Hardware Acceleration of Hash Operations in Modern Microprocessors

Abbas A. Fairouz*, *Member, IEEE,* Monther Abusultan†, Viacheslav V. Fedorov†, and Sunil P. Khatri†, *Member, IEEE*

Abstract—Modern microprocessors contain several special function units (SFUs) such as specialized arithmetic units, cryptographic processors, etc. In recent times, applications such as cloud computing, web-based search engines, and network applications are widely used, and place new demands on the microprocessor. Hashing is a key algorithm that is extensively used in such applications. Hashing can reduce the complexity of search and lookup from O(N) to O(N/n), where n bins are used. Hashing is typically performed in software. Thus, implementing a hardware-based hash unit on a modern microprocessor would potentially increase performance significantly. In this paper, we propose a novel hardware hash unit (HU) design for use in modern microprocessors, at the microarchitecture level and at the circuit level. First, we present the design of the HU at the microarchitecture level. We simulate the HU to compare its performance with a software-based hash implementation. We demonstrate a significant speedup (up to 15×) for the HU. Furthermore, the performance scales elegantly with increasing database size and application diversity, without increasing the hardware cost. Second, we present the circuit design of the HU for use in modern microprocessors, using a 45nm technology. Our proposed hardware hash unit is based on the use of a content-addressable memory (CAM) to implement each bin of the hash function. We simulate the HU circuit and compare it with a traditional CAM design. We demonstrate an average power reduction of 5.48× using the HU over the traditional CAM. Also, we show that the HU can operate at a maximum frequency of 1.39 GHz (after accounting for process, voltage and temperature (PVT) variations and accounting for wiring parasitics). Furthermore, we present the delay, power and area trade-offs of the HU design with varying hash table sizes.

Index Terms—Special Function Unit (SFU), Hashing, Hardware Hash Unit, Modern Microprocessors, Hash Table, Hash Function, Content-Addressable Memory (CAM).

1 Introduction

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 processing technology. As a result, new techniques need to be explored to achieve high performance computations that meet the diverse demands placed on modern microprocessors.

There have been many special function units (SFUs) introduced to speedup commonly occurring tasks in the application set. Some of these include cryptographic units, floating point units and memory management units. To the best of authors' knowledge, there has been no work on a general purpose hash unit. In order to perform such hashing-intensive algorithms, microprocessors simply compile the code and run the resulting instructions. As we demonstrate in this paper, 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. Designing 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. Most hash function and hash table designs have to be implemented separately per application. Another common use of hash functions involves search and membership checks.

Modern microprocessors do not have a hardware-based hash unit. In this paper, we propose a new hardware hash unit (HU) as a special function unit for use in modern microprocessors. We present the microarchitecture and the circuit level designs of the HU. The hash unit uses a special hash table memory, to store hash operations. The hash table memory module uses content-addressable memory (CAM) to enable fast memory access. Our HU is embedded in the architecture pipeline of modern microprocessors. Our work in this paper is an extension of our previous works in [1], [2].

The HU design utilizes a hash function of class H3 [3] and a CAM-based implementation of the hash table bins. For our microarchitectural design of the HU, we obtain a speedup of up to 15×, with minimal increase in the area. For our circuit design of the HU, we implement the HU in a 45nm technology, We obtain an average power reduction of 5.48× using the HU over the traditional CAM, and a clock frequency of up to 1.39 GHz.

The key contributions of this paper 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 HU at the microarchitecture level using GEM5 [4] (a Computer Architecture Platform) in the x86 ISA and verified the correctness of all hash operations.

• * *The author is with Computer Engineering Department, College of Engineering & Petroleum, Kuwait University, Kuwait.* † *The authors are with Electrical and Computer Engineering Department, Texas A&M University, College Station, Texas, USA.*

- We observe a speedup of up to 15× while varying the size of the hardware hash table, CPU speed, cache size, main memory (DRAM) technologies, 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 increases the performance of these applications.
- Our approach supports multiple applications running hash operations in parallel, without affecting the correctness of any of the applications.
- We implement the HU at the circuit level for use in modern microprocessors.
- We simulated the HU circuit design in Synopsys VCS and HSPICE [5], using a 45nm PTM [6] and verified the correctness of lookup, insert and delete hash operations. RC parasitic are extracted using Synopsys Raphael [5].
- We demonstrate an average power reduction of 5.48× 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 paper is organized as follows. In Section 2, we discuss related previous work. Section 3 describes our microarchitecture approach, and Section 4 discusses our circuit design approach. In Section 5 we present experimental results of the microarchitecture simulations, and we present experimental results of the circuit simulations in Section 6. We conclude in Section 7.

2 Previous Work

In this work, we present a hardware-based realization of a hash function *and* a hash table, at the microarchitecture level and at the circuit level.

We first discuss previous research which focused solely on implementing hash function alone. Hardware hash function implementations have been proposed in [7], [8], [9], [10]. 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 [11] and detect and authenticate messages [12]. A hashing technique has been proposed in [13] for high speed networks. In [14], 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 [15]. 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 [16]. Image compression techniques use hashing in [17] to speed up data compression.

Unlike the above 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.

Next, we discuss previous work in implementing hash tables. In [18], 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 [19]. A hashing scheme design has been utilized in [20] for packet processing. The authors implemented the hash table as a set-associative memory module. They simulated their design with various hash functions in C++. Furthermore, they proposed a multiple hash functions to reduce collision list in hash table. A key difference of [18], [19], [20] from our approach is that in [18], [19], [20], 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. We design the hardware hash function and hash table at the microarchitecture level and at the circuit level.

3 The HU Microarchitecture Design

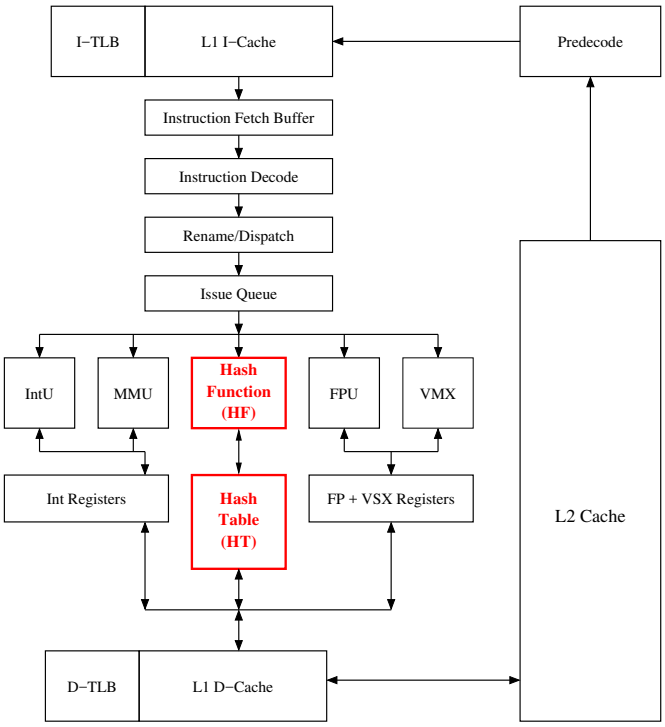


Fig. 1: Proposed Hash Unit (HU) for Modern Microprocessors

In this section, we first discuss our proposed hash unit (HU) in modern microprocessors, at the microarchitecture level design. The HU is comprised of a hardware hash function (HF) and a hardware hash table (HT), as shown in Figure 1. Next, we present a brief overview of the hash function (HF). Then, we introduce the hardware structure of the HT in our approach. Next we discuss the functionality of the HU. After that, we discuss the latency of our hash table implementation, along with cache and main memory latencies. Finally, we discuss the benchmarks used for quantifying the usefulness of the HU.

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 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 [21], [22]. We implement one CAM per bin of the HT. We will discuss HF in Section 3.2 and the HT in Section 3.3.

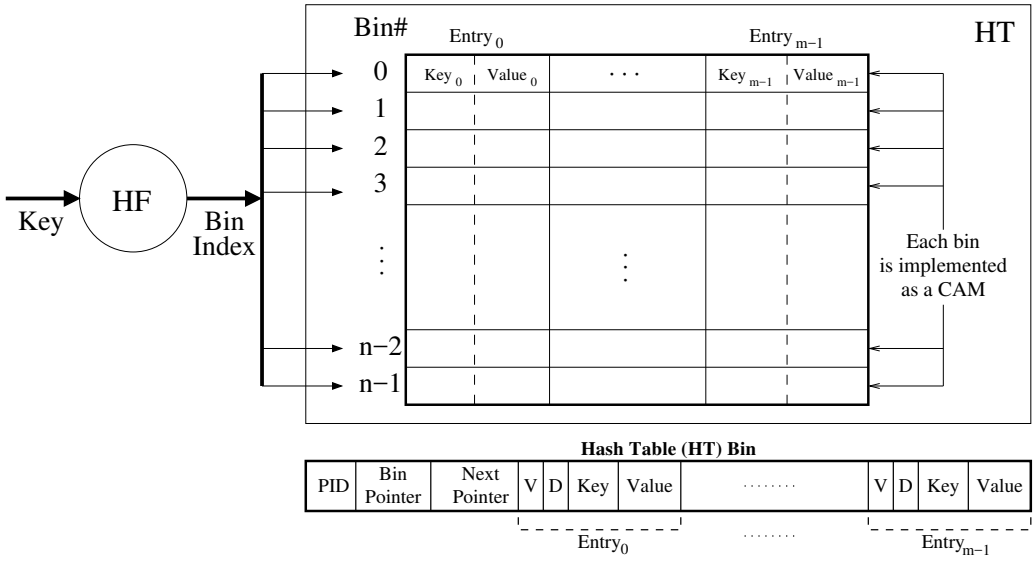


Fig. 2: HF, HT, and HT Bin Configuration

3.2 Hash Function (HF)

The hash function performs a hash operation based on a key and value pair. The HF takes a key as an input to produce a bin index as shown on the left side of Figure 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 [3], [23]. 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.

Let's show an example of H3 class hash functions. Assume that we have a 4-bit input key to a hash function h_k of class H3, that generates a 3-bit output $index$. Each H3 class hash function has a *q-matrix* [3], [23] used to hash an n bits input ($n = 4$), to generate an m bit output ($m = 3$). The sample space of the input $key = \{0, ..., 15\}$, and the sample space of the output $index = \{0, ..., 7\}$. Assume that we have a randomly chosen 4×3 *q-matrix*:

$$q = \begin{bmatrix} 101 \\ 001 \\ 010 \\ 111 \end{bmatrix} \tag{1}$$

Then the output $index$ of the hash function h_k for the input $key = (13)_{10}$ is:

$$\begin{aligned} h_k(13)_{10} &= h_k(1101)_2 = q(0) \oplus q(1) \oplus q(3) \\ &= (101)_2 \oplus (001)_2 \oplus (111)_2 \\ &= (011)_2 = (3)_{10} \end{aligned} \tag{2}$$

3.3 Hash Table (HT) Configuration

In order to provide fast memory access, we design our HT using CAM memory blocks [24], [25]. 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 3.2, the bin index produced by the HF will point to a bin in the HT. If an entire bin is utilized, the next available entry in the insert operation will go to DRAM, to effectively extending 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 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 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 $A+m$. 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,

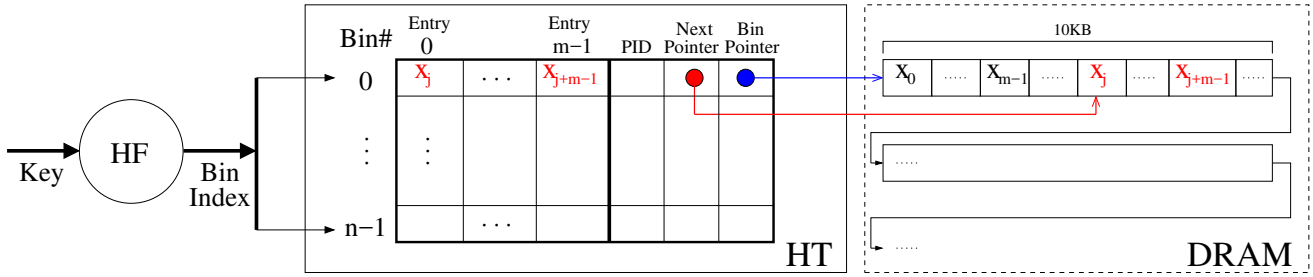


Fig. 3: Replacement operation in the HU – in case of a HT miss and a DRAM hit of a lookup operation

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 0. This indicates that the HT is empty.

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 instruction set. Since our HT is constructed using CAMs [21], [22], HT lookups take one clock cycle, unless the bin data does not fit in the HT bin. These operations are described as follows (including a replacement operation):

- 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 aborted. 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.
- Replacement: 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, the *Bin Pointer* refers to the beginning of the bin in the DRAM, while the *Next Pointer* refers 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 in Figure 3. 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.

3.5 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. The best-case scenario is when we only have a single hash application running at a time, and all the entries fit in the HT. In this case, each hash operation completes in one clock cycle. In contrast, the worst-case scenario is when you have multiple hash applications running at the same time, and they cannot fit in the last level cache (LLC). In this case, the DRAM access latency decreases the overall performance of hash applications, as discussed in Section 5.

3.6 Benchmarks Used

We created a hash benchmark (B_H) which provides both insert and lookup operations with random entries, using Yahoo! Cloud Serving Benchmark (YCSB) [26]. 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. For constructing the hash benchmark B_H, we use Memcached [27] as a back-end 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× the size of the HT. We also study the performance of the HU as the number of parallel instances of B_H is increased. We also use programs from the PARSEC benchmarks suite [28], [29] to test the performance of the HU under a diverse configuration of other applications running in parallel with the B_H benchmark. 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_P instances versus the number of B_H instances, we can study the HU performance under varying system loads.

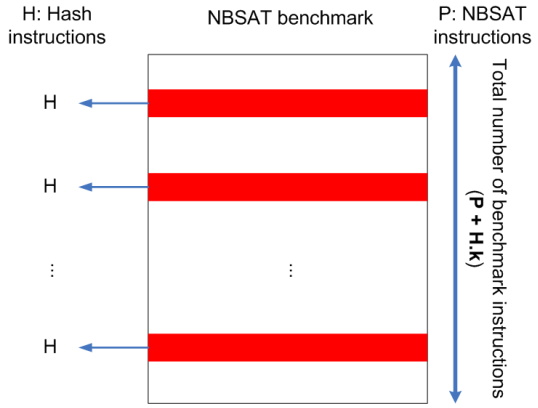


Fig. 4: Percentage of Hash Instructions in NBSAT Benchmark to generate a B_{HV} Benchmark

The B_H benchmark performs only hash operations. In order to create a benchmark in which a user-defined fraction of instructions are hash operations, we modified a Noise-based Boolean Satisfiability (NBSAT) [30] benchmark. In this benchmark, we injected hash instructions, to obtain a benchmark (B_{HV}) with a user-defined fraction of hash instructions. Assume that the original NBSAT benchmark has P instructions, as shown in Figure 4. 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_{HV} ($\%Hash_{inst}$) is shown in Equation 3. By varying k, we vary the *density* of hash instructions in B_{HV}, and test the performance of the HU while varying k.

$$\%Hash_{inst} = \frac{H.k}{P + (H.k)} \tag{3}$$

In Section 5, we will quantify the performance of the HU, and compare it with a software-based hashing approach. In the next section, we will discuss the HU circuit design.

4 The HU Circuit Design

In Section 3, we discussed the hash unit (HU) microarchitecture design. In this section, we discuss our circuit implementation of the HU for use as an SFU in modern microprocessors. The block diagram of the HU is shown in Figure 5. We start with a top-level discussion of the HU circuitry. Next, we discuss the hash function (HF) of class H3 [3] (as shown in Figure 6) and the design of the *Bin Selector* circuit. Then, we introduce the control signals unit (CSU) circuit design as shown in Figure 7. Finally, we discuss the hash table (HT) circuit design as shown in Figure 8.

4.1 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 5. This figure shows the 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.
- 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 logic 0 to the valid bit of the entry. This is shown in Figure 8 and will be explained later.

As shown in Figure 5, 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.

4.2 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 5. 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 5, 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 [3]. It performs hash operations based on AND and XOR logic as shown in Figure 6. 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 [3]. 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 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 5. 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 (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 1 illustrates the mapping between the operations and the values of (W2,W1) and the (C1,C2) signals.

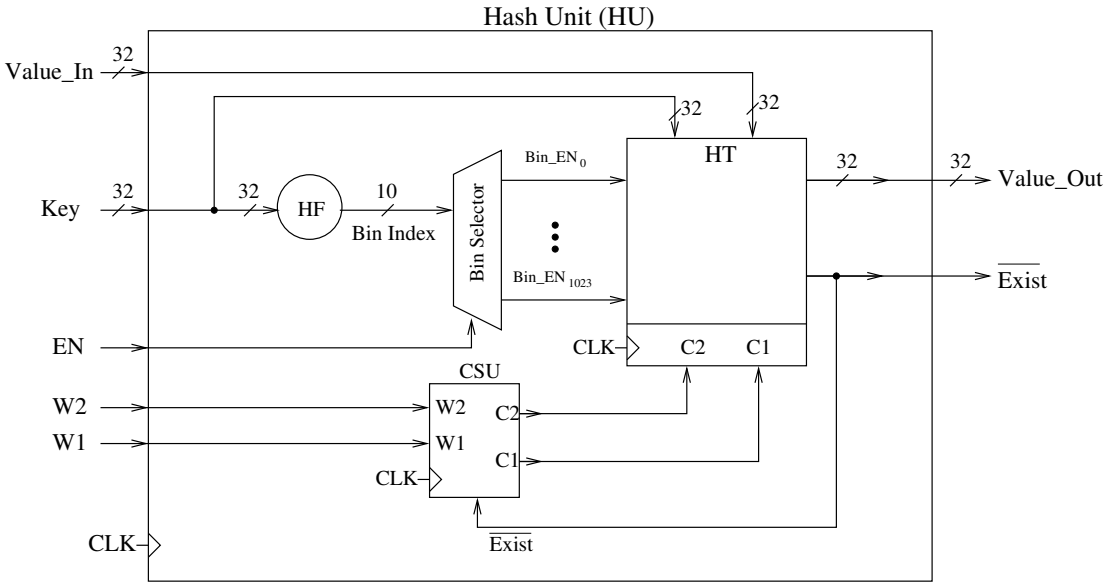


Fig. 5: Hardware Hash Unit (HU) Block Diagram

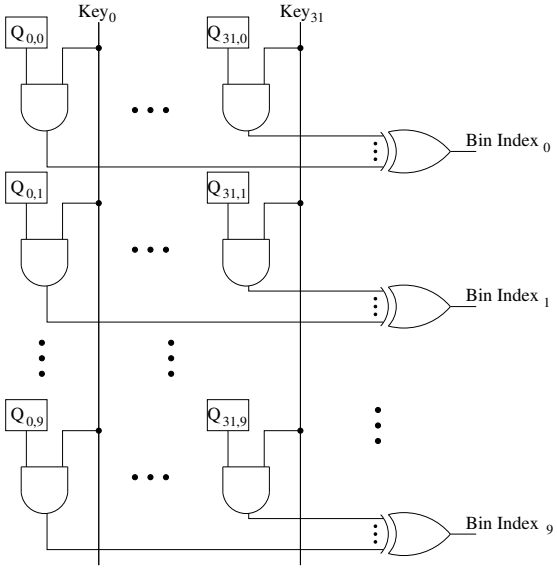


Fig. 6: Hardware Hash Function (HF) of Class H3 [3]

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 1: CSU op-codes for the HT operations

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 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 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,

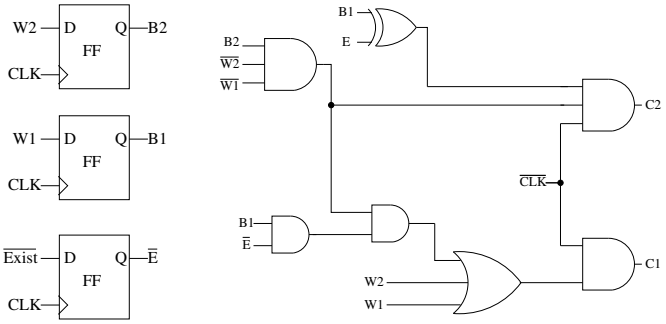


Fig. 7: Control Signals Unit (CSU)

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 7.

4.4 Hash Table (HT)

We design each bin of our Hash Table (HT) using a basic 9-T NOR-type CAM cell [31] to store a *key*, and a conventional 6-T SRAM cell [32] to store a *value*. Each HT bin is implemented as a CAM in order to perform fast hash operations.

The HT has 6 inputs and 2 outputs as shown in Figure 5. 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 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

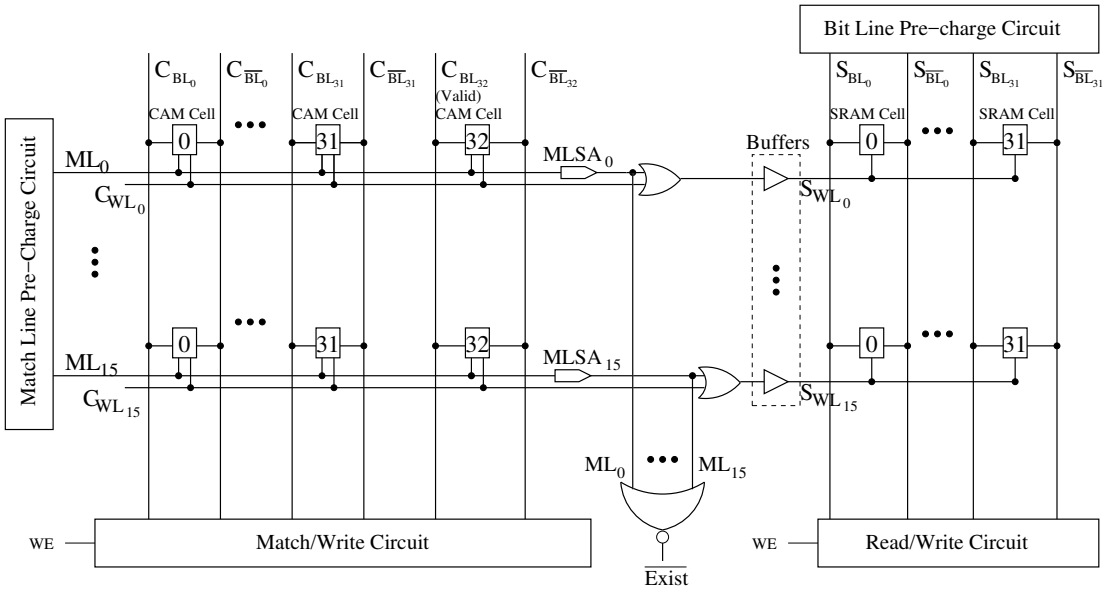


Fig. 8: Hardware Hash Table (HT) for a Single Bin

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 8, 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 8). 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}, $S_{\overline{BL}}$). There is a Match/Write circuit for CAM (since the polarity of C_{BL_i} and $C_{\overline{BL_i}}$ 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 $S_{\overline{BL_i}}$ 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.

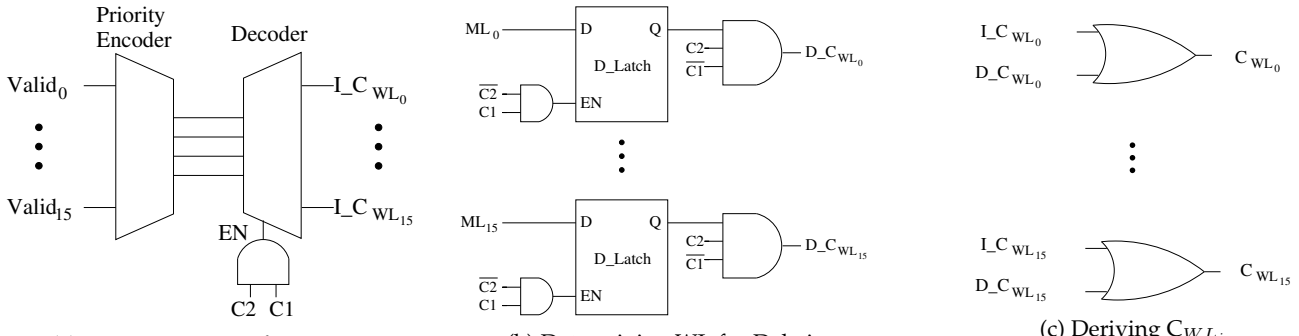
During a lookup operation, all C_{WL_i} lines are driven to a *zero* value. All ML lines are pre-charged, and at most one (say ML_i) stays *high* (indicating a match). This ML_i is amplified by the sense amplifier $MLSA_i$, and results in the $\overline{Exist}$ line being driven to a *zero* value. If no ML_i stays *high* (a mis-match condition), then $\overline{Exist}$ ends up being driven to a logic 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 9 (b). 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 9 (c)). Hence the deletion is performed on the i^{th} row of the CAM, as required.

For an insert operation, we use a priority encoder to find the index of the first row which is not valid. Such a row can be written into since it is invalid. This is shown in Figure 9 (a). 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 9 (a)) and drive a single $I\_C_{WL_i}$ line *high*. This causes C_{WL_i} to be driven *high* (Figure 9 (c)), resulting in an insertion in the first invalid entry as desired.

In Section 5 and Section 6, we will present our experimental results of the HU, at the microarchitecture level and the circuit



(a) Determining WL for Insertion (b) Determining WL for Deletion (c) Deriving C_{WL_i}

Fig. 9: CAM Word Line Signal Generator inside HT

level.

5 Experimental Results: Microarchitecture Level

In this section, we present our experimental results of the HU for use in modern microprocessors at the microarchitecture level.

5.1 Simulation Environment

We implement the HU in GEM5 [4] (a Computer Architecture Simulator). We use x86 ISA as our base instruction set. 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 YCSB workload to create our hash benchmark B_H as discussed in Section 3.6.

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.

5.2 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 [33], while some AMD Athlon parts have L1D of size 128kB [34]. The IBM Power8 processors use an L1D cache of size 64kB [35].

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:

$$Speedup = \frac{SW_{cycles}}{HW_{cycles}} \quad (4)$$

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.

5.3 Results and Analysis

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_H) instances in parallel with multiple PARSEC benchmarks (B_P).
6) Run multiple B_H instances alone.
7) Vary a fraction of hash instructions in the benchmark B_{HV}.

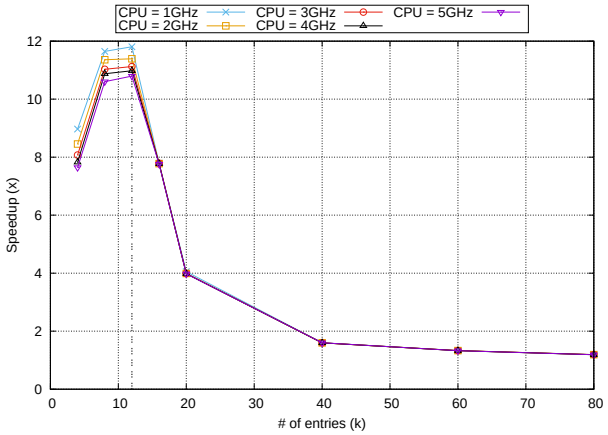


Fig. 10: Speedup as CPU Speed is Varied – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. DRAM: DDR3 1600MHz x64.

CPU speed plays an important role for SW_{hash}. In Figure 10, we vary the CPU speed to study its effect on the performance. The application run is a single instance of B_H, and the simulation is of type G1. We note that the peak speedup is between 10.8× and 11.8×. The speedup is maximum when the entire hash table fits in the HT (i.e. when the hash table has ∼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 11 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×. 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 12 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

Percentage of Hash benchmarks (%)	PARSEC benchmarks B_P									number of instances of B_H
	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 2: PARSEC [29] Benchmarks Utilization in Parallel with Hash Benchmarks

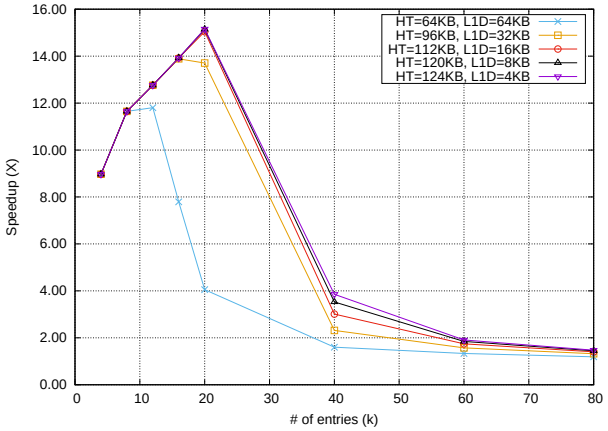


Fig. 11: Tradeoff of HT and L1D Sizes – SW_{hash}: L1D=128kB. DRAM: 1GB DDR3 1600MHz x64. CPU=1GHz.

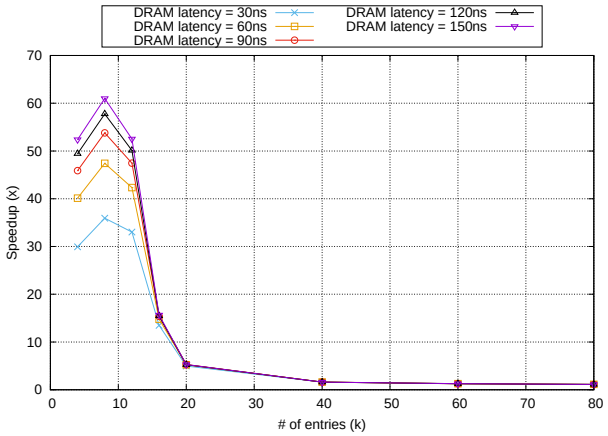


Fig. 13: Vary DRAM Latency – No Caches. HW_{hash}: HT=64kB, L1D=0kB. SW_{hash}: L1D=0kB. CPU=1GHz.

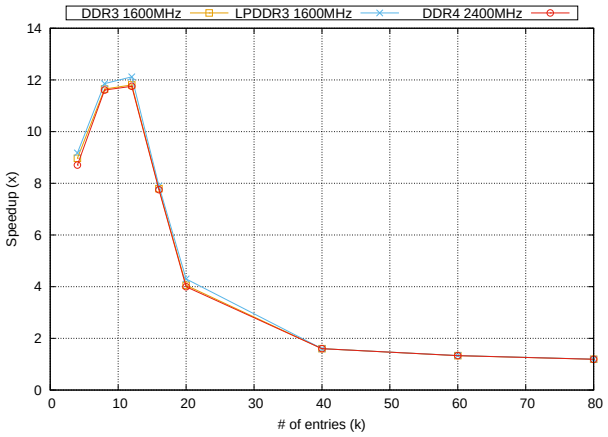


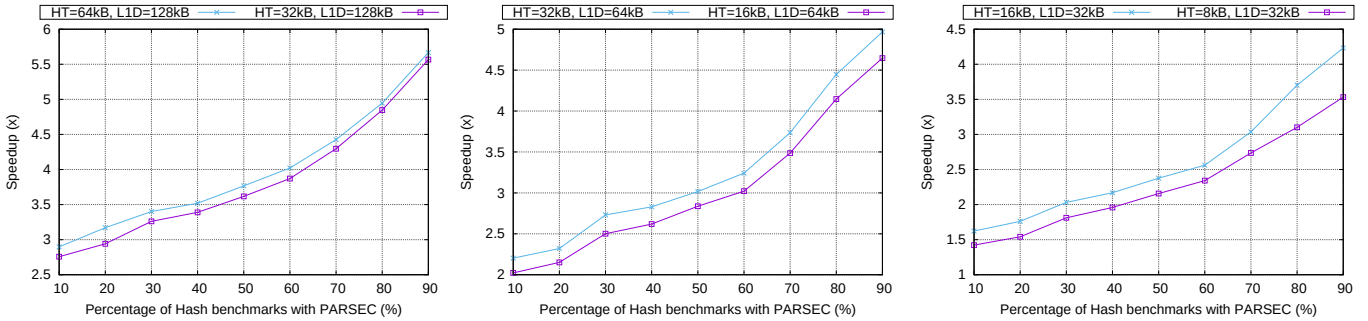
Fig. 12: Vary Memory Types – HW_{hash}: HT=64kB, L1D=64kB. SW_{hash}: L1D=128kB. CPU=1GHz.

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 13. 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× and 61× for a single instance of B_H, This speedup is substantially inversely proportional to DRAM latency.

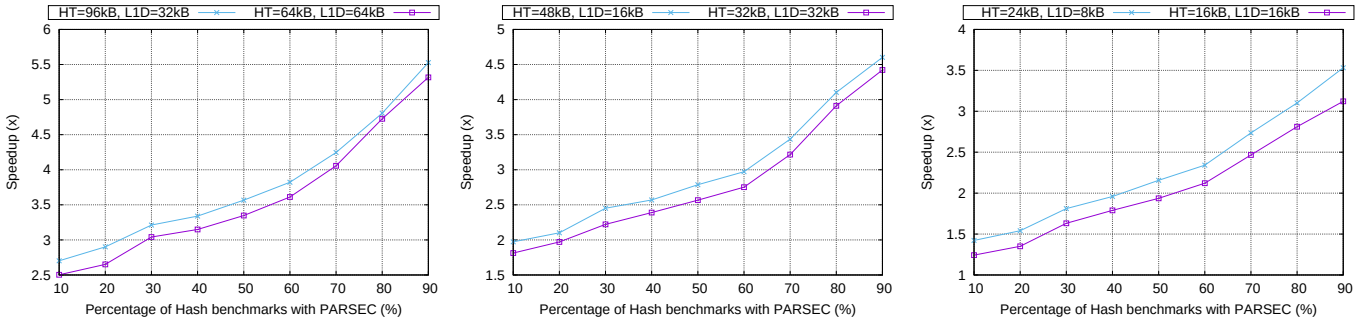
The experiments so far were run for a single instance of B_H. In the experiments that follow, we report the performance for a diverse computational load, with multiple B_H and B_P processes running in parallel. This would result in cache and HT *pollution*, providing a more realistic idea of the value 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_H) 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_H and B_P instances as shown in Table 2. The percent of hash benchmarks is therefore the number of B_H 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 Figure 14 and Figure 15. Figure 14 shows the G2 configuration group, while Figure 15 shows the G1 results. For both Figure 14 and Figure 15, for each plot, we note that the speedup 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 14) is higher than for configuration G1 (Figure 15), as expected, since the total amount of CAM in G2 is larger. The speedup reaches up to 5.7×, in Figure 14a, when size(HT)=64kB and size(L1D)=128kB, and running 9 instances of B_H.

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 16. In Figure 16, 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.



(a) SW_{hash}: L1D=128kB (b) SW_{hash}: L1D=64kB (c) SW_{hash}: L1D=32kB

Fig. 14: Hash and PARSEC Benchmarks in Parallel (B_H and B_P) – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})



(a) HW_{hash}: (HT+L1D)=128kB. SW_{hash}: L1D=128kB (b) HW_{hash}: (HT+L1D)=64kB. SW_{hash}: L1D=64kB (c) HW_{hash}: (HT+L1D)=32kB. SW_{hash}: L1D=32kB

Fig. 15: Hash and PARSEC Benchmarks in Parallel – Simulation Type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

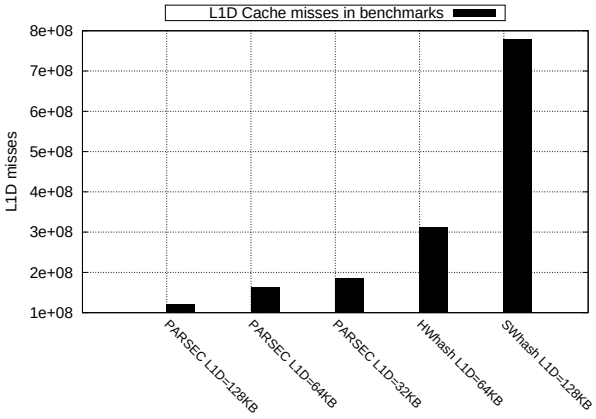


Fig. 16: (HW_{hash}: HT = 64kB) First Three Bars: 9 PARSEC Benchmarks with Varying L1D. Fourth Bar: 9 PARSEC + 1 B_H Running on HW_{hash}. Fifth Bar: 9 PARSEC + 1 B_H Running on SW_{hash}.

In the fifth bar, we simulate the same 9 PARSEC benchmarks along with one instance of B_H, in the software-only (SW_{hash}). Note that the HU reduces the cache misses by about 2.49×.

Figure 17 and Figure 18 report the results from our experiments to quantify the effect of sharing hardware HT among several B_H instances. We observe each G2 plot achieves a greater speedup than the corresponding G1 plot, as expected. For each of the 6 plots in Figure 17 and Figure 18, 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_H contending for the HT resource. When 9 instances of B_H 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 17a).

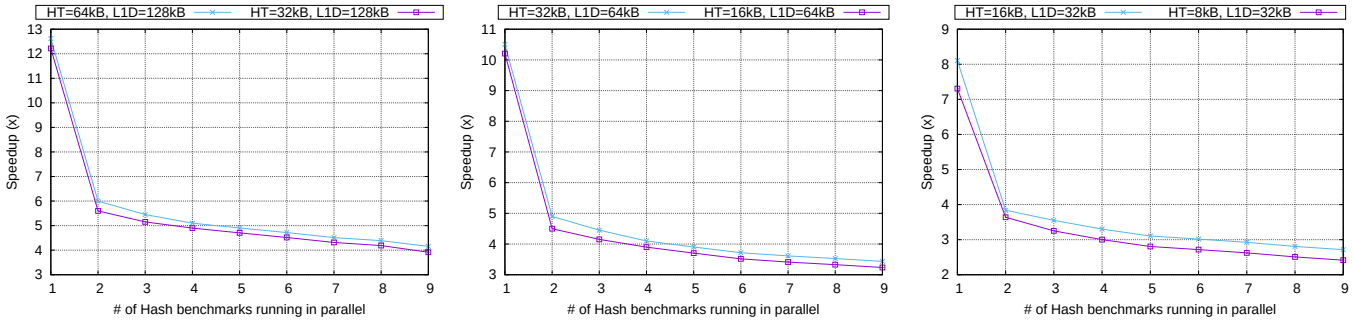
For a single instance of B_H, the speedup is as high as 12.6× (Figure 17a), in which size(HT)=64kB and size(L1D)=128kB. For up to 3 instances of B_H, in all 6 plots, the speedup is higher than 3.25×, reducing gradually as the number of B_H instances increases.

Finally, we perform an experiment to test the speedup due to the HU, for a single benchmark (B_{HV}), as the fraction of hash instructions in the benchmark (%$Hash_{inst}$) varies, as discussed in Section 3.6 and illustrated in Figure 4. The benchmark B_{HV} we chose was NBSAT [30], and the results are shown in Figure 19 and Figure 20. We note that the speedup of the single NBSAT instance (with injected hash instructions) increases as %$Hash_{inst}$ increases, in all 6 plots of Figure 19 and Figure 20. For each plot in Figure 19, the corresponding plot of Figure 20 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 20a).

In the next section, we will discuss the experimental results of the HU at the circuit level.

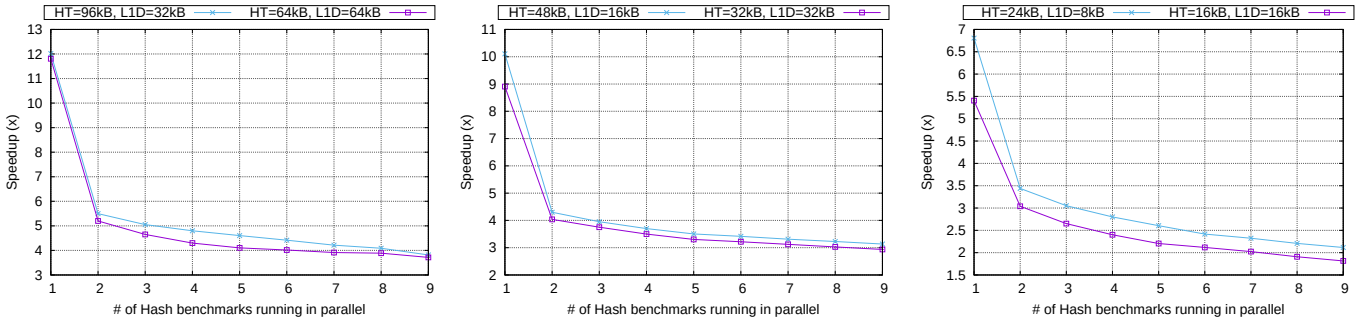
6 Experimental Results: Circuit Level

In this section, we present the simulation results of the HU at



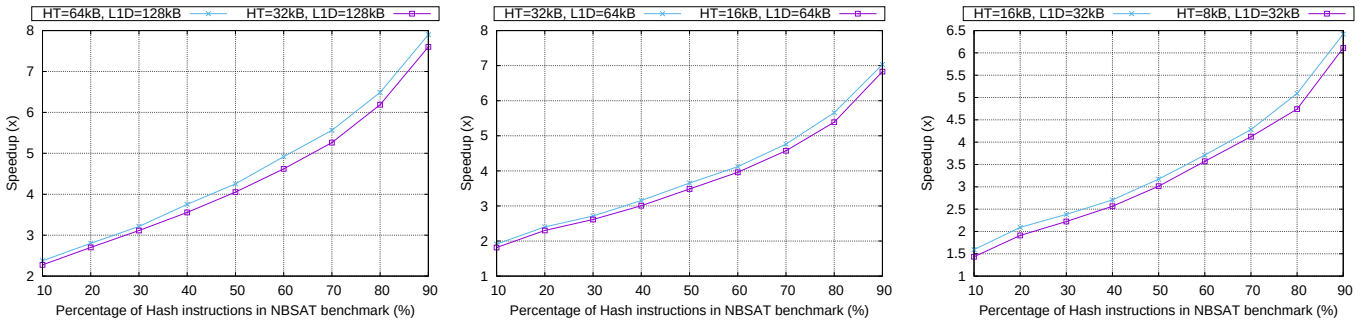
(a) SW_{hash}: L1D=128kB (b) SW_{hash}: L1D=64kB (c) SW_{hash}: L1D=32kB

Fig. 17: Multiple B_H Instances – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})



(a) HW_{hash}: (HT+L1D)=128kB. SW_{hash}: L1D=128kB (b) HW_{hash}: (HT+L1D)=64kB. SW_{hash}: L1D=64kB (c) HW_{hash}: (HT+L1D)=32kB. SW_{hash}: L1D=32kB

Fig. 18: Multiple B_H Instances – Simulation Type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})



(a) SW_{hash}: L1D=128kB (b) SW_{hash}: L1D=64kB (c) SW_{hash}: L1D=32kB

Fig. 19: NBSAT Benchmark (B_{HV}) with $\%Hash_{inst}$ Varying – Simulation Type G2 (size(L1D) in SW_{hash} = size(L1D) in HW_{hash})

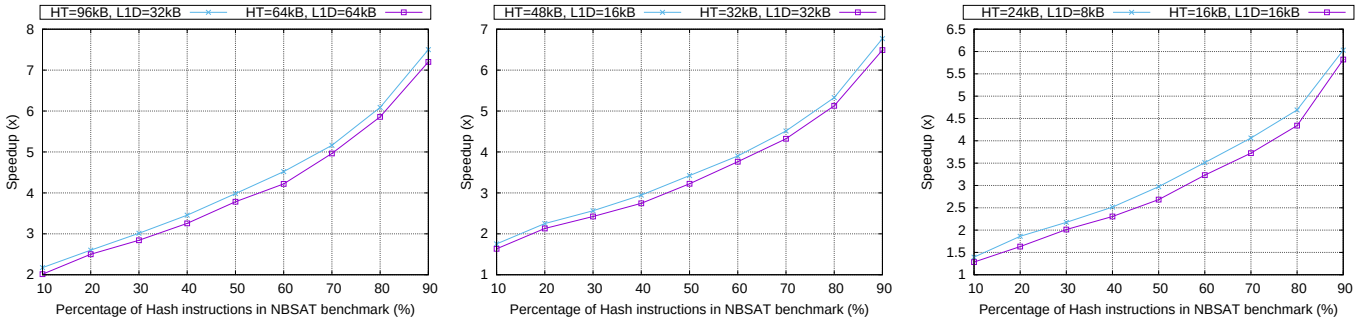
the circuit level. First, we present the simulation environment used to implement the proposed design of the hash unit. Then, we discuss the procedure we used to determine that the design operates correctly. Finally, we present the circuit-level simulation results along with a discussion.

6.1 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 [5] and 45nm PTM [6] high-performance process model card. We used custom Perl [36] 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 [37]. Also, we choose minimum size for the pull-up transistors in the SRAM and CAM cells [37] in order to obtain minimum size, read stability and writeability.



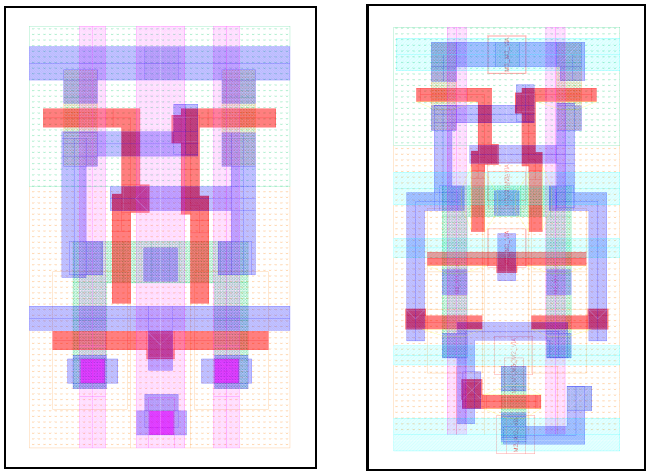
(a) HW_{hash}: (HT+L1D)=128kB. SW_{hash}: L1D=128kB

(b) HW_{hash}: (HT+L1D)=64kB. SW_{hash}: L1D=64kB

(c) HW_{hash}: (HT+L1D)=32kB. SW_{hash}: L1D=32kB

Fig. 20: NBSAT Benchmark (B_{HV}) with $\%Hash_{inst}$ Varying – Simulation Type G1 (size(L1D+HT) in HW_{hash} = size(L1D) in SW_{hash})

We also generated the layout of the CAM and SRAM cells as shown in Figure 21a and Figure 21b. These layouts are based on the 45nm design rules [38] and are generated using the Cadence Virtuoso [39] layout tool. We then used Synopsys Raphael [5] 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 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 parasitic. In our design, we pre-charge the bit-lines of the SRAM cells and the match lines of the CAM cells to VDD.



(a) SRAM cell layout

(b) CAM cell layout

Fig. 21: SRAM and CAM cell layouts in 45nm design rules [38]

6.2 Design Verification

We designed the HF, the *Bin Selector*, and the CSU in structural Verilog and then generated a HSPICE netlist using Synopsys V2S [5]. Then, we exhaustively simulated and verified the logical correctness of these units using Synopsys VCS [5].

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.

6.3 Results and Analysis

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

Table 3 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.

In Table 4, we report results for different HT sizes that were used in the architectural simulations of Section 5. 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 3 to be the slowest and most power hungry operation) and all other bins are not enabled, and consume static power.

From Table 4, 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 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.

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 3: Delay and Dynamic Power analysis of Insert, Delete, and Lookup operations in HT

HT (kB)	Bins	Entries	Area μ^2			HU Operating Speed (GHz)	Total Power (mW)			Dyn Power (mW)	
			HU	Tr.CAM	HU/Tr.CAM		HU	Tr.CAM	Tr.CAM/HU	HF+BS	CSU
8	128	16	120223.11	120117.25	1.0009×	1.39	82	389	4.74×	0.2578	0.0242
16	256	16	240347.12	240234.50	1.0005×	1.39	136	750	5.51×	0.2963	
24	512	12	360472.84	360351.74	1.0003×	1.45	224	1141	5.09×	0.3329	
32	512	16	480590.09	480468.99	1.0003×	1.39	246	1477	6.00×	0.3329	
48	1024	12	720830.96	720703.49	1.0002×	1.45	426	2261	5.31×	0.3699	
64	1024	16	961065.45	960937.98	1.0001×	1.39	464	2921	6.30×	0.3699	
96	2048	12	1441545.64	1441406.98	1.0001×	1.45	831	4501	5.42×	0.4143	
Avg.					1.00033×				5.48×		

TABLE 4: Area, Delay and Power analysis of HU for different configuration of HT

In Table 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.

Our HU can support large hash tables, but there are trade-offs between the size of the HT and the operating frequency of the HU, as well as the power consumption and area overhead. When the HT size increases, the delay of the hash operations increase, due to RC parasitics (as shown in Table 3). Similarly, the power consumption and the area overhead increases, as shown Table 4.

7 Conclusions

In this paper, 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 the HU at the microarchitecture level and at the circuit level, and verified the operation of the HU. We showed that the HU obtains a speedup of up to 15× over the software hash implementation with a reduced cache miss rate. We demonstrated an average power improvement of 5.48× for our HU design compared to a traditional CAM design. We also quantified 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.

References

[1] 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)*, Oct 2016, pp. 412–415.

[2] 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*. ACM, May 2017, pp. 101–106.

[3] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers," *IEEE Transactions on Computers*, vol. 46, no. 12, pp. 1378–1381, Dec 1997.

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

[5] Synopsys, Inc, http://www.synopsys.com.

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

[7] F. Yamaguchi and H. Nishi, "Hardware-based hash functions for network applications," in *2013 19th IEEE International Conference on Networks (ICON)*, Dec 2013, pp. 1–6.

[8] 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*, Oct 2011, pp. 156–166.

[9] 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*, June 2010, pp. 1–6.

[10] A. Satoh, "ASIC hardware implementations for 512-bit hash function Whirlpool," in *2008 IEEE International Symposium on Circuits and Systems*, May 2008, pp. 2917–2920.

[11] 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*, Dec 2015, pp. 1214–1219.

[12] 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)*, Dec 2006, pp. 150–155.

[13] 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, Aug 2016, pp. 58–63.

[14] 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)*, June 2015, pp. 68–71.

[15] 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)*, March 2015, pp. 1–5.

[16] 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)*, June 2016, pp. 146–154.

[17] 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)*, March 2016, pp. 397–406.

[18] 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*, May 2015, pp. 105–112.

[19] 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*, May 2015, pp. 163–163.

[20] 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*, ser. ANCS '09. New York, NY, USA: ACM, 2009, pp. 153–162. [Online]. Available: http://doi.acm.org/10.1145/1882486.1882521

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

[22] L. Chisvin and R. J. Duckworth, "Content-addressable and associative memory: alternatives to the ubiquitous RAM," *IEEE Computer*, vol. 22, no. 7, pp. 51–64, July 1989.

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

[24] 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, no. 4, pp. 712–716, Apr 2001.

[25] 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, 2002, pp. IV–583–IV–586 vol.4.

[26] 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*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

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

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

[29] 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.

[30] 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*, June 2012, pp. 1256–1257.

[31] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, March 2006.

[32] 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)*, March 2016, pp. 974–979.

[33] Intel Corporation, http://www.intel.com.

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

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

[36] Perl, http://www.perl.org.

[37] 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)*, May 2016, pp. 713–718.

[38] NCSU EDA, "Free pdk 45nm," http://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[39] Cadence Design Systems, Inc, http://www.cadence.com.

Abbas A. Fairouz received his BS (in 2006) and MS (in 2011) degrees in Computer Engineering from CE department, Kuwait University. He received his PhD in Computer Engineering from the department of ECE, Texas A&M University. He currently works as an Assistant Professor in the CE department in Kuwait University. His research of interests are in VLSI circuit design, low-power circuit design, computer microarchitecture design, hardware security, design of SFUs in modern microprocessors and coprocessor acceleration units using FPGAs.

Fairouz received several awards including President of Kuwait award for student honors in 2012, and 1st of class 2011 in MS of Computer Engineering from Kuwait University.

Monther Abusultan received his B.Sc. degree with highest honors in CE with a minor in EE (2008) and his M.S. degree (2010) from Montana State University, Bozeman. He received his PhD degree (2017) from Texas A&M University, College Station. He joined Advanced Design team at Intel Oregon (2017). He published in many areas of VLSI including low power FPGA design, GPGPUs, flash-based digital design, logic synthesis and optimization. He received several awards including Alpha Lambda Delta award, MSU (2008), Outstanding Senior Student at ECE, MSU (2008), Undergraduate Scholars fellowship, MSU (2007-2008), PLUS Scholarship, USDOS, (2006-2008).

Viacheslav V. Fedorov received his B.Sc. and M.S. degrees in Applied Math and Physics in 2007 and 2009 respectively, from Moscow Institute for Physics and Technology (MIPT) in Russia. From 2007 to 2010 he was involved in logic design for the MCST-R1000 SPARC processor core. In 2016 he received his PhD degree in Computer Engineering from Texas A&M University. He then joined NXP Semiconductors as a Systems and Architecture engineer. Interests include high-performance memories and interconnects, Linux kernel, virtualization, and low-level performance analysis and debug.

Sunil P. Khatri received his B.Tech (EE) from IIT Kanpur, his MS (ECE) from UT Austin and his PhD (EECS) from UC Berkeley. He currently serves as a Professor in ECE at Texas A&M University. His research areas are VLSI IC/SOC design, hardware and software algorithm acceleration, and interdisciplinary extensions of these topics to other areas such as security and machine intelligence.

Dr. Khatri has over 248 peer reviewed publications. Among these papers, 5 received a best paper award. He has co-authored 9 research monographs and one edited research monograph, 3 book chapters and 6 US Patents. He was invited to serve as a panelist at a conference 7 times, and has presented 2 conference tutorials.