HASHI: An Application-Specific Instruction Set Extension for Hashing

Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, Tomas Karnagel, Wolfgang Lehner

Technische Universität Dresden
Dresden, Germany
Motivation

Today’s Database Systems
- Fat Cores (area & power)
- Few HW adaptions
- CMOS Scaling

Database Processors
- Processors build from scratch
- Long development cycle
- High development costs

Our Approach
- HW/SW codesign
- Customizable processor
- Hashing-specific ISA extensions
- Tool flow → short HW development cycles
Application Scenario 1: Integer Hash Function

- **Bit Extraction**
  Selection of specific bits in a 32-bit key via arbitrary hash mask

- **Sampling**
  Scanning a subset of the data set to choose the most efficient hash mask
Application Scenario 2

- **CityHash32**
  - Non-cryptographic hash function for strings
  - Returns 32-bit hash value

```c
unsigned int 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);
}
```

- **Hash Table Operators (Insert, Lookup)**
  - Operate on 32-bit keys
  - Apply integer hash function
Customizable Processor Model

**Basic Core:** Tensilica LX5
unsigned int hash, shVal, shVal_neg;
unsigned int mask = 0xFFFFFFFF;

for(i=0; i<keySize; i++){
    //load key, 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;
        }
    }

    //store hash value
    hashValue[i] = hash;
}

Pure C code
**Integer Hash Function: C code**

```c
unsigned int hash, shVal, shVal_neg;
unsigned int mask = 0xFFFFFFFF;

for(i=0; i<keySize; i++){
    //load key, 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;
        }
    }

    //store hash value
    hashValue[i] = hash;
}
```

**Pure C code**

```c
//init pointer, variables
init_states(key, hashValue, hashFunc);

LD_0(); LD_1();

//load keys, extract bits, store hash values
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();
```

**C code with new instructions**
Integer Hash Function: C code

unsigned int hash, shVal, shVal_neg;
unsigned int mask = 0xFFFFFFFF;

for(i=0; i<keySize; i++){
    //load key, 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;
        }
    }
    //store hash value
    hashValue[i] = hash;
}

//init pointer, variables
init_states(key, hashValue, hashFunc);

LD_0(); LD_1();

//load keys, extract bits, store hash values
for(i=0; i<(keySize/16); i++){
    LD_0(); LD_1(); HOP(); 1 cycle
    LD_0(); LD_1();
    HOP(); ST_0(); ST_1(); 1 cycle
}

HOP();
ST_0(); ST_1();

Pure C code

C code with new instructions
Integer Hash Function: ISA Extensions
Integer Hash Function: Pipeline Snippet

Latency: 6 cycles
Integer Hash Function: Throughput

Throughput  \( T = \frac{n_{\text{key}}}{t} \)

- \( n_{\text{key}} \): number of keys
- \( t \): time to perform the operation

TU Dresden
## Results: Throughput

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>108MINI</th>
<th>HASH_RISC</th>
<th>HASHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency [MHz]</td>
<td>442</td>
<td>555</td>
<td>488</td>
</tr>
<tr>
<td>Hash + Lookup [MHashes/s]</td>
<td>1.0</td>
<td>2.1</td>
<td>386</td>
</tr>
<tr>
<td>Hash + Insert [MHashes/s]</td>
<td>1.1</td>
<td>2.3</td>
<td>389</td>
</tr>
<tr>
<td>Hash Keys [MHashes/s]</td>
<td>1.1</td>
<td>2.4</td>
<td>2,533</td>
</tr>
<tr>
<td>Hash Sampling [MHashes/s]</td>
<td>2.0</td>
<td>3.4</td>
<td>2,575</td>
</tr>
<tr>
<td>CityHash32 [MChars/s]</td>
<td>38.3</td>
<td>64.5</td>
<td>4,770</td>
</tr>
</tbody>
</table>

**Speedup:**
- HashI vs. 108Mini: 386x
- HashI vs. HASH_RISC: 354x
- HashI vs. Final processor: 2303x
- HashI vs. CityHash32: 1288x
- HashI vs. CityHash32: 125x
# Results: Timing and Area

<table>
<thead>
<tr>
<th>Process</th>
<th>Processor</th>
<th>$A_{\text{LOGIC}}$ [mm$^2$]</th>
<th>$A_{\text{MEM}}$ [mm$^2$]</th>
<th>$f_{\text{MAX}}$ [MHz]</th>
<th>$P$ [mW] @ $f_{\text{MAX}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 nm</td>
<td>108MINI</td>
<td>0.220$^1$</td>
<td>-</td>
<td>442$^1$</td>
<td>27.4$^1$</td>
</tr>
<tr>
<td></td>
<td>HASH_RISC</td>
<td>0.164</td>
<td>0.874</td>
<td>555</td>
<td>63.2</td>
</tr>
<tr>
<td></td>
<td>HASHI</td>
<td>0.731</td>
<td>0.874</td>
<td>488</td>
<td>138.4</td>
</tr>
<tr>
<td>28 nm</td>
<td>HASHI</td>
<td>0.214</td>
<td>0.213</td>
<td>500</td>
<td>-</td>
</tr>
</tbody>
</table>


## Final processor

- **HASHI**

## Relative Area Consumption (HASHI)

<table>
<thead>
<tr>
<th>Part</th>
<th>Area [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Core</td>
<td>18.4</td>
</tr>
<tr>
<td>Hash + Lookup</td>
<td>4.4</td>
</tr>
<tr>
<td>Hash + Insert</td>
<td>26.9</td>
</tr>
<tr>
<td>Hash Keys</td>
<td>5.7</td>
</tr>
<tr>
<td>Hash Sampling</td>
<td>13.3</td>
</tr>
<tr>
<td>CityHash32</td>
<td>25.8</td>
</tr>
<tr>
<td>ALL</td>
<td>5.5</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>
# Results: Comparison

<table>
<thead>
<tr>
<th>Measures: HASHI vs. INTEL</th>
<th>HASHI</th>
<th>INTEL 17-4550U (HASWELL)</th>
<th>INTEL 17-3960X (SANDY-BRIDGE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology [nm]</td>
<td>65</td>
<td>22</td>
<td>32</td>
</tr>
<tr>
<td>Frequency [GHz]</td>
<td>0.488</td>
<td>1.5</td>
<td>3.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(3.0&lt;sup&gt;3&lt;/sup&gt;)</td>
<td>(3.9&lt;sup&gt;3&lt;/sup&gt;)</td>
</tr>
<tr>
<td>Power Consumption [W]</td>
<td>0.138</td>
<td>7.9&lt;sup&gt;1&lt;/sup&gt;</td>
<td>24.3&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>Area [mm&lt;sup&gt;2&lt;/sup&gt;]</td>
<td>1.605</td>
<td>181</td>
<td>434.7</td>
</tr>
<tr>
<td>Hash Keys [MHashes/s]</td>
<td>2,533</td>
<td>20.6</td>
<td>14.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2,063&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Sampling [MHashes/s]</td>
<td>2,575</td>
<td>14.5</td>
<td>14.0</td>
</tr>
<tr>
<td>CityHash32 [MChars/s]</td>
<td>4,770</td>
<td>4,720</td>
<td>3,678</td>
</tr>
</tbody>
</table>

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

- Hardware/Software Codesign approach

- Results
  - High database throughput
  - Highly reduced area and power consumption
  - 170x less energy consumption than a high-end x86 processor (@ same performance)

- Silicon Prototype
  - Tape-out April 2014
  - 28 nm LP process: Globalfoundries
  - ISA: Hash Functions, Hash Table Operators etc.

1 Nöthen et al., A 105GOPS 36mm\textsuperscript{2} Heterogeneous SDR MPSoC with Energy-Aware Dynamic Scheduling and Iterative Detection-Decoding for 4G in 65nm CMOS, ISSCC. 2014