

# Data Processing on Modern Hardware

Jens Teubner, TU Dortmund, DBIS Group  
`jens.teubner@cs.tu-dortmund.de`

Summer 2014

## Part VII

# FPGAs for Data Processing

Modern hardware features a number of “speed-up tricks”:

- caches,
- instruction scheduling (out-of-order exec., branch prediction, ...),
- parallelism (SIMD, multi-core),
- throughput-oriented designs (GPUs).

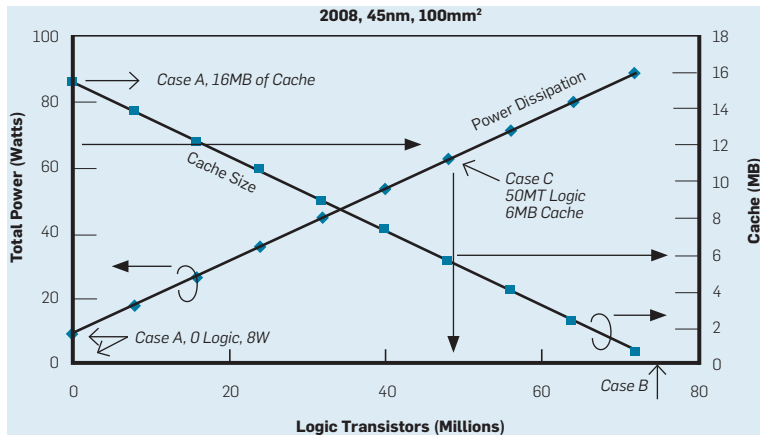
Combining these “tricks” is essentially an **economic choice**:

→ chip space  $\equiv$  €€€

→ chip space  $\leftrightarrow$  component selection  $\leftrightarrow$  workload

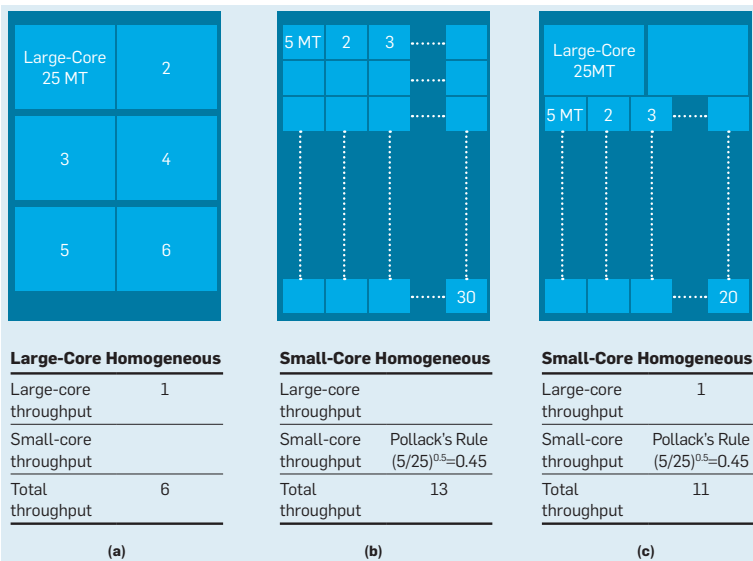
# Another Constraint: Power

- Can use transistors for either logic or caches.



→ Power consumptions limits amount of logic that can be put on chip.

# Heterogeneous Hardware



**Field-Programmable Gate Arrays (FPGAs)** are yet-another point in the design space.

- “Programmable hardware.”
- Make (some) design decisions **after** chip fabrication.

**Promises** of FPGA technology:

- ~> Build application-/workload-specific circuit.
- ~> Spend chip space only on functionality that you really need.
- ~> Tune for throughput, latency, energy consumption, . . .
- ~> Overcome limits of general-purpose hardware with regard to task at hand (e.g., I/O limits).

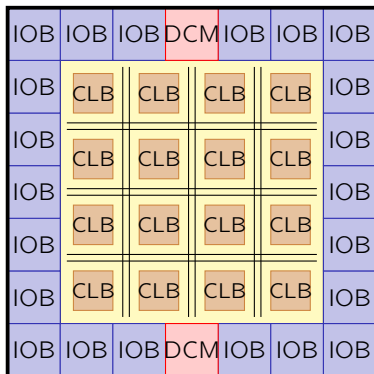
# Field-Programmable Gate Arrays



- An **array** of logic **gates**
- Functionality fully **programmable**
- Re-programmable after deployment (“in the **field**”)  
→ “programmable hardware”

- FPGAs can be configured to implement **any** logic circuit.
- Complexity bound by available **chip space**.  
→ Obviously, the effective chip space is less than in custom-fabricated chips (ASICs).

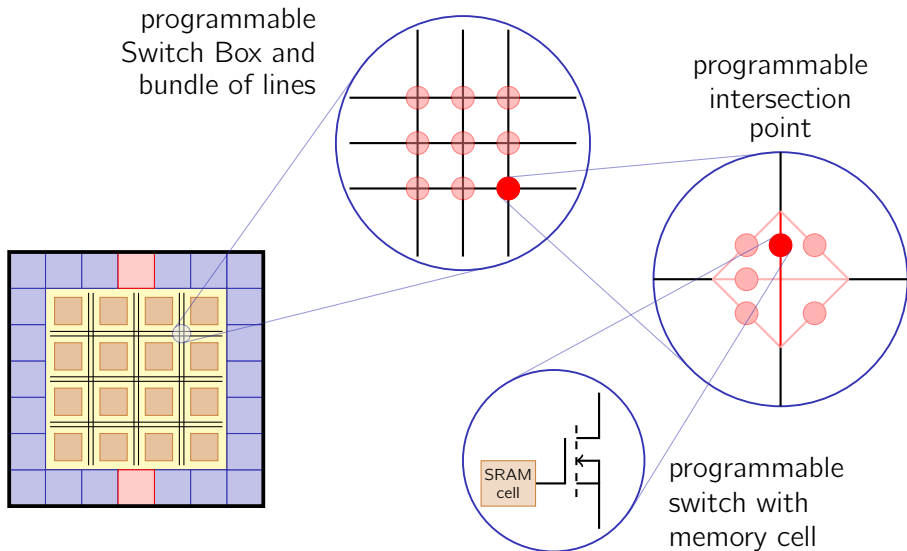
# Basic FPGA Architecture



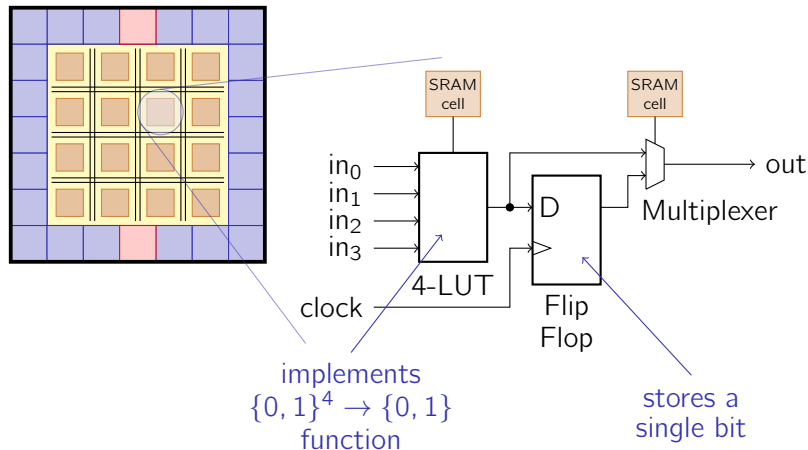
- chip layout: 2D array
- Components
  - CLB: Configurable Logic Block (“logic gates”)
  - IOB: Input/Output Block
  - DCM: Digital Clock Manager
- Interconnect Network
  - signal lines
  - configurable switch boxes



# Signal Routing



# Configurable Logic Block (CLB)

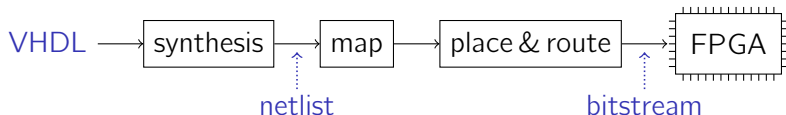


# Programming FPGAs

Programming is usually done using a **hardware description language**.

- E.g., **VHDL**<sup>6</sup>, Verilog
- High-level circuit description

Circuit description is compiled into a **bitstream**, then loaded into SRAM cells on the FPGA:



---

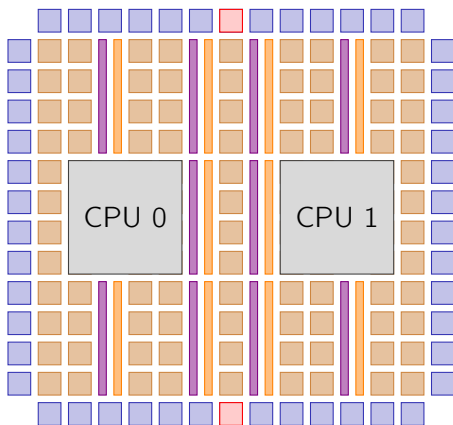
<sup>6</sup>VHSIC Hardware Description language

## Example: VHDL

HDLs enable programming language-like descriptions of hardware circuits.

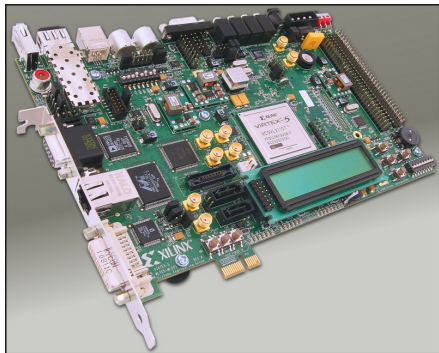
```
architecture Behavioral of compare is
begin
    process (A, B)
    begin
        if ( A = B ) then
            C <= '1';
        else
            C <= '0';
        end if;
    end process;
end Behavioral;
```

VHDL can be synthesized, but also executed in software (**simulation**).



- Simplified Virtex-5 XC5VFXxxxT floor plan
- Frequently used high-level components are provided in discrete silicon
- **BlockRAM (BRAM)**: set of blocks that each store up to 36 kbits of data
- **DSP48 slices**: 25x18-bit multipliers followed by a 48-bit accumulator
- CPU: two full embedded PowerPC 440 cores

# Development Board with Virtex-5 FPGA



source: Xilinx Inc., ML50x Evaluation Platform. User Guide.



Low-level speed of configurable gates is slower than in custom-fabricated chips (clock frequencies:  $\sim 100$  MHz).  
→ Compensate with efficient circuit for problem at hand.

---

	Virtex-5 XC5VLX110T
--	------------------------

---

Lookup Tables (LUTs)	69,120
Block RAM (kbit)	5,328
DSP48 Slices	64
PowerPC Cores	0
max. clock speed	$\approx 450$ MHz
release year	2006

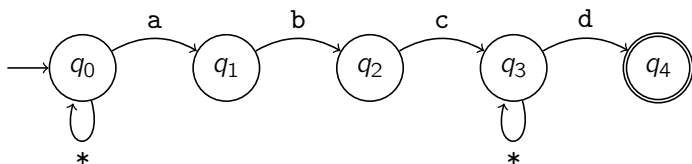
---

# State Machines

The key asset of FPGAs is their inherent **parallelism**.

- Chip areas naturally operate independently and in parallel.

For example, consider **finite-state automata**.



→ non-deterministic automaton for `.*abc.*d`

## How would you implement an automaton in software?

Problems with state machine implementations in software:

- In **non-deterministic automata**, several states can be active at a time, which requires **iterative** execution on sequential hardware.
- **Deterministic automata** avoid this problem at the expense of a significantly higher **state count**.



Automata can be translated mechanically into hardware circuits.

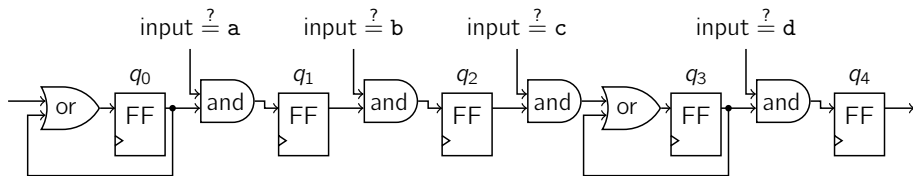
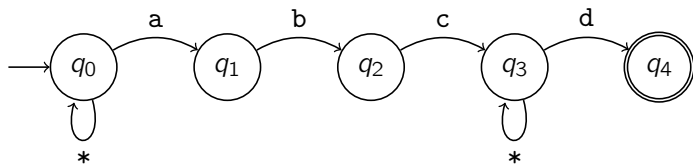
- each **state** → **flip-flop**

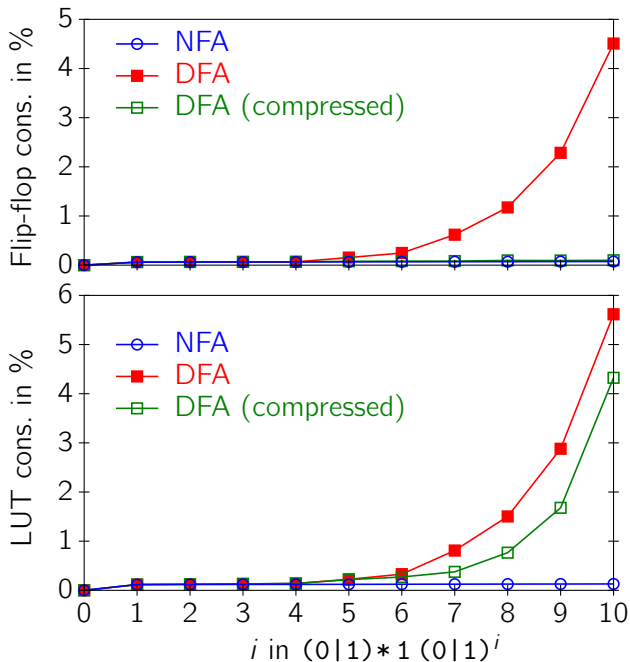
(A flip-flop holds a single bit of information. Just the right amount to keep the 'active'/'not active' information.)

- **transitions:**

- → **signals** ("wires") between states
- **conditioned** on current input symbol ( $\leadsto$  'and' gate)
- **multiple sources** for one flip-flop input → **'or' gate**.

# State Machines in Hardware





# Use Case: Network Intrusion Detection

Analyze network traffic using **regular expressions**.

- Scan for known attack tools.
- Prevent exploitation of known security holes.
- Scan for shell code.

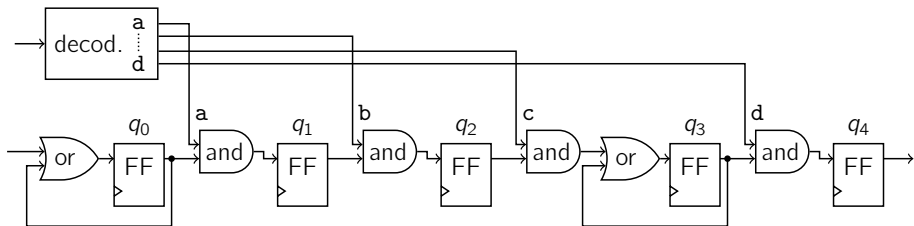
*E.g.*, Snort (<http://www.snort.org/>)

→ Hundreds of (regular expression-based) **rules**.

**Idea:** Instantiate a hardware state machine for each rule.

- Leverage available hardware parallelism.
- Challenge: optimize for **high throughput**.

## Optimization 1: Centralized character classification

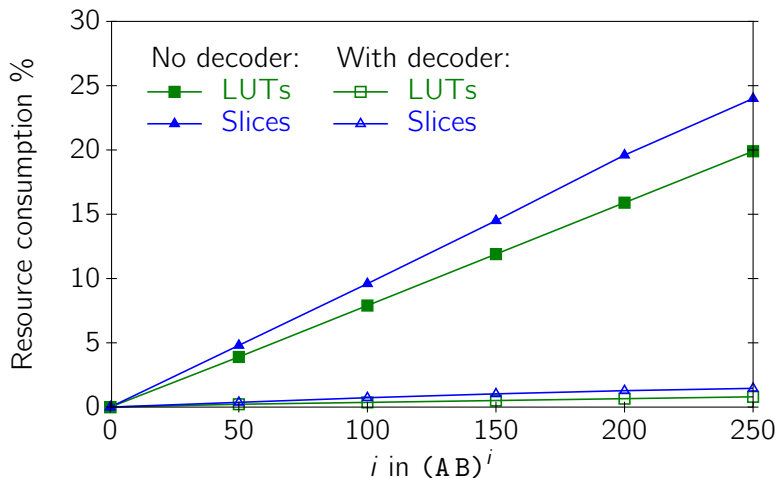


→ Optimizes for **space**, **not** for speed.

Character/predicate decoder:

- Use FPGA logic resources **or**
- use on-chip **BRAM** (configure as ROM and use as lookup table).

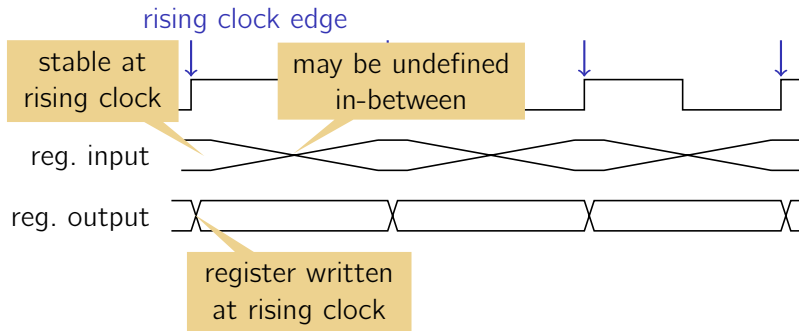
# Predicate Decoding Factored Out



# Signal Propagation Delay

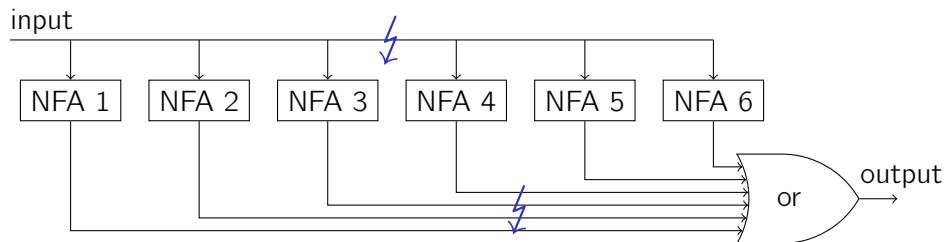
**Signal propagation delays** determine a circuit's **speed**.

- Here: One state transition per clock cycle.
- **Longest signal path** → **maximum clock frequency**



# Propagation Delays and Many State Machines

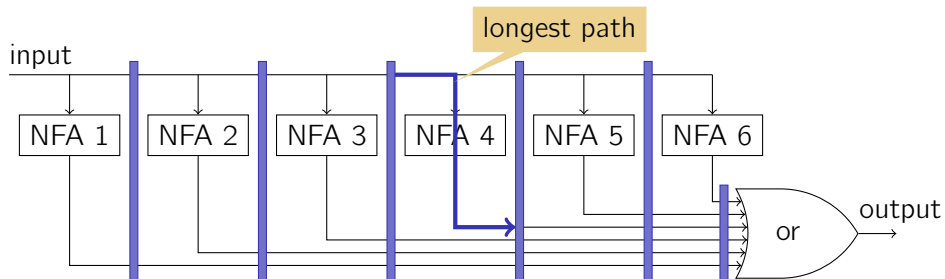
Straightforward design with many rules and one input:





## Optimization 2: **Pipelining**

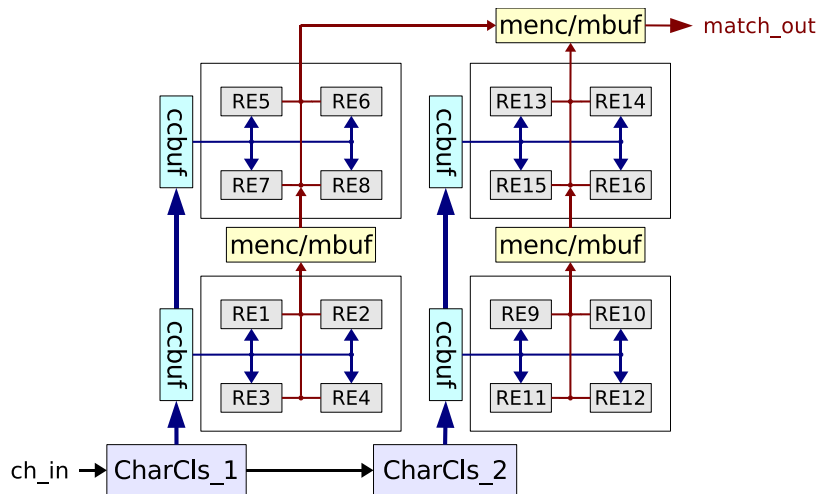
→ What matters is longest path between any two registers (flip-flops).



→ Introduce **pipeline registers**.

→  **Flip side of the idea?**

# Pipelining in Practice



Yang et al. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. ANCS 2008.

# Multi-Character Matching

In a finite state automaton, the state  $s_{i+1}$  at step  $i + 1$  depends on

- the previous state  $s_i$ ,
- the input symbol  $\sigma_i$ , and
- a transition function  $f$ :

$$s_{i+1} = f(s_i, \sigma_i) \text{ .}$$

Consequently:

$$s_{i+2} = f(s_{i+1}, \sigma_{i+1}) = f(f(s_i, \sigma_i), \sigma_{i+1}) \text{ .}$$

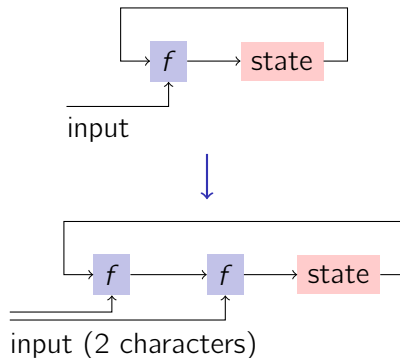
That is, with help of a new transition function

$$F(s_i, \sigma_i, \sigma_{i+1}) \stackrel{\text{def}}{=} f(f(s_i, \sigma_i), \sigma_{i+1}) \text{ ,}$$


an automaton can accept **two input symbols per clock cycle**.

# Multi-Character Encoding

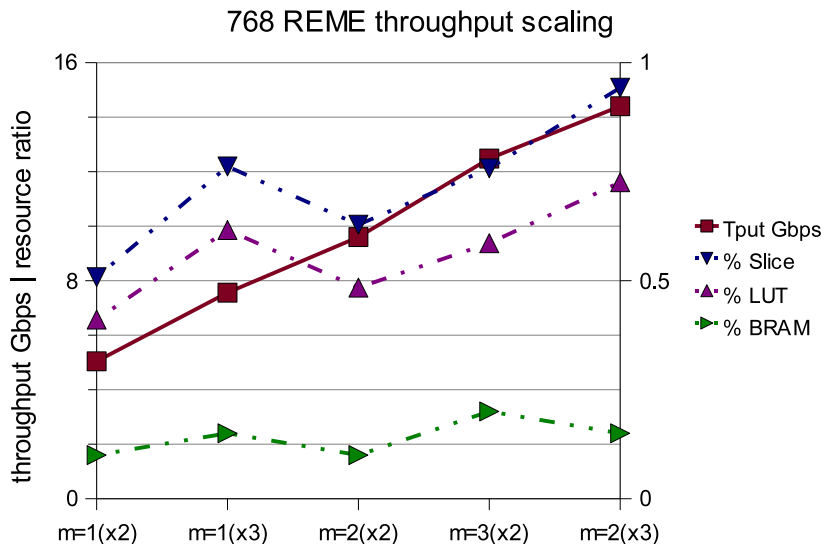
In hardware:



- Trade-off: space  $\leftrightarrow$  performance

-  longer signal paths

# Putting it Together (Snort Workload)



(Virtex-4 LX100;  $\approx 100k$  4-LUTs;  $\approx 100k$  flip-flops)

# Use Case: XML Projection


## Example:

```
for $i in //regions//item
  return <item>
    { $i/name }
    <num-categories>
      { count ($i/incategory) }
    </num-categories>
  </item>
```

## Projection paths:

```
{ //regions//item,
  //regions//item/name #,
  //regions//item/incategory }
```

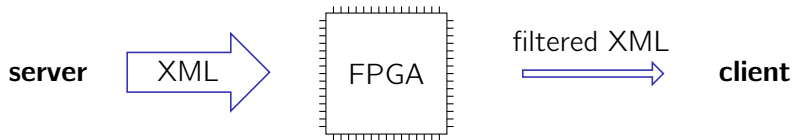
keep descendants



**Challenge:** Avoid explicit synthesis for each query.

# Advantage: FPGA System Integration

**Here:** In-network filtering

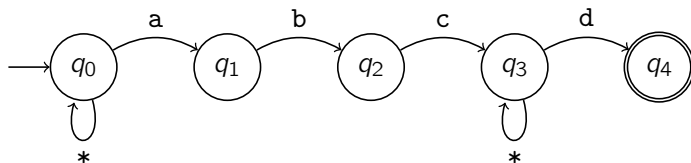


In general: FPGA **in the data path**.

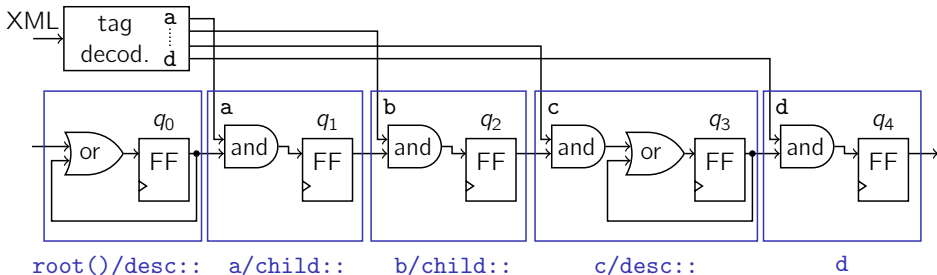
- disk → CPU
- memory → CPU
- ...

# XPath → Finite State Automata

Automaton for `//a/b/c//d`:

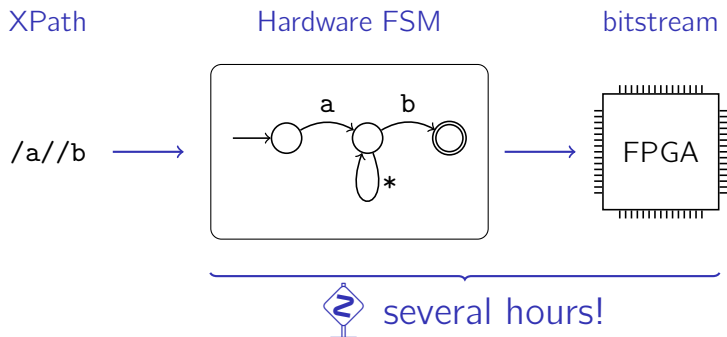


**In hardware:** (see also earlier slides)





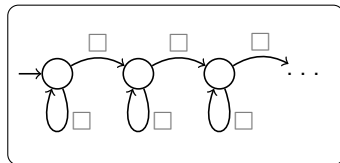
# Compilation to Hardware



# Skeleton Automaton

Separate the **difficult** parts from the **latency-critical** skeleton

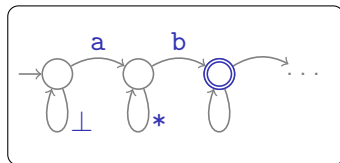
XPath spec.  
→



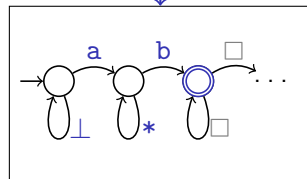
static part (off-line)

dynamic part (runtime)

user query  
/a//b  
→



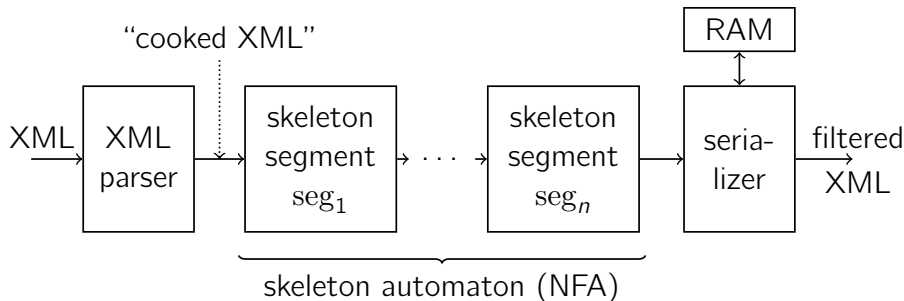
configuration param.



FPGA

# Skeleton Automaton

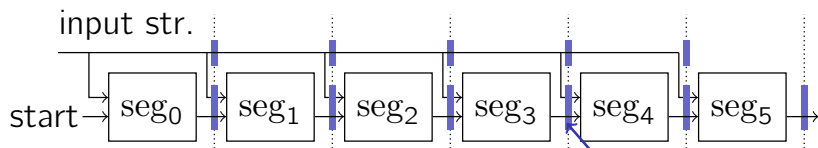
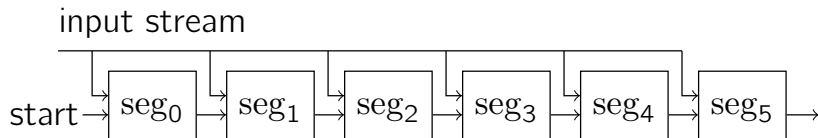
**Thus:** Build skeleton automaton that can be **parameterized** to implement **any** projection query.



**Intuitively:**

- Runtime-configuration determines presence of  $\cup^*$ .

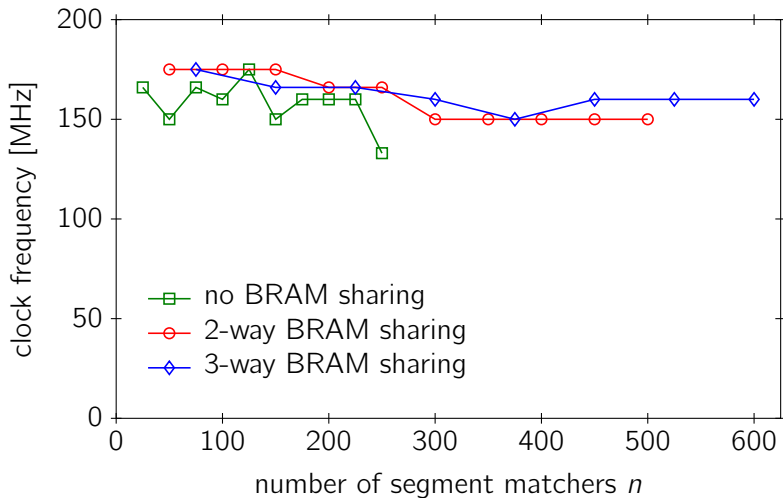
## Again: Pipelining



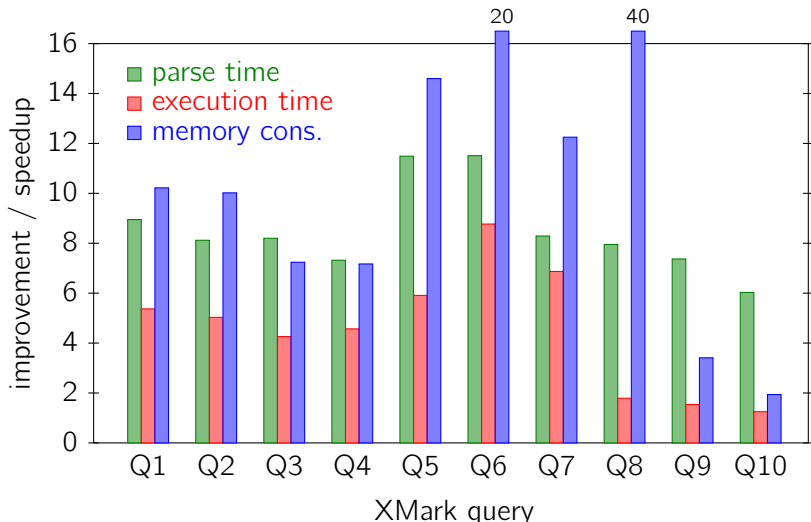
**pipeline registers**

→ Side effect: Can support **self** and **descendant-or-self** axes.

# Scalability



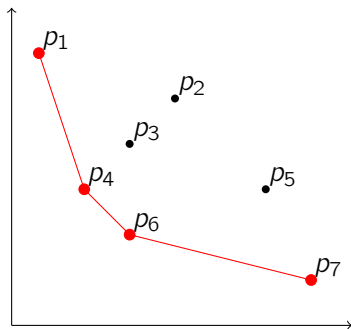
# Application Speedup



↗ Jens Teubner, Louis Woods, and Chongling Nie. Skeleton Automata for FPGAs: Reconfiguring without Reconstructing. *SIGMOD 2012*.

## Problem:

- **Pareto-optimal** set of multi-dimensional data points.
- $x$  **dominates**  $y$  ( $x \prec y$ ) iff for every dimension  $i$ :  $x_i \leq y_i$  and for at least one dimension  $j$ :  $x_j < y_j$ .
- **Skyline points**: all  $y$  not dominated by any  $x$ .

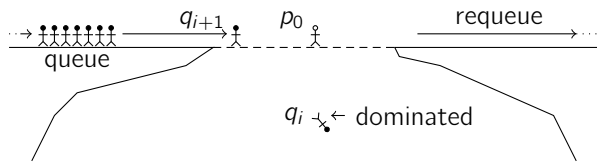


→ **Parallelize**, keep on-chip **routing distance short**

# “Lemming’s Got Talent”

- Lemmings have multiple skills (dimension)
- Determine “best” Lemmings

Let Lemmings **battle** on a **narrow bridge**:

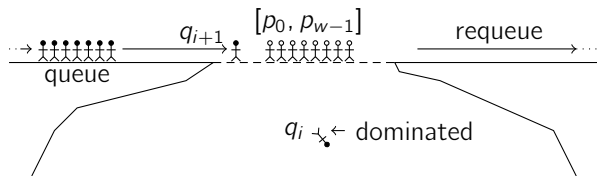


- $p_0$  dominates  $q_i \rightarrow q_i$  falls off the bridge.
- $q_i$  dominates  $p_0 \rightarrow p_0$  falls off bridge,  $q_i$  becomes new  $p_0$
- Battle undecided  $\rightarrow$  let  $q_i$  requeue.
- A Lemming that has survived a full round is a “skyline Lemming.”



# “Lemming’s Got Talent” —Second Year

To speed up the process, let a **set of**  $p_i$  stay on bridge:



- Challengers  $q_i$  fight against multiple  $p_j$  in turn.
- $q_i$  and/or multiple  $p_j$  might fall off the bridge.
- Keep surviving  $q_i$  on bridge if there is space, otherwise requeue.
- Standard algorithm **Block Nested Loops (BNL)**.

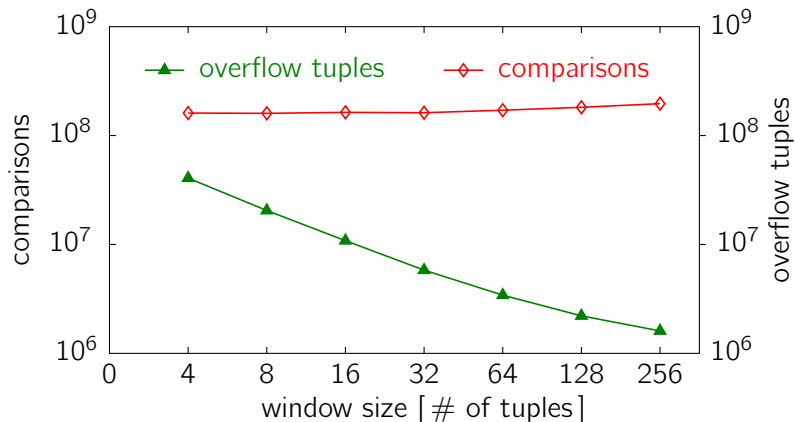
```

1  foreach Lemming  $q_i \in queue$  do
2       $isDominated = false$ ;
3      foreach Lemming  $p_j \in bridge$  do
4          if  $q_i.timestamp > p_j.timestamp$  then
5               $\lfloor$   $bridge.movetoskyline(p_j)$ ;          /*  $p_j \in$  Lemming skyline */
6          else if  $q_i \prec p_j$  then
7               $\lfloor$   $bridge.drop(p_j)$ ;
8          else if  $p_j \prec q_i$  then
9               $\lfloor$   $isDominated = true$ ;
10              $\lfloor$   $break$ ;
11
12     if not  $isDominated$  then
13          $\lfloor$   $timestamp(q_i)$ ;
14         if  $bridge.isFull()$  then
15              $\lfloor$   $queue.insert(q_i)$ ;
16         else
17              $\lfloor$   $bridge.insert(q_i)$ ;

```

# Block Nested Loops Algorithm

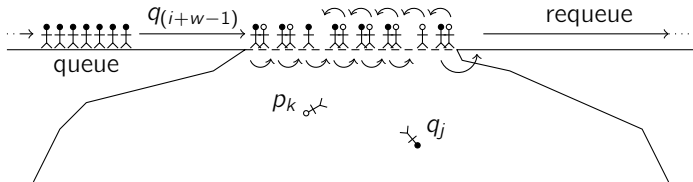
Design goal of BNL: **Eliminate I/O Bottleneck**



→ Compute load remains (mostly) unchanged.

# “Lemming’s Got Talent” —Third Year

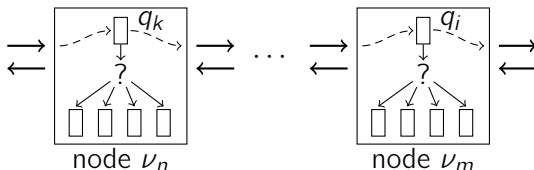
Let multiple (pairs of) Lemmings battle **in parallel**.



- Challengers  $q_i$  move from left to right.
- Potential skyline Lemmings  $p_j$  move from right to left.
- Either can fall off the cliff if dominated.
- On the right end, challengers become potential skyline Lemmings (if there is space on the bridge), otherwise they requeue.

# Parallel BNL with FPGAs

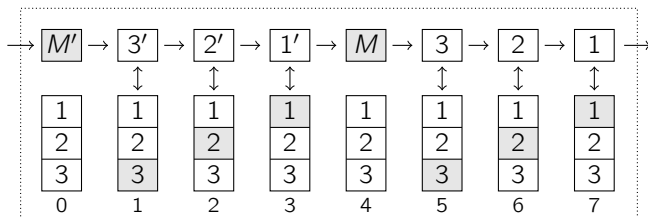
Parallel battles can be realized on distinct processing nodes  $\nu_i$ .



- Nodes form a list where  $\nu_j$  only communicates with  $\nu_{j-1}$  and  $\nu_{j+1}$ .
  - Challengers  $q_i$  forwarded from left to right.
  - Potential skyline tuples forwarded from right to left.
- Effectively,  $q_i$  **scans** over current window (as in BNL).
- **Trick: Causality** still holds.  $q_i$  “sees” effect of any preceding challenger, but not of any following challenger.

# Implementation

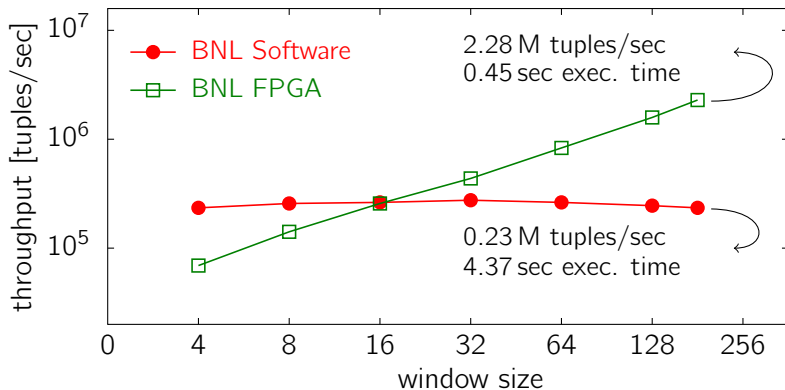
- Let all  $\nu_i$  operate in lock-step.<sup>7</sup>
- Process in **two alternating phases**:
  - 1 **Evaluate**: Compute dominance; drop tuples if need be.
  - 2 **Shift**: Exchange data (“Lemmings”) between nodes.
- In practice, exchanging tuples is more tricky. For high dimensionality data can be passed only **one dimension at a time**.



<sup>7</sup>We tried to avoid this when we did “handshake joins” on multi-core hardware, because of the high synchronization cost. But on FPGAs this is really cheap.

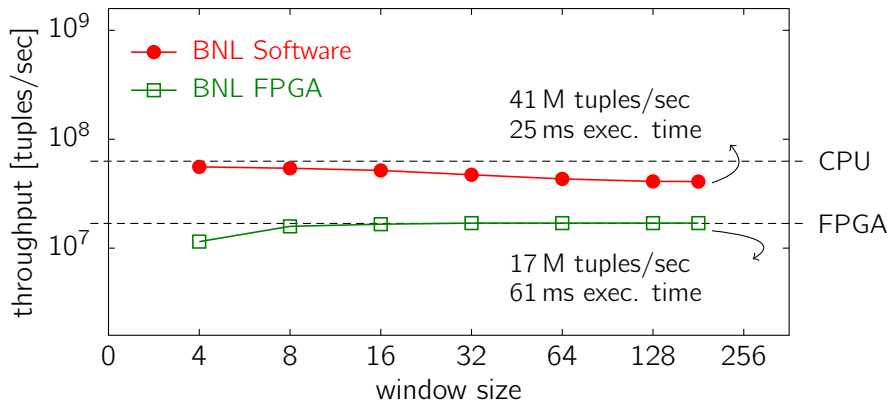
# Experiments

Randomly distributed data; seven dimensions (1.48 % skyline density).



# Experiments

Correlated data; seven dimensions (0.013 % skyline density).

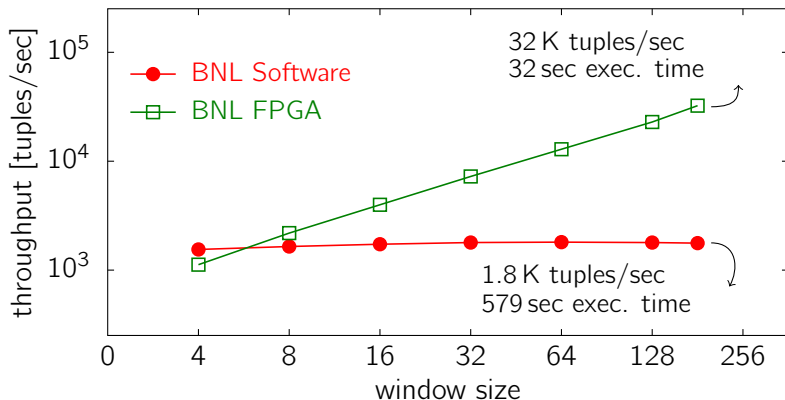


→ FPGA bottlenecked by the **memory interface** of the particular FPGA board.



# Experiments

Anti-Correlated data; seven dimensions (19.8 % skyline density).



→ Benefit of FPGA solution is greatest when it is most needed (*i.e.*, when running times are very high).

# The Frequent Item Problem

## Problem:

Given an input stream  $S$ , which items in  $S$  occur most often?

- Exact solution too expensive ( $\mathcal{O}(\min\{|S|, |A|\})$  space)
- Good **approximate** solutions available.
  - *Space-Saving* by Metwally *et al.*
  - In-depth study: Cormode and Hadjieleftheriou (VLDB 2008)

*Space-Saving* tries to “monitor” only items that are frequent.

```
1 foreach stream item  $x \in S$  do
```

lookup by *item*

```
2   find bin  $b_x$  with  $b_x.item = x$  ;
```

```
3   if such a bin was found then
```

```
4      $b_x.count \leftarrow b_x.count + 1$  ;
```

```
5   else
```

lookup by *count*

```
6      $b_{min} \leftarrow$  bin with minimum count value ;
```

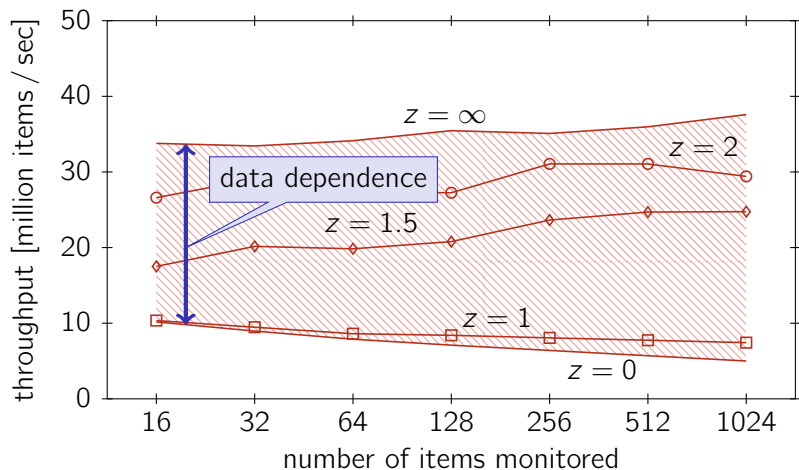
```
7      $b_{min}.count \leftarrow b_{min}.count + 1$  ;
```

```
8      $b_{min}.item \leftarrow x$  ;
```

## Main complexity:

- Look up bin that monitors the input item  $x$ .
- Find bin with minimum *count* value.

# Space-Saving in Software

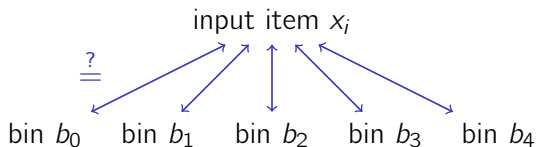


■ Code by Cormode and Hadjieleftheriou, Intel Core2 Duo, 2.66 GHz

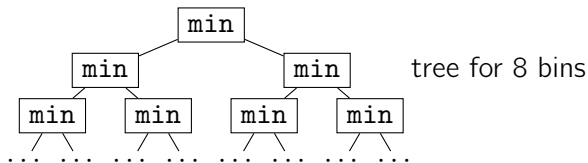
# Data-Parallel Frequent Item on FPGAs

**Idea:** Use available (data) parallelism to make searches efficient.

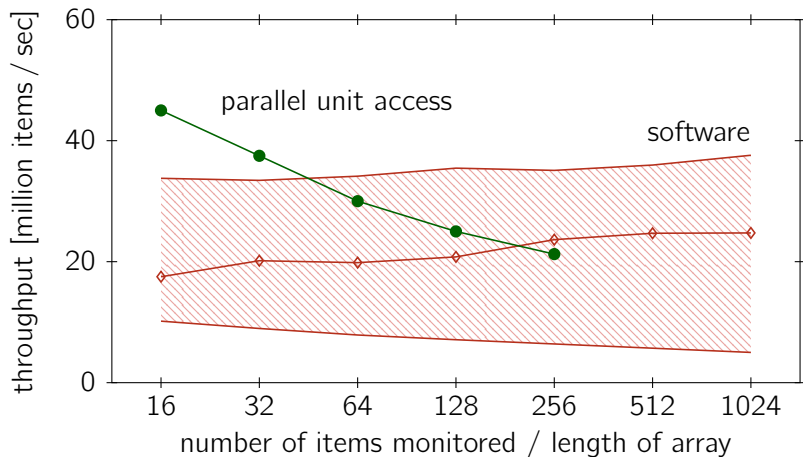
Perform all **item searches** in parallel:



Find bin with **minimum count** using a tree:



# Evaluation

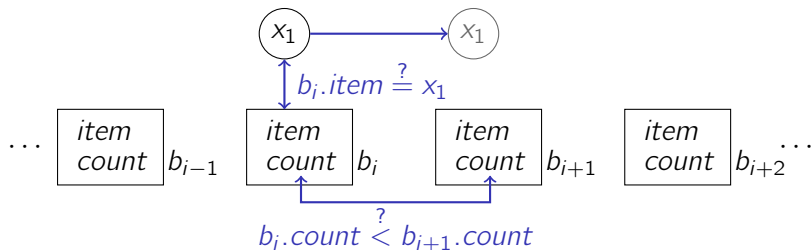


**Problem:** Increasing **signal propagation delays**.

Teubner, Müller, and Alonso. FPGA Acceleration for the Frequent Item Problem. *ICDE 2010*.

# Don't Think in Software

- Organize monitored items as an **array** ( $\rightarrow$  keep things local).



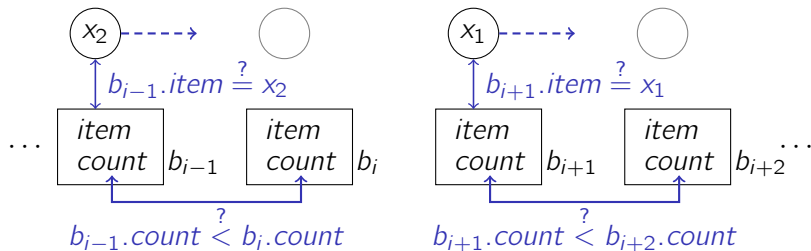
- Compare** input item  $x_1$  to content of bin  $b_i$  (and **increment** *count* value if a match was found).
  - Order** bins  $b_i$  and  $b_{i+1}$  according to *count* values.
  - Move**  $x_1$  **forward** in the array and **repeat**.
- $\rightarrow$  Drop  $x_1$  into **last bin** if no match can be found.

# Pipelining

The idea seems **terribly inefficient**:  $\mathcal{O}(\# \text{ bins})$  vs.  $\mathcal{O}(\log(\# \text{ bins}))$ .

**But:**

- All sub-tasks are **simple**, all processing stays **local**.
- Thus, the processing of multiple input items can be **parallelized**.



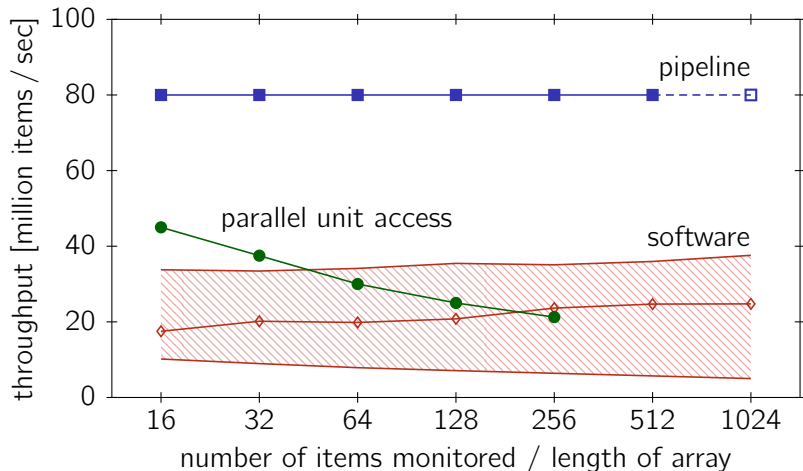
- Multiple input items  $x_i$  can traverse this **pipeline** if they keep **sufficient distance**.



# Algorithm

```
1 foreach stream item  $x \in S$  do
2    $i \leftarrow 1$  ;
3   while  $i < k$  do
4     if  $b_i.item = x$  then
5        $b_i.count \leftarrow b_i.count + 1$  ;
6       continue foreach ;
7     else if  $b_i.count < b_{i+1}.count$  then
8       swap contents of  $b_i$  and  $b_{i+1}$  ;
9     else
10       $i \leftarrow i + 1$  ;
11    /* replace last bin if  $x$  was not found */
12     $b_k.count \leftarrow b_k.count + 1$  ;
13     $b_k.item \leftarrow x$  ;
```

# Evaluation



Teubner, Müller, and Alonso. FPGA Acceleration for the Frequent Item Problem. *ICDE 2010*.