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.

This design point matches the dual-core microprocessor on 45nm technology (Core2 Duo), integrating two cores of 25 million transistors each and 6MB of cache in a die area of about 100mm². If this analysis is performed for future technologies, assuming (our best estimates) modest frequency increase 15% per generation, 5% reduction in supply voltage, and 25% reduction of capacitance, then the results will be as they appear in Table 1. Note that over the next 10 years we expect increased total transistor count, following Moore's Law, but logic transistors increase by only 3x and cache transistors increase more than 10x. Applying Pollack's Rule, a single processor core with 150 million transistors will provide only about 2.5x microarchitecture performance improvement over today's 25-million-transistor core, well shy of our 30x goal, while 80MB of cache is probably more than enough for the cores (see Table 3).

The reality of a finite (essentially fixed) energy budget for a microprocessor must produce a qualitative shift in how chip architects think about architecture and implementation. First, energy-efficiency is a key metric for these designs. Second, energy-proportional computing must be the ultimate goal for both hardware architecture and software-application design. While this ambition is noted in macro-scale computing in large-scale data centers, the idea of micro-scale energy-proportional computing in microprocessors is even more challenging. For microprocessors operating within a finite energy budget, energy efficiency corresponds directly to higher performance, so the quest for extreme energy efficiency is the ultimate driver for performance.

In the following sections, we outline key challenges and sketch potential approaches. In many cases, the challenges are well known and the subject of significant research over many years. In all cases, they remain critical but daunting for the future of microprocessor performance:

- **Organizing the logic:** Multiple cores and customization.

The historic measure of microprocessor capability is the single-thread performance of a traditional core. Many researchers have observed that single-thread performance has already leveled off, with only modest increases expected in the coming decades. Multiple cores and customization will be the major drivers for future microprocessor performance (total chip performance). Multiple cores can increase computational throughput (such as a 1x–4x increase could result from four cores), and customization can reduce execution latency.

---

### Table 1: Throughput Comparison

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Large-core throughput</th>
<th>Small-core throughput</th>
<th>Total throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large-Core Homogeneous</td>
<td>1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Small-Core Homogeneous</td>
<td>1</td>
<td>(5/25)^0.5 = 0.45</td>
<td>11</td>
</tr>
<tr>
<td>Small-Core Homogeneous</td>
<td>1</td>
<td>(5/25)^0.5 = 0.45</td>
<td>11</td>
</tr>
</tbody>
</table>
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).
FPGAs are not instruction set processors.
→ Cannot run (sequential) “programs.”

One could build an instruction set processor using an FPGA.
→ Bad idea. FPGA \(\approx 14\times\) slower than equivalent ASIC.
→ “If you want an instruction set processor, buy an instruction set processor.”

Instead:
- Create arbitrary logic circuits.
- Hardware description language (HDL).
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

programmable Switch Box and bundle of lines

programmable intersection point

programmable switch with memory cell

SRAM cell
Configurable Logic Block (CLB)

- Configurable Logic Block (CLB) in
- implements \( \{0, 1\}^4 \rightarrow \{0, 1\} \) function
- stores a single bit
- SRAM cell
- 4-LUT
- Flip Flop
- Multiplexer
- out
- clock
Programming is usually done using a \textit{hardware description language}.

- \textit{E.g.}, \texttt{VHDL}\textsuperscript{12}, Verilog
- High-level circuit description

Circuit description is compiled into a \texttt{bitstream}, then loaded into SRAM cells on the FPGA:
HDLs enable programming language-like descriptions of hardware circuits.

```vhdl
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).
Real-World Hardware

- 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 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

Virtex-5
XC5VLX110T

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup Tables (LUTs)</td>
<td>69,120</td>
</tr>
<tr>
<td>Block RAM (kbit)</td>
<td>5,328</td>
</tr>
<tr>
<td>DSP48 Slices</td>
<td>64</td>
</tr>
<tr>
<td>PowerPC Cores</td>
<td>0</td>
</tr>
<tr>
<td>max. clock speed</td>
<td>$\approx$ 450 MHz</td>
</tr>
<tr>
<td>release year</td>
<td>2006</td>
</tr>
</tbody>
</table>


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.
The key asset of FPGAs is their inherent **parallelism**.
- Chip areas naturally operate independently and in parallel.

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

\[
\begin{array}{c}
q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{c} q_3 \xrightarrow{d} q_4 \\
\end{array}
\]

→ 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 (→ ‘and’ gate)
  - **multiple sources** for one flip-flop input → ‘or’ gate.
State Machines in Hardware

\[
\begin{array}{cccccc}
q_0 & \xrightarrow{a} & q_1 & \xrightarrow{b} & q_2 & \xrightarrow{c} & q_3 & \xrightarrow{d} & q_4 \\
\text{or} & \text{FF} & \text{and} & \text{FF} & \text{and} & \text{FF} & \text{or} & \text{FF} & \text{and} & \text{FF} \\
\ast & \text{input } \overset{?}{=} a & \text{input } \overset{?}{=} b & \text{input } \overset{?}{=} c & \text{input } \overset{?}{=} d \\

\end{array}
\]
Flip-flop cons. in %

LUT cons. in %

\(i \text{ in } (0|1)^* 1 (0|1)^i\)
Use Case: XML Projection

Example:

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

Projection paths:

```xml
{ //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

XPath → Hardware FSM → bitstream

/a//b

several hours!

© Jens Teubner · Data Processing on Modern Hardware · Winter 2018/19
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

© Jens Teubner · Data Processing on Modern Hardware · Winter 2018/19
Thus: Build skeleton automaton that can be parameterized to implement any projection query.

Intuitively:
- Runtime-configuration determines presence of \( \cup \ast \).
Pipelining

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

![Graph showing the relationship between clock frequency and the number of segment matchers with different BRAM sharing configurations.]