30–36, May 1988.

[81] K. Sakiyama, N. Mentens, L. Batina, B. Preneel, and I. Verbauwhede, "Reconfigurable Modular Arithmetic Logic Unit Supporting High-Performance RSA and ECC Over GF(p)," *International Journal of Electronics*, vol. 94, no. 5, pp. 501–514, 2007.

[82] W. Wang, Y. Ding, S. Cao, and X. Zhao, “Design of a Dynamically Reconfigurable Arithmetic Unit for Matrix Algorithms,” in *2015 IEEE 11th International Conference on ASIC (ASICON)*, pp. 1–4, Nov 2015.

[83] P. M. Grant, “Digital Signal Processing. 1. Digital Filters and the DFT,” *Electronics Communication Engineering Journal*, vol. 5, pp. 13–21, Feb 1993.

[84] G. D. Forney, “The Viterbi Algorithm,” *Proceedings of the IEEE*, vol. 61, pp. 268–278, March 1973.

[85] Cadence Design Systems, Inc, “Stratus High-Level Synthesis (HLS).” http://www.cadence.com.

[86] Xilinx Kintex UltraSCALE+, “https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale-plus.html.”

[87] Python, “https://www.python.org.”

[88] P. S. Battiato, “High Performance Median Filtering Algorithm Based on NVIDIA GPU Computing,” *International Symposium for Young Scientists in Technology*, pp. 1–10, 2016.

[89] A. Smirnov and T. cker Chiueh, “An Implementation of a FIR Filter on a GPU,” *Experimental Computer Systems Lab, Stony Brook University, Tech. Rep*.

[90] M. Masoumi and H. Ahmadifar, “Performance of HEVC discrete cosine and sine transforms on GPU using CUDA,” in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pp. 0857–0861, Dec 2017.

[91] R. Li, Y. Dou, and D. Zou, “Efficient Parallel Implementation of Three-Point Viterbi Decoding Algorithm on CPU, GPU, and FPGA,” *Concurrency and Computation: Practice and Experience*, vol. 26, no. 3, pp. 821–840, 2014.

[92] D. Zhang, R. Zhao, L. Han, T. Wang, and J. Qu, “An Implementation of Viterbi Algorithm on GPU,” in *Proceedings of the 2009 First IEEE International Conference on Information*

Science and Engineering, ICISE '09, (Washington, DC, USA), pp. 121–124, IEEE Computer Society, 2009.

[93] V. V. Fedorov, M. Abusultan, and S. P. Khatri, "FTCAM: An Area-Efficient Flash-Based Ternary CAM Design," *IEEE Transactions on Computers*, vol. 65, pp. 2652–2658, Aug 2016.

[94] IBM: Coherent Accelerator Processor Interface (CAPI), "https://developer.ibm.com/linuxonpower/capi/."

[95] Microsoft Azure, "https://azure.microsoft.com/."