Overview on Hardware Optimizations for Database Engines

Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, Wolfgang Lehner

BTW 2017, Stuttgart, Germany, 2017-03-09
Interaction DB-Engine and Hardware

Applications/Database Engines

Well-Known Challenge:
Exploit hardware technology by specific data management techniques (indexing, data storage, query & transaction processing)

Modern Hardware

Main Memory

CPU

<table>
<thead>
<tr>
<th>Year</th>
<th>Memory (KByte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>10</td>
</tr>
<tr>
<td>1980</td>
<td>1e+05</td>
</tr>
<tr>
<td>1990</td>
<td>1e+07</td>
</tr>
<tr>
<td>2000</td>
<td>1e+09</td>
</tr>
<tr>
<td>2010</td>
<td>1e+11</td>
</tr>
<tr>
<td>2020</td>
<td>1e+13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Year</th>
<th>#cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970</td>
<td>2</td>
</tr>
<tr>
<td>1980</td>
<td>4</td>
</tr>
<tr>
<td>1990</td>
<td>8</td>
</tr>
<tr>
<td>2000</td>
<td>10</td>
</tr>
<tr>
<td>2010</td>
<td>14</td>
</tr>
<tr>
<td>2020</td>
<td>18</td>
</tr>
</tbody>
</table>
Era of Dark Silicon

**Moore’s Law**

- Number of transistors in a dense integrated circuit doubles approximately every two years.

**Dark Silicon**

- We can no longer power the transistors that Moore is giving us.

http://engineering.nyu.edu/garg/node/31
HW/SW Co-Design for DB-Engines

Applications/Database Engines

Challenge:
HW/SW Co-Design for Database Engines
Specialization of Hardware to overcome Dark Silicon

Modern Hardware
Outline

HARDWARE FOUNDATION

EXTENSIONS FOR PROCESSING ELEMENTS

INTELLIGENT DMA CONTROLLER
Hardware Foundation

**Tomahawk Platform**
Hardware Foundation – Zoom In

Control Flow

Task Description

Dynamic out-of-order task dispatching
Hardware Foundation – Zoom In (2)

**CORE MANAGER (CM)**
- Extended Xtensa-LX5 from Tensilica (now Cadence)
- 32KB for code
- 64KB for data

**PROCESSING ELEMENTS (PE)**
- Xtensa-LX5 from Tensilica (now Cadence)
- 32KB for code
- 2x32KB for data on PE

**APPLICATION CORE (APP)**
- 570T core from Tensilica (now Cadence)
PART I: EXTENSIONS OF PROCESSING ELEMENTS
Development Flow

Development of Instruction Set Extensions with Tensilica Tools

- Tensilica Instruction Extension (TIE) language
- C/TIE compiler
- Cycle accurate simulator/debugger
- Processor generator

Synthesis of RTL code

- Synopsys Design Compiler, PrimeTime PX
- TSMC CMOS LP 65nm libraries
Investigated Database Primitives

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Bitmap Compression and Processing (AND, OR, XOR)</th>
<th>Hashing</th>
<th>Sorted Set Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLWAH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPAX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash + Lookup</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash + Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Keys</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Sampling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CityHash32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intersection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Union</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Difference</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort-Merge Join</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort-Merge Aggregation (SUM)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ASAP 2016
The 27th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors

CityHash32
Merge Sort
Intersection
Union
Difference
Sort-Merge Join
Sort-Merge Aggregation (SUM)
General Approach for all Extensions

Extended Tensilica LX5 Processor

Instruction Set
- Basic RISC Instruction Set
- Application-Specific Instruction Set

Register Files
- Basic Registers
- Application-Specific Registers
- Application-Specific States

Instruction fetch
- 64 bit

Load-Store Unit 0
- 128 bit

Load-Store Unit 1
- 128 bit

Local Instruction Memory
- Local Data Memory 0
- Local Data Memory 1

Data Prefetcher

Interconnect
**Bitmap Primitives**

**Bitmaps are a Special Kind of Index**

Table $T$

<table>
<thead>
<tr>
<th>OID</th>
<th>$X$</th>
<th>Bitmap Index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$=0$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Bit-wise OR

select * from $T$ where $X < 2$

**Bitmaps Compression**

- bit length equals number of tuples

**Word-Aligned Hybrid (WAH) Code**

- Stateless compression
- Run-length-Encoding (RLE)
  - run of 0’s and 1’s
- WAH bitmaps contain RLE
  - compressed fills and
  - uncompressed literals
Bit-Wise OR on Compressed Bitmaps

32 bit words

In hex

<table>
<thead>
<tr>
<th>b1</th>
<th>40000380</th>
<th>00000000</th>
<th>00000000</th>
<th>001FFFFF</th>
<th>...</th>
</tr>
</thead>
</table>

**WAH b1**

| 40000380 | 8000002 | 00000000 | 001FFFFF |

**Bit-wise OR**

10<runlength>

**WAH b2**

| C0000002 | 7C0001E0 | 3FE00000 |

**b2**

| 7FFFEFFFFF | 7FFFEFFFFF | 7C0001E0 | 3FE00000 | ... |

**Logical operations (AND, OR, XOR)**

1) Load WAH word(s)
2) Calculate output (Fill-Fill, Literal-Fill, Literal-Literal)
3) Combine output
WHILE(XIdx=Xsize && YIdx=Ysize) {
    //new X or Y? Calculate new fill count …

    if(XisFill==1 && YisFill==1) { //2 fills
        if(XfillWords<YfillWords)
            min=XfillWords;
        else
            min=YfillWords;
        writeFill(comprResultBI, &Zidx, X[Xidx]|Y[Yidx], min);
        XfillWords=min;
        YfillWords=min;
    }
    else if((XisFill==1) || (XisFill==0 && YisFill==1)) {
        if(XisFill==1){
            XfillWords--;
            if((X[Xidx]|0xC0000000)==0xC0000000)
                writeFill(comprResultBI, &Zidx, 0xC0000000, 1);
            else {
                comprResultBI[Zidx]=Y[Yidx];
                Zidx++;
            }
        }
        if(YisFill==1){
            YfillWords--;
            if((Y[Yidx]|0xC0000000)==0xC0000000)
                writeFill(comprResultBI, &Zidx, 0xC0000000, 1);
            else {
                comprResultBI[Zidx]=X[Xidx];
                Zidx++;
            }
        }
    }
    else {
        result=X[Xidx]|Y[Yidx];
        if((result&0x7FFFFFFF)==0x7FFFFFFF)
            writeFill(comprResultBI, &Zidx, 0xC0000000, 1);
        else if((result&0x7FFFFFFF)==0)
            writeFill(comprResultBI, &Zidx, 0x80000000, 1);
        else {
            comprResultBI[Zidx]=X[Xidx]|Y[Yidx];
            Zidx++;
        }
    }
}
Processing with PE Extension

Initial Load
Memory 0
Memory 1

Load
Application specific states

Preprocessing
Operation
Postprocessing

Prepare Store
Application specific states

Store
Memory 0
Memory 1

Memory 0
40000380
80000002
001FFFFF
0000000F

Memory 1
C0000002
7C0001E0
3FE00000
00000003

Align to 128-bit lines

ldXstream()

ldYstream()

Proceed to next word (4x)

4 x WAHinst()

Write to output stream
-> append or overwrite previous word with increased fill counter

Buffer result

11001110
00000000
00000000
10000000
00101010
11000001
00000010
01110111
00000000
00000000
00000000

Is word fill or Literal?
-> fill -> overwrite input words

00000000...00000000...00000000...

00000000...00000000...00000000...

11111111...11111111...11111111...11111111...

Perform operation OR

00000000...  v  11111111...  =>  11111111...

01101010...01101010...01101010...01101010...
### Bit-Wise OR on Compressed Bitmaps

32 bit words in hex

<table>
<thead>
<tr>
<th></th>
<th>b1</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>40000380 00000000 00000000 001FFFFF</td>
<td>7FFFFFFF 7FFFFFFF 7C0001E0 3FE00000</td>
</tr>
<tr>
<td>WAH</td>
<td>Literal</td>
<td>0 fill</td>
</tr>
<tr>
<td>b1</td>
<td>40000380 8000002</td>
<td>001FFFFF</td>
</tr>
<tr>
<td>WAH</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>C0000002</td>
<td>7C0001E0 3FE000000</td>
<td>3FE000000</td>
</tr>
<tr>
<td>1 fill</td>
<td>Literal</td>
<td>Literal</td>
</tr>
</tbody>
</table>

**Code with Extension**

```java
do{
    ldXstream();
    ldYstream();
    WAHinst();
    WAHinst();
    WAHinst();
} while(WAHinst());
```
## Many More Extensions

<table>
<thead>
<tr>
<th>Extension</th>
<th>Bitmap Compression and Processing (AND, OR, XOR)</th>
<th>Hashing</th>
<th>Sorted Set Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitiX</td>
<td>WAH, PLWAH, COMPAX</td>
<td>Hash + Lookup, Hash + Insert, Hash Keys, Hash Sampling, CityHash32</td>
<td>Merge Sort, Intersection, Union, Difference, Sort-Merge Join, Sort-Merge Aggregation (SUM)</td>
</tr>
<tr>
<td>HASHI</td>
<td></td>
<td>X, X, X, X</td>
<td></td>
</tr>
<tr>
<td>Tomahawk DBA</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Bitmap Compression and Processing (AND, OR, XOR)
- WAH
- PLWAH
- COMPAX

### Hashing
- Hash + Lookup
- Hash + Insert
- Hash Keys
- Hash Sampling
- CityHash32

### Sorted Set Operations
- Merge Sort
- Intersection
- Union
- Difference
- Sort-Merge Join
- Sort-Merge Aggregation (SUM)
**Evaluation**

**Reference Processors**

- Tomahawk DBA Processor --> Set of different DB-Extensions for WAH-Compression, Hashing, and Sorted-Set Operations

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
<th>Technology [nm]</th>
<th>$A_{\text{total}}$ [mm$^2$]</th>
<th>$f_{\text{MAX}}$ [GHz]</th>
<th>$P_{\text{MAX}}$ [W] @ $f_{\text{MAX}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tomahawk without DBA</td>
<td>Basic Xtensa LX5 without instruction set extensions, 1 LSU, 32-bit memory interface</td>
<td>28</td>
<td>15.92</td>
<td>0.555</td>
<td>0.7</td>
</tr>
<tr>
<td>Tomahawk with DBA</td>
<td>Set of different DB-Extensions for WAH-Compression, Hashing and Sorted-Set Operations</td>
<td>28</td>
<td>18</td>
<td>0.5</td>
<td>0.753</td>
</tr>
</tbody>
</table>

**Comparison**

<table>
<thead>
<tr>
<th>Processor</th>
<th>Description</th>
<th>Technology [nm]</th>
<th>$A_{\text{total}}$ [mm$^2$]</th>
<th>$f_{\text{MAX}}$ [GHz]</th>
<th>$P_{\text{MAX}}$ [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel i7-6500U</td>
<td>Low-power Intel 2-core processor based on Skylake architecture, 4MB L3 cache</td>
<td>14</td>
<td>99*</td>
<td>3.1</td>
<td>25</td>
</tr>
</tbody>
</table>
Evaluation - Bitmaps

![Execution time comparison for different operations and processors](image-url)
Outline

Part 2: Intelligent DMA Controller
Problem Statement

Problem:
Many round-trips for key lookups

Approach:
“Teach B-trees to the memory controller”
Intelligent Main Memory Controller (iDMA)

Vision (and first simulations)
- Intelligent memory controller
- Is aware of the semantics of memory layout
- Implements core operations (e.g. lookup)

Implementation (no yet in silicon)
- 0.183mm² PE with 200Mhz
First iDMA Design
Evaluation using Simulator

(a) Execution times in number of clocks of iDMA with pointer chaser and a processor with an SDRAM

(b) Percentage area fraction of the submodules of the pointer chaser component
Summary

**Hardware Foundation**

- **PE**
  - RISC
  - DMA

- **R**
  - RISC
  - DMA

- **FPGA-Interface**
  - **PE**
    - RISC
    - DMA

**Intelligent DMA Controller**

**Extensions For Processing Elements**

![Graph showing performance comparison between different technologies](image)
Overview on Hardware Optimizations for Database Engines

Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Eric Mier, Gerhard Fettweis, Wolfgang Lehner