HASHI: An Application-Specific Instruction Set Extension for Hashing

Oliver Arnold, Sebastian Haas, Gerhard Fettweis
Vodafone Chair Mobile Communications Systems
Technische Universität Dresden
Dresden, Germany

Benjamin Schlegel*, Thomas Kissinger, Tomas Karnagel, Wolfgang Lehner
Database Technology Group
Technische Universität Dresden
Dresden, Germany

ABSTRACT

Hashing is one of the most relevant operations within query processing. Almost all core database operators like group-by, selections, or different join implementations rely on highly efficient hash implementations. In this paper, we present a way to significantly improve performance and energy efficiency of hash operations using specialized instruction set extensions for the Tensilica Xtensa LX5 core. To show the applicability of instruction set extensions, we implemented a bit extraction hashing scheme for 32-bit integer keys as well as the *CityHash* function for string values. We identify the individual parts of the algorithms required to be optimized, we describe our hashing-specific instruction set, and finally give a comprehensive experimental evaluation. We observed that the hash implementation using the hashing-specific instruction set (1) is up to two orders of magnitudes faster than the basic core without extensions, (2) exhibits always better performance compared to hand-tuned code running on modern high-end general purpose CPUs, and (3) has a significantly better footprint with respect to energy consumption as well as chip area. Especially the third observation has the potential for a higher packing density and therefore a significantly better overall system performance.

1. INTRODUCTION

Database systems can be optimized in many different directions, while the overall challenges are often optimizations of algorithms or adapting the software to given hardware features like multiple cores, SIMD, and memory hierarchies. Algorithms deployed in database systems are therefore highly tuned and very often either reach the processor's peak performance or they are limited by some system characteristics like memory bandwidth or latency, the number of available cores, or down-scaled core frequencies due to power and heat dissipation limitations.

Therefore, a high performance database system design also depends on hardware specializations to achieve even better performance. Unfortunately, general purpose processors are reaching their limits since single-threaded performance has almost stopped to increase, because the maximum core frequency is limited by physical constraints. Even the current solution, to put more and more homogeneous cores onto a single socket, will also reach physical limitations soon. As the feature size in which processors are manufactured will shrink, the growing number of transistors will increase the occurrence of *dark silicon* [4, 6]. Dark silicon is caused by thermal problems, since supplying all transistors with energy at the same time would overheat the processor. Having an energy-efficient processor design and introducing specialized instruction sets that are usable on-demand, mitigates the impact of dark silicon. Especially the latter idea can be implemented on a fraction of the chip space, where the instruction set extensions can be power-gated whenever needed without compromising the overall general purpose characteristics of the chip itself.

To follow the idea of specialized instruction sets and to push the envelope in database performance, we strive to widen the view towards adjusting processors for today's and future query processing needs. Processors themselves can be extended to allow for a higher query throughput and lower latency by adding specialized instruction sets, optimized for supporting query processing primitives. Special purpose hardware extensions for database operations can improve the performance of the supported algorithms significantly while – at the same time – saving energy in the processor, mitigating the impact of dark silicon and therefore allowing a much higher packing density of individual cores due to good electrical and thermal properties.

In previous work, we proposed an instruction set extension to accelerate set-oriented database primitives [2]. In this paper, we want to go further with this novel way of optimization by providing a specialized instruction set extension for database hashing primitives in combination with a low-energy processor design. Hashing is widely used in modern databases for join implementations, aggregation operators, as well as indexing. To support all these operators with instruction set extensions, we investigate hashing primitives like integer and string hashing, as well as insert and lookup operations. For all these operations, we use well-

*This author works now at Oracle Labs, Belmont, CA, USA.

known and established hashing algorithms and implement them in an instruction set extension. Furthermore, for integer hashing, we decided for an adaptive hashing algorithm where the hash mask can be adjusted to support arbitrary data sets, efficiently decreasing collisions when using hash tables. To further support this adjustment we also investigated a sampling approach, where a subset of the data set is scanned to calculate the mentioned hash mask.

Our key contributions can be summarized as follows:

(1) We introduce the idea to extend customizable processors with specialized instruction set extensions for hashing primitives in database systems.

(2) We discuss a novel instruction set extension to speed up hashing primitives. The instructions can be used, for example, for efficient join or aggregation algorithms as well as indexing. We discuss each instruction in detail with their implementation in hardware.

(3) We provide a broad evaluation of the instruction set extension. Besides a performance comparison, we evaluate the required chip area of the instructions, their impact on the processor's core frequency, and the processor's power consumption. Moreover, we compare the performance of hash algorithms running on our processor with existing, highly-optimized algorithms running on modern x86 processors.

Please note that a system-level evaluation of our instruction set extension is not part of this paper. At this point, we focus on the selection and implementation of low-level hashing algorithms and their performance. This is the first step towards accelerating more complex database operators that rely on efficient hashing operations.

The paper is structured as follows: In Section 2 we give the required background knowledge for our hash algorithms and introduce the model of our customizable processor. Afterwards, we present our hashing specific instruction set extension (Section 3) in detail. In Section 4 finally, we evaluate our instruction set extension with respect to throughput, chip area size, and power consumption. We also compare our results to modern x86 processors. Section 5 states related works and Section 6 summarizes and concludes the paper.

2. BACKGROUND

Before diving into detail, we sketch the necessary background information of our hash algorithms as well as the customizable processor model used for the instruction set extension.

2.1 Hash Algorithms

Many hash functions have been proposed for many different application scenarios respectively key distributions. To cover the most important data types of a DBMS, we decided to pick a configurable hash function for integers as well as the popular *CityHash* function for string keys. In addition to investigating the hardware optimization potential of both hash functions, we also discuss the optimization potential for general insert and lookup operations.

Integer Hash Function and Sampling

The integer hash function we selected is based on bit extraction. The hash function considers only specific bits in the key to calculate the smaller hash key. For example, the input integer key is 32 bit wide and we extract the bits at position 2, 4, 8, and 10 to a four-bit hash key, thus being able to address 16 buckets in a hash table. To implement this hash function, the specific bits need to be extracted from the integer key and are required to be condensed into the smaller hash key. This can be seen in the related C code of Figure 1. *key* points to the memory location of the 32-bit keys. The resulting hash values are written to the memory pointed by *hashValue*. We assume that the hash mask (*hashFunc*) contains at most 16 set bits distributed over all 32 bits. Hence, the size of the hash values do not exceed 16 bits. As depicted in the C code, the bit extraction is implemented by a series of bit wise shifts and logical operations, which consumes a fairly large number of CPU cycles on general purpose processors. Thus, the overall goal of our approach is to implement a specialized instruction that executes this specific operation within a single CPU cycle to speed up hash number computations considerably.

```
void int_hash(
    unsigned int* key,          //input keys
    unsigned short* hashValue,  //output hash values
    int keySize,                //number of keys
    unsigned int hashFunc       //hash mask
){
    int i, j;
    unsigned int hash;
    unsigned int mask = 0xFFFFFFFF, shVal, shVal_neg;

    for(i=0; i<keySize; i++){
        //bit selection
        hash = key[i] & hashFunc;

        //extract bits
        for(j=30; j>=0; j--){
            if(!(hashFunc & (0x1<<j))){
                //partial shift right
                shVal = hash & (mask<<j);
                shVal_neg = hash & ~(mask<<j);
                hash = (shVal>>1) | shVal_neg;
            }
        }

        hashValue[i] = hash;
    }
}
```

Figure 1: Algorithm of Integer Hash Function

In addition to performing this hash function efficiently, we also strive to accelerate the sampling step. A sampling step is required to decide upon the significant bits (described within a corresponding hash mask) to be extracted and used to build the hash key. The quality of the final hash function heavily depends on the proper selection of the hash mask. Thus, a careful choice of a hash mask determines the efficiency of the execution of the hash operations at runtime by minimizing collisions while mapping the input keys to uniformly distributed hash keys. If a set of keys is already given, sampling can be used to investigate a subset of the given input data to estimate the characteristics of the complete data set. Based on this output, the hash function can be configured with the computed hash mask.

Our sampling algorithm therefore determines the distribution of all 32 bits of the sampled data set to determine

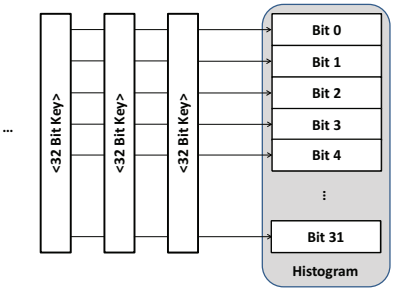


Figure 2: Sampling of 32-bit input keys using a histogram representation

the bits with the highest entropy for the hash function. This is done by counting the number of set bits of the sampled input data set. The result is stored in a histogram as shown in Figure 2. In particular, the chosen hash mask should include the bits, which are set in roughly 50% of the input keys.

String Hash Function

As an example for string hashing we decided to choose the *CityHash* [5] function. Although, the algorithm is quite complex, the provided implementation on general purpose CPUs is highly optimized especially through SSE support. Figure 3 depicts the CityHash32 function. In general, the function takes the pointer to the string and the string length as input.

```
unsigned CityHash32(char *s, int len){
    int hash = comp_1(s+len-20);
    int i = (len-1)/20;

    do {
        hash = comp_2(s, hash);
        s += 20;
    } while(--i != 0);

    return comp_3(hash);
}
```

Figure 3: Algorithm of CityHash32 Hashing

In our example, we assume that the length n is greater than 24. Otherwise, the algorithm is calculating the hash value slightly differently. For readability, we denote with *comp_X* computations consisting of additions, multiplications with constants, bitwise rotations, and bitwise XOR. After a first calculation (comp_1) of the last 20 ASCII characters, the CityHash32 function sequentially loads 20 characters starting from the beginning of the string within a loop. As can be seen in Figure 3, the output of comp_2 is reused in conjunction with 20 new characters within the next calculation. Finally, a last computation (comp_3) returns the final hash value. In Section 3 we describe how to implement specialized instructions to speed up the computations of the algorithm.

Insert and Lookup

To achieve high performance for hash-based database operators, fast insert, and lookup operations are essential. Within our setup, we assume a hash table with 128-bit buckets, i.e., one bucket has four dedicated 32-bit positions. After applying the integer hash function as described before, a 32-bit key is written to the next free position of the bucket. Since the payload is optional (e.g., for simple hash probes) or maybe of a variable size, it is stored separately to preserve the generality of the instruction. As already stated, bucket overflow handling and extensions for wider keys and payloads are preserved for future work.

To lookup a given key in the hash table, the key is – in a first step – hashed by the integer hash function. In a second step, the bucket is searched for the targeted key. If the key is found, the related bucket and the position of the key within the bucket is returned by the function. To accelerate both operations, we aim to add a specialized insert and lookup instruction, which is combined with the integer hash function into a single instruction.

2.2 Processor Model

Within our work, we use a Tensilica Xtensa LX5 customizable processor as a basis for the instruction development. In Figure 4, the main components (white) and possible extensions (colored) are depicted. The LX5 supports a basic RISC instruction set and contains several registers for temporary data storage. These two components are extensible by hash-specific versions. Therefore, the approach and the tool flow presented in [2] are employed. Additionally, hash-specific states are used. In contrast to registers, states are not managed by the compiler but by the newly introduced instructions itself. For instance, data pointers are explicitly incremented by the corresponding instructions.

The processor solely operates on local memories and is consequently facing no cache misses. The processor has one 32 KByte instruction memory and two 32 KByte data memories integrated. Data bus width is 64 and 2x128 bit for the instruction and data memories, respectively. Each data memory bank uses a dedicated load-store unit. Memory access latency is one cycle for all memories. An integrated prefetcher triggers data transfers over the interconnection network to the local memory before it is actually needed using burst mode for fast data transfer; processor and data prefetcher operate simultaneously.

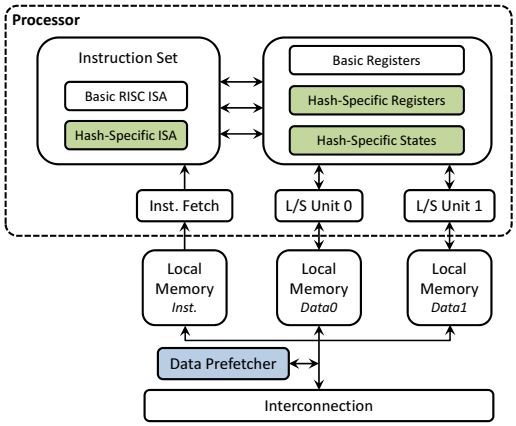


Figure 4: Hash-specific Processor Model

The extended hash-specific instructions are developed using the *Tensilica Instruction Extension* (TIE), a specialized *Hardware Description Language* (HDL). Using this language

and the development environment allows us to easily employ optimization techniques such as SIMD and Very Large Instruction Words (VLIW). This VLIW instruction format is called *Flexible Length Instruction Xtension* (FLIX).

3. INSTRUCTION SET EXTENSIONS

In this section, we present our newly developed hashing-specific instruction set. We implemented the previously outlined hash functions, i.e., the bit extraction and the corresponding sampling, CityHash32, as well as the insert and lookup operations.

3.1 Integer Hash Function and Sampling

As already outlined, the integer hash function performs bit extractions out of a 32-bit key. We choose this bit size, since it is a very conventional format in many programming languages. Theoretical, we have the potential to develop instructions for every arbitrary fixed-width data type.

Figure 5 depicts the processing steps of this integer hash function, starting with the selection of n bits. They are determined by a bit mask, which is derived from the actual hash mask. Up to 16 bits can be selected. In the next step all selected bits are shuffled. Both steps are fully configurable at runtime. The entire hashing operation is executed within a single clock cycle.

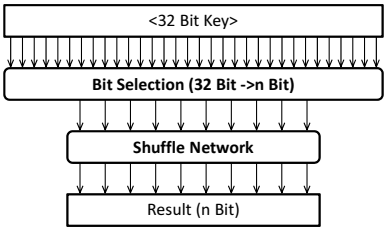


Figure 5: Processing steps of the integer hash operation

To push the performance even more, we extend the instruction to simultaneously execute eight hashing operations (8-fold SIMD). For that purpose additional load instructions are necessary to load data from the local data memory into the internal processor states `Key_0` to `Key_7`. Their interaction is depicted in Figure 6. There is one load operation for each load-store unit (`LSU`). `LD_0` loads up to 128 bits from local memory 0 with `LSU0` and `LD_1` loads up to 128 bits from local memory 1 with `LSU1`. The hash operation (`HOP`) performs the actual hashing on the loaded keys. All results are stored in eight 16 bit states, called `Result_0` to `Result_7`. In the next step, the results are stored to the local memory (`ST`).

In contrast to the pure C code implementation in Figure 1 of section 2.1, all computations are merged into the `HOP` instruction. The now resulting C code is depicted in Figure 7. The input parameters are assigned to internal states of the processor by the instruction `init_states`. This enables a fast access to the memory addresses and the hash mask in one clock cycle. Now, in every loop iteration 16 keys are processed. Hence, the number of iterations of the loop is reduced to a sixteenth. Instructions residing on the same code line are executed simultaneously.

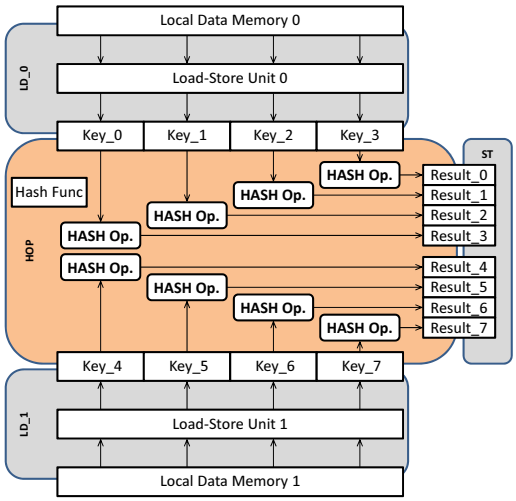


Figure 6: 8-fold SIMD Hash Key Computation

```
void int_hash(
    unsigned int* key,          //input keys
    unsigned short* hashValue,  //output hash values
    int keySize,                //number of keys
    unsigned int hashFunc       //hash mask
){
    int i;
    init_states(key, hashValue, hashFunc);

    LD_0(); LD_1();

    for(i=0; i<(keySize/16); i++){
        LD_0(); LD_1(); HOP();
        LD_0(); LD_1();
        HOP(); ST_0(); ST_1();
    }

    HOP();
    ST_0(); ST_1();
}
```

Figure 7: Integer Hash Function: C code with newly developed instructions

To improve the comprehension of the progression, Figure 8 visualizes the processor's pipeline. The load instructions `LD_0` and `LD_1` are used twice subsequently. Thereby, within the next two clock cycles 16 32-bit keys are loaded from the two local data memories. Two cycles later, `HOP` is performed over the loaded keys. After two or three cycles, `ST_0` and `ST_1` store eight 16-bit hash keys each into the local data memories. Altogether, by using the presented pipelined approach, it consumes three clock cycles for hashing 16 keys. The latency is six cycles.

Sampling uses the same load instructions `LD_0` and `LD_1` as the integer hashing operation explained above. As before, each `LSU` fetches 128 bits from the local data memory. It differs only in the actual operation, which is executed on eight keys in parallel (see Figure 9). Thereby, the bit-values at the same bit-position of each key are added to a dedicated 16 bit histogram accumulator processor state (aggregated bit-weights). Altogether, 32 accumulator states are present – one for each bit position. A configurable increment for each pointer enables the selection of the keys, which shall

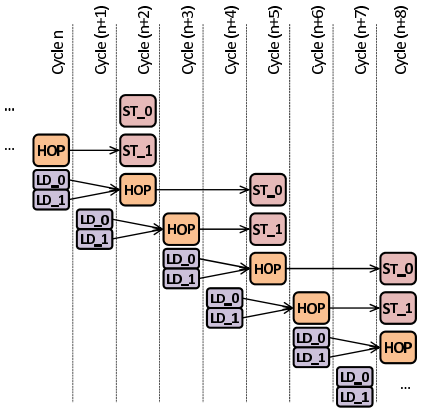


Figure 8: Hashing pipeline snippet and data flow

be included into the data sample. This sampling operation (`SOP`) processes all previously listed actions in one clock cycle. Even this hardware block alone is sufficient to achieve a high sample throughput due to its powerful parallelism. Hence, no further logic is necessary, e.g., to identify dependencies among the keys for speeding up the algorithm. A corollary is that the actual values of the keys are not taken into consideration.

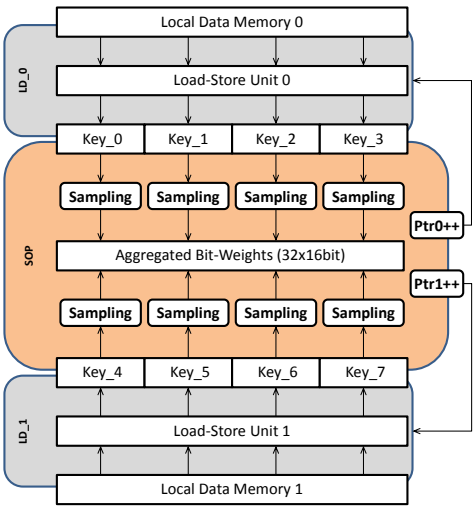


Figure 9: Sampling Instruction

3.2 String Hashing

Due to the high sequential fraction in the CityHash32 function, the algorithm itself can not be fully parallelized. However, the string is distributed over the two local data memories to utilize our two load-store units. That enables us to parallelize load and store instructions. Due to our developed application-specific hardware, multiple operations are merged into one instruction, which is executed within one cycle. The corresponding C code, including the newly developed instructions, is depicted in Figure 10.

The algorithm starts with an initialization of the processor's internal states. For instance, the states contain the address and the length of the string. Afterwards, the operation `CityHash32_load` loads two 128-bit vectors from the

```
unsigned CityHash32(char *s, int len){
    int i;
    init_states(s, len);

    CityHash32_load();
    CityHash32_comp_1();

    for(i=len/20; i>0; i--){
        CityHash32_load();
        CityHash32_comp_2();
    }

    return CityHash32_comp_3();
}
```

Figure 10: CityHash32: C code with newly developed instructions

local data memories in parallel by using `LSU0` and `LSU1`. All following operations perform the same calculations as shown before in Figure 3 of Section 2.1. Again, the internal result and the next 20 characters are used within the next call of the instruction. In contrast to the pure C code, these internal results are stored in states. However, the computations operate on 20 characters (160 bit). Hence, the load instruction holds remaining characters in internal states and performs a first-in-first-out implementation. This ensures the availability of 20 characters in every loop iteration. Finally, `CityHash32_comp_3` executes the last computations and returns the 32-bit hash value.

3.3 Lookup and Insert

The lookup operation is accelerated by searching on eight keys in parallel. The procedure is executed as follows:

1) Load eight 32-bit keys from the local data memories using `LSU0` and `LSU1`, consuming one single clock cycle.
2) Extract the specific bits of those eight keys by applying the 8-fold SIMD hash key computation of Figure 6. Additionally, calculate the addresses of the buckets by using the hash values. Altogether, both steps needs only one additional clock cycle, too.
3) Load the buckets that are corresponding to the hash values. Since the hash table is distributed over the two local memories, the implementation tries to use both load-store units to load two buckets simultaneously. This is only applicable, if the considered buckets are not located within the same local memory. If the buckets are in the same local memory, the needed number of clock cycles changes from four to eight.
4) Compare each key with all four fields of the related bucket. The comparison returns a result containing the hash value and the information of possible matched keys. The results of all eight keys are stored to the local data memory. Again, this consumes two clock cycles.

In summary, to search eight 32-bit keys, our instructions need at least eight clock cycles, but at most twelve cycles. For random input values, the average number of clock cycles is ten.

As well as the Hash + Lookup algorithm, our extended instruction set provides an optimized insert operation, which maps a key into the corresponding hash table. We again use the hash key computation of Figure 6 to parallelize hashing of eight 32-bit keys. Figure 11 exemplarily ties in with the depiction. The shown processing chain uses the eight result-

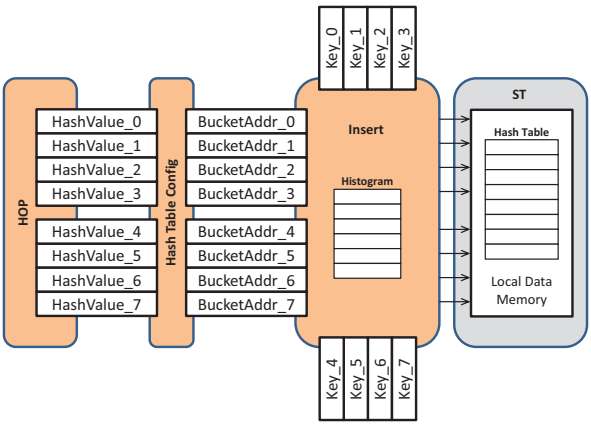


Figure 11: Hash + Insert Instruction

ing hash values and the hash table parameters to compute the addresses of the related buckets. Up to this point, the steps are equal to steps 1) and 2) of the Hash + Lookup algorithm. Then, the eight keys are stored on the next free position of the related bucket. A histogram located in intern states of the processor indicates the inserted keys. That consumes one clock cycle for each key, resulting in eight cycles for all eight keys. In total, we need ten clock cycles to insert eight 32-bit keys into the hash table. Wether we assume a hash function decreasing collisions efficiently, the keys are stored into arbitrary distributed buckets of the local data memory. Thus, we do not gain of our 128-bit memory interface.

4. EXPERIMENTAL RESULTS

In this section, we evaluate our hashing-specific instruction set and compare the results to base line implementations on general purpose architectures.

4.1 Processor Setup

As already mentioned, we use the configurable Tensilica Xtensa LX5 Core, called HASH_RISC, as a foundation of our processor. The processor has similar features like the 108Mini[1]. The 108Mini is a Standard Diamond processor from Tensilica, which forms the initial point in our investigations. In contrast to the two 32 KByte local data stores of the HASH_RISC, the 108Mini includes one 64 KByte data store. Furthermore, the HASH_RISC integrates a 32 KByte local instruction memory. In comparison to the 108Mini, the instruction and data bus width of the processor is increased from 32 to 64 and from 32 to 2x128 bit, respectively. Each data bus can be accessed by a dedicated load-store unit. Our HASHI processor is equivalent to the HASH_RISC core, but includes additionally the newly developed application-specific instructions.

4.2 Performance

In the first set of experiments, we compare the performance of our processors for several hash relevant algorithms. We obtain the throughput T for the integer hash functions and the sampling algorithm by calculating $T = \frac{n_{key}\, f_{max}}{n_{cycle}}$

[1] `http://ip.cadence.com/uploads/pdf/108Mini.pdf` contains more information about this processor.

Benchmark	108Mini	HASH_RISC	HASHI
Frequency [MHz]	442	555	488
Hash + Lookup [MHashs/s]	1.0	2.1	386
Hash + Insert [MHashs/s]	1.1	2.3	389
Hash Keys [MHashs/s]	1.1	2.4	2,533
Hash Sampling [MHashs/s]	2.0	3.4	2,575
CityHash32 [MChars/s]	38.3	64.5	4,770

Table 1: Performance Results

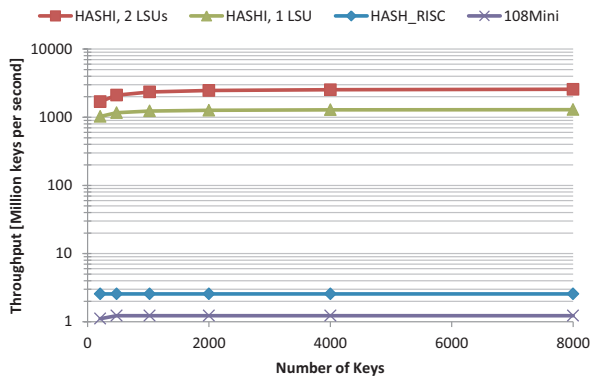
where n_{key} denotes the number of keys, f_{max} is the maximum frequency for each processor, and n_{cycle} means the required number of cycles to process the complete operation. The throughput for CityHash32 is achieved by substituting the number of keys n_{key} of the previous equation by the string length n_{char}, i.e., the number of 8-bit characters. Furthermore, all algorithms are performed with cycle-accurate simulation models of the processors.

Except for the Hash + Lookup benchmark all other algorithms have data-independent processing times on our HASHI. The other evaluation-cores perform slightly different with various input data sets and hash table sizes, respectively. For example, the more bits are set to one in the hash mask, the faster the algorithm has finished. Excluded from all is the CityHash32 algorithm never showing data dependencies on all processors. In summary, all implementations of the algorithms compute their result in expected time $\mathcal{O}(n)$, where n is the number of processed elements.

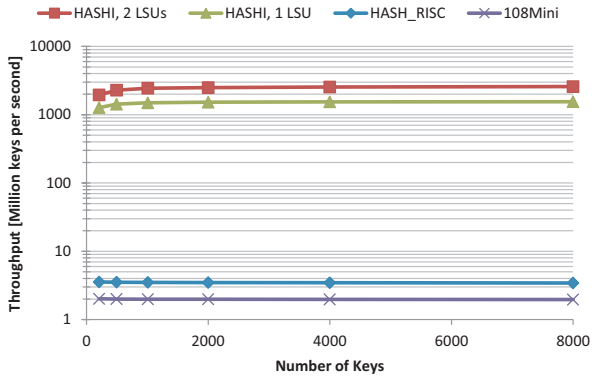
Table 1 provides the clock frequency and the achieved throughput. Due to the data dependency, the Hash + Search algorithm uses randomly distributed values residing in their respective domains. Thereby, we make sure to obtain an average throughput. The input data sets of all other algorithms consist of a linear increasing sequence of numbers. The used hash mask holds ten arbitrary set bits resulting in a hash table size of 1024 buckets. Note that the distribution of a fixed number of bits in the hash mask has no impact on the processing time. The throughputs in Table 1 are obtained with the following set sizes: the Hash + Insert and Hash + Search algorithms use 5600 keys each, Sampling and Parallel Hashing perform on 8000 keys, respectively, and CityHash32 operates on 16000 characters.

In general, the throughput of the HASH_RISC is nearly doubled related to the 108Mini. In the case of the Hash + Insert and the Hash + Lookup algorithm, the average throughput is increased by around 350x (108Mini) and 180x (HASH_RISC) to almost 390 MHashs/s (HASHI). Nevertheless, maximum throughput of Hash + Lookup on the HASHI is 488 MHash/s. The hash key generation of the HASHI is approximately 2,300x and the CityHash32 algorithm is 120x faster than our standard RISC controller 108Mini.

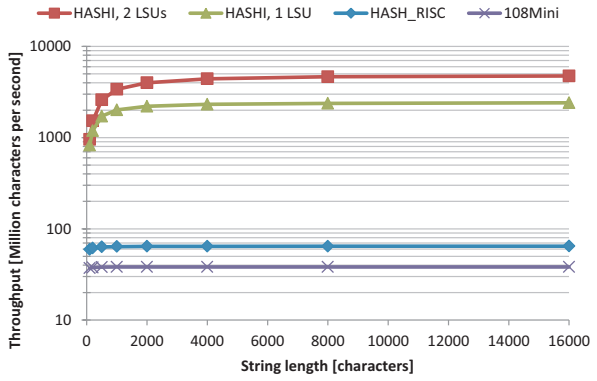
Figure 12 depicts the throughputs of our processor configurations for a various number of keys and string lengths, respectively. The used hash mask and the distribution of the input values are adopted from the previous explanation. Our extended processor HASHI is employed with one or rather



(a) Hash Keys



(b) Sampling



(c) CityHash32

Figure 12: Throughputs by varying input data sizes

two load-store units (LSU). In contrast to the HASHI with one LSU, the throughput of HASHI applying two LSUs is nearly doubled for all three algorithms.

Additionally, the throughput slightly increases while performing on greater input set sizes. Then, the percentage of the initialization part is negligible and the throughput tends to the ideal case. For example, our implementation of the integer hash function consumes three cycles while performing on 16 keys per loop iteration (cf. Figure 7 of section 3.1). This leads to a maximum theoretical throughput of 2,603 MHashs/s for HASHI at 488 MHz. As already discovered in Table 1, 108Mini and HASH_RISC always

Process	Processor	A_{LOGIC} [mm^2]	A_{MEM} [mm^2]	f_{MAX} [MHz]	P[mW] @f_{MAX}
65 nm	108Mini	0.220[1]	-	442[1]	27.4[1]
	HASH_RISC	0.164	0.874	555	63.2
	HASHI	0.731	0.874	488	138.4
28 nm	HASHI	0.214	0.213	500	-

[1] `http://ip.cadence.com/uploads/pdf/108Mini.pdf`

Table 2: Synthesis Results

exhibit the lowest throughputs. Due to the pure C code implementations, this throughput is not dependent of the input set sizes.

4.3 Area, Timing and Power Consumption

In the next set of experiments, we compare the maximum frequency, area, and power consumption of the different processor configurations. All versions of the processors, including the memories, have been synthesized with Synopsys Design Compiler for a 65 nm low-power TSMC process using typical case conditions (25°C, 1.25 V). Synopsys PrimeTime estimation tool is used to obtain the power consumption of the processor configurations. Table 2 provides the measurement results for all three processor configurations. Due to the integration of the newly developed instructions, the overall area of the basic core HASH_RISC is increased from 1.038 mm^2 to 1.605 mm^2 for the HASHI processor. These areas include 0.874 mm^2 each for the local memories. Furthermore, the maximum clock frequency of the HASH_RISC core is slightly decreased from 555 MHz to 488 MHz (HASHI). Power consumption of HASH_RISC core is increased from 63.2 mW to 138.4 mW for the HASHI processor (including memories). The same tool flow is used with a 28 nm super low-power (SLP) Global Foundries process, including super low-voltage (SLVT) parameters. Typical case conditions are applied (25°C, 1.0 V). Due to the improvements in fabrication technology, the area of the HASHI processor is decreased to 0.427 mm^2.

Table 3 provides the relative area for the newly developed instructions for the 65 nm low-power TSMC process. Major parts of the area are consumed by Hash + Insert and the CityHash32 instructions. Especially, due to the processor intern histogram, the Hash + Insert algorithm consumes much area. Similarly, additional 32 bit multipliers are required by CityHash32, inflating the area consumption. The area part ALL contains general load instructions with dedicated states that are used by each algorithm.

Part	Area[%]
Basic Core	18.4
Hash + Lookup	4.4
Hash + Insert	26.9
Hash Keys	5.7
Hash Sampling	13.3
CityHash32	25.8
ALL	5.5
SUM	100

Table 3: Relative area consumption per newly introduced instruction (HASHI processor)

4.4 Comparison with Other Architectures

Within the last set of experiments, we compare the performance of algorithms using our instruction set extension with existing highly-optimized algorithms running on general purpose CPU architectures. Table 4 compares the HASHI with an Intel i7-4550U[2], based on the Haswell architecture, and an Intel i7-3960X[3], based on the Sandy-Bridge architecture. The input data is placed in the L1 cache before starting the experiments. Despite the advantage in fabrication technology and clock frequency of the Intel cores the HASHI outperforms the i7-4550U as well as the i7-3960X processor. The hash key generation of the HASHI is 120x and 170x faster than the i7-4550U and i7-3960X, respectively. However, Intel introduced the AVX2-PEXT instruction in their new architectures, speeding up hash key generation significantly. There, HASHI shows only a slight speedup.

In the case of the sampling algorithm, HASHI reveals our advantage of newly developed instructions. Due to a grand data level parallelism as explained in section 3.1, we achieve a high speedup of around 180x compared to the Intel processors.

Note that all processors have similar throughputs for the CityHash32 algorithm. The reason for that is that on the one hand, the algorithm is well suited for a cache-based architecture due to many predictable operations. On the other hand, the algorithm cannot be parallelized by SIMD, because of a long sequential execution path. Hence, the HASHI has not the ability to deploy all its optimization techniques thoroughly. Nevertheless, it is also important to mention that the Intel i7-4550U has an advanced fabrication technology and its clock frequency is over 6x higher.

The HASHI keeps up with modern general purpose processors and further has a clear advantage regarding area as well as power consumption. It consumes less than one percent of the area and only two percent of the power compared to the Intel i7-4550U processor. Compared with the Intel i7-3960X it is even better.

	HASHI	Intel i7-4550U (Haswell)	Intel i7-3960X (Sandy-Bridge)
Technology [nm]	65	22	32
Frequency [GHz]	0.488	1.5 (3.0[3])	3.3 (3.9[3])
Power Consumption [W]	0.138	7.9[1]	24.3[1]
Area [mm^2]	1.605	181	434.7
Hash Keys [MHashs/s]	2,533	20.6 2,063[2]	14.9
Sampling [MHashs/s]	2,575	14.5	14.0
CityHash32 [MChars/s]	4,770	4,720	3,678

[1]Measured with RAPL Counter (only core power: PP0)
[2]w/ Intel AVX2-PEXT instruction
[3]Max turbo frequency

Table 4: Performance, Power Consumption and Area Comparison

5. RELATED WORK

Modern general-purpose CPUs already implement a rich set of instruction set extensions (e.g., MMX, SSE, AVX, and AES). Those instruction set extensions are usually designed for a wide field of different applications and thus do not primarily target the acceleration of database systems. Nevertheless, those instructions became attractive when in-memory DBMSs became more and more popular, because the processing bottleneck moved closer to the computing power of the CPU. Thus, several previous works focused on leveraging general-purpose instruction sets for database operations like compression [10, 12], tree traversal [8], or hashing [11]. In our specific scenario, the PEXT instruction [7] of the AVX2 extensions allows the execution of bit-extraction-based hashing within a single cycle. However, this instruction neither implements our SIMD optimizations to increase its throughput nor does it execute as energy-efficient as our instruction as proved by our evaluation. To summarize, all these algorithms only depend on a small fraction of the implemented instruction space of general-purpose CPUs and thus a lot of chip space is wasted for instructions that are not beneficial for the database system. In contrast, our approach of using a customizable processor, allows the directed construction of comprehensive instruction sets that aim primarily at database operators. Additionally, the extensible processor model enables us to add additional load-store units to the core, to find a trade-off between memory bandwidth and computing power.

Efficient hashing has been researched for the last 40 years. However, only a few works concentrate on hardware-supported hashing acceleration. These works mostly focus on optimizing the algorithm for a given platform. On CPUs, parallelism and the memory hierarchies are mainly exploited [3, 13, 9] through best-effort partitioning and optimal hash table sizes. GPUs offer a much higher degree of parallelism, which was investigated in the context of hashing [1]. However, GPUs are not suitable for data-intensive applications, because of the bandwidth limitation between system memory and the dedicated GPU memory. Moreover, this approach faces severe scalability limitations, because of the low energy efficiency of general purpose CPUs and GPUs.

6. CONCLUSIONS

In this paper, we improved the performance of hashing algorithms by a newly developed hashing-specific instruction set. Performance, area- and energy-efficiency is significantly increased compared to modern general purpose processors. We attached the hashing-specific instruction set to a customizable processor. To show the applicability of our instruction set extensions, we implemented a bit extraction hashing scheme for 32-bit integer keys and the corresponding sampling as well as the CityHash32 function for string values. For instance, the sampling is 178x and 184x faster compared to the Intel i7-4550U and the Intel i7-3960X, respectively. Moreover, the hash key computation is 170x faster than the Intel i7-3960X. We showed that area- and energy-efficiency is improved by several orders of magnitude allowing a higher packing density and thus increased scalability of the hardware in terms of parallelism.

[2]`http://ark.intel.com/products/75112`
[3]`http://ark.intel.com/products/63696`

7. ACKNOWLEDGEMENTS

This work has been supported by the state of Saxony under grant of ESF 100098198 (IMData) and 100111037 (SREX) and the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

Furthermore, we would like to thank Synopsys and Tensilica for sponsoring software and IP.

8. REFERENCES

[1] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5), 2009.

[2] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner. An application-specific instructions set for accelerating set-oriented database primitives. In *SIGMOD*, 2014.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[5] Google Inc. CityHash v1.1.1. `http://code.google.com/p/cityhash/`, June 2013.

[6] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 31(4), 2011.

[7] Intel Corp. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, March 2014.

[8] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, 2010.

[9] H. Lang, V. Leis, M.-C. Albutiu, T. Neumann, and A. Kemper. Massively Parallel NUMA-aware Hash Joins. In *INDM*, 2013.

[10] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 2013.

[11] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, 2007.

[12] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2009.

[13] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. *DaMoN*, 2006.