High-level Description of Instruction Sets and Compiler Generation

Ulrich Rückert
Uwe Kastens
Hierarchical System Architecture
From RISC to chip multiprocessor

- **RISC CPU as a starting point (S-Core)**
  - not enough performance due to increasing application demands
  - acceleration through instruction set enhancements (N-Core)
  - processor subsystem with local memory (32 KB)

- **Parallel processor cluster**
  - increase in performance by usage of fine-grained parallelism
  - communication via on-chip bus (Wishbone / AMBA)
  - efficient parallelizing ANSI-C compiler

- **Group of parallel processor clusters**
  - increase in performance by usage of coarse-grained parallelism on task level
  - efficiency through homogeneous architecture
  - powerful communication infrastructure (GigaNoC)
# Hierarchical System Architecture

*From RISC to chip multiprocessor*

<table>
<thead>
<tr>
<th>SoC main components</th>
<th>90 nm (typical case)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>area [mm²]</td>
</tr>
<tr>
<td>32 N-Cores</td>
<td>32 x 0.12</td>
</tr>
<tr>
<td>8 switch-boxes [with 5 ports]</td>
<td>8 x 0.53</td>
</tr>
<tr>
<td>32 local RAMs, (32 KB)</td>
<td>32 x 0.875</td>
</tr>
<tr>
<td>+ 8 local packet buffers (2 x 16 KB)</td>
<td>+ 8 x 2 x 0.466</td>
</tr>
<tr>
<td>8 local on-chip busses</td>
<td>8 x 0.02</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>43.7</strong></td>
</tr>
</tbody>
</table>
System optimization – hierarchical, directed approach

- Processor optimization
- Instruction set extensions
- Extension through tightly coupled HW accelerators
- Extension through loosely coupled HW accelerators
- Parallel processor clusters
**Workflow**

**Determination of super-instructions – two steps**

**first step** is based on a compiler-related workflow

- application is analyzed with respect to the occurrence of *instruction pairs*
- frequently used pairs are extracted and modeled as *super-instructions*
- high abstraction level of the simulation:
  - exploration can be iterated *very fast*
  - allows a *broad search* on the instruction space

**second step** is based on hardware implementation

- delivers detailed information about the hardware parameters like *chip area* and *power consumption*
- consumes much more simulation time
  - detailed evaluation of instructions in the first step assures that only the most promising candidates are implemented and analyzed
Compiler-related workflow
Overview

design spaces
- functional units
- register files
- internal parallelism
- busses
- instruction timing

optimization strategy
- stack layout
- register usage
- calling conventions

instruction set

Compiler Generator

Compiler

Simulator Generator

Simulator

formal model of processor

interpretation of results

representative applications, benchmarks

cycle-accurate simulation
**UPSLA Concept I**

**machine model**
- functional units
- register banks
- connectivity
- virtual resource

**scheduling**
- instruction $i$
  - | cycle | resources |
  - | 0     | X         |
  - | 1     | X         | X         |
  - | 2     | X         | X         |
  - exec-time 3
  - latency 2

**Specification captures instruction semantics:**
- effects
- used resources, FUs, registers
- restrictions
- timing

does not include a hardware design description
**specification language UPSLA (Unified Processor Specification Language)**
UPSLA: Unified Processor Specification Language

- supports grouping of elements into named groups with integer index
- supports equivalence classes of elements with identical resource mappings

- sequence of element declarations
- elements have name, type, and attributes
- fixed repertoire of seven element types in UPSLA
- type determines set of valid attributes and their domains
**UPSLA Concept III**

**Inheritance**

```plaintext
abstract instr unOp
  opddirs out,in
  cycle 0 readPort/in
  cycle 1 writePort/out
endinstr unOp
```

```plaintext
instruction neg
  parent unOp
  asmformat "neg %0,%1"
endinstr neg
```

**Indexed Groups**

```plaintext
funcunit aluA
  group alu
  ...
endfu aluA

funcunit aluD
  group alu
  ...
endfu aluD
```

**Element Names**

- `aluA`
- `aluB`
- `aluC`
- `aluD`

**Group Name and Index**

- `alu[0]`
- `alu[1]`
- `alu[2]`
- `alu[3]`
Automatic Generation of Compiler and Simulator

- instruction set
- resource
- encoding
- semantics

compiler backend
- code selection
- scheduling

simulator
- decoder
- interpreter

evaluation of measurements

processor specification with UPSLA

HEINZ NIXDORF INSTITUTE
University of Paderborn
System and Circuit Technology
Prof. Dr.-Ing. Ulrich Rückert
Code Selection

Code selection by tree pattern matching specified for, generated by augmented burg (BURS)

Code selection generates

Grammar rule

... R: add(R,mult(R,int 4))
cost 2
{
    Reg t = get_reg(node);
    emit_mov(t,kids[0]);
    emit_ixw(t,kids[1]);
    set_reg(node,t);
}
... equivalent textual specification
Exemplary Specification and Code Generation

N-core Specification

... instruction addc
  parent binarith_regreg
  bitpattern 00000110yyyyxxxx
  memberof + cbitread, cbitwrite
  simrecord + readcbit, writecbit
endi addc

instruction addi
  parent binarith_regimd
  bitpattern 0010000oooooxxxx
endi addi

instruction addu
  parent binarith_regreg
  bitpattern 0011100yyyyxxxx
endi addu
...

Baseline-Compiler-Development: 4 weeks
Processor specific optimization: additional 3 weeks

if ((wrd & (1 << 9)) == 0)
  {
    if ((wrd & 0xfffffe00) == 0x00002000)
      return icode_addi;
  } else {
    if ((wrd & 0xfffffe00) == 0x00002200)
      return icode_cmplti;
  }

case asopc_addi:
  p = "addi";
  while (*p) *(buf++) = *(p++);
  *(buf++) = ' ';
  buf = sprint_opd(buf, asm_opd(in, 0));
  *(buf++) = ' ,';
  *(buf++) = ' ';
  buf = sprint_opd(buf, asm_opd(in, 1));
  break;

Generate

Generated Compiler

Generated Simulator
Configuration of Compiler Backend

UPSLA specification

Instruction Set
- Semantics
- Execution Time
- Resource Requirements

Functional Units

Resource Constraints

Register Set
- Register Banks

Optimization Patterns

Compiler Generator
- Recursive Descent Compiler
- generates C-Code and Specs for other Generators (BURS, RD …)

Machine-Independent Optimization

Code Selection

Scheduling
- Allocation of Functional units

Register Allocation

Peephole Optimization
**UPSLA-Framework**  
*Key Characteristics*

Performance comparison (target: PowerPC 405)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Generated Compiler</th>
<th>ANSI-C Compiler</th>
<th>ANSI-C Compiler</th>
<th>ANSI-C Compiler</th>
<th>GNU Compiler</th>
<th>GNU Compiler</th>
<th>GNU Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>none</td>
<td>none</td>
<td>–O1</td>
<td>–O2</td>
<td>none</td>
<td>–O1</td>
<td>–O2</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>5,9</td>
<td>13</td>
<td>12,9</td>
<td>5,7</td>
<td>8,7</td>
<td>5,9</td>
<td>5,8</td>
</tr>
<tr>
<td>Stanford</td>
<td>44,7</td>
<td>68,4</td>
<td>67,8</td>
<td>25,4</td>
<td>76,3</td>
<td>36,5</td>
<td>34,6</td>
</tr>
<tr>
<td>AVL-Tree</td>
<td>5,2</td>
<td>5,7</td>
<td>5,6</td>
<td>4,9</td>
<td>4,4</td>
<td>3,8</td>
<td>3,8</td>
</tr>
<tr>
<td>Scanner</td>
<td>3,1</td>
<td>5,1</td>
<td>5</td>
<td>2,4</td>
<td>5,7</td>
<td>2,8</td>
<td>2,6</td>
</tr>
</tbody>
</table>

**The compiler generator delivers**

- runtime and code size comparable to state of the art compilers
- short compilation times
- easy integration of additional optimization strategies for features of custom architectures
Functional Principle of IL-Simulator:

- Processor structure:
  - Registers
  - Memory
  - Cflag

- Pipeline Model:
  - IF
  - ID
  - EX
  - WB

- Interpreter:
  - Statistical collector

- Program code:
- Machine model:
  - (Gen) decoder
  - (Gen) instr. semantic
  - (Gen) res. allocation

- Statistic data:
  - Mov 99
  - Ldb 20
  - Stw 30
  - Br 13
  - Mov 90
  - Ldb 24
  - Stw 34
  - Br 19
  - Mov 79
  - Ldw 20
  - Stw 20
  - Br 53
**Generated Simulator**

```
instruction addi
    asm "addi %0,%1,#%2"
    bits 0011ddddaaaaabbbbbbbb
    ...
endinstruction addi

instruction mov
    asm "mov %0,%1"
    bits 0011ddddssss00000000
    ...
endinstruction mov
```

```
void sim_addi(unsigned d, unsigned a, unsigned b)
{
    int i;
    i = ((int) b << 24) >> 24;
    reg[d] = reg[a] + i;
    cond = reg[d] == 0;
}

void sim_mov (unsigned d, unsigned s)
{
    reg[d] = reg[s];
}
```

### Instruction Decoding

### Operational Semantics

```
if (((word & m) == c) ... else ...
```

**Balanced Decision Tree**

- executes fast
- collects data on
- cycle-accurate timing
- utilization of functional units
- frequency of used instructions and pairs
Compiler-related Workflow

First step – performance analysis

➢ suggestion: combined instruction \texttt{ldw-xor}
Compiler-related Workflow

First step – performance analysis

- specify instruction ldw-xor ➤ xorldw
- re-generate compiler and simulator
- re-run re-compiled benchmark / application
- compare performance

<table>
<thead>
<tr>
<th>Instruction</th>
<th>S-Core</th>
<th>N-Core (XORLDW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR</td>
<td>262,144</td>
<td>221,184</td>
</tr>
<tr>
<td>LDW</td>
<td>253,959</td>
<td>221,184</td>
</tr>
<tr>
<td>XORLDW</td>
<td>0</td>
<td>221,184</td>
</tr>
<tr>
<td>cycles</td>
<td>3,048,165</td>
<td>2,777,829</td>
</tr>
</tbody>
</table>

*cycle counts of DES code before and after extension*
Hardware-related workflow
Second step – detailed HW analysis

- Instruction set simulator
- Statistical analysis
- C source
- Generated & extended C compiler
- Memory Image
- New super instructions
- Processor modification
- Main processor S-Core
- RTL simulation
- Processing engine
- RAM
- Main processor N-Core
- Processor bus
- Switching activities
- Gate-level synthesis
- Power model creation
- Annotation of power model
- Power estimation
- Analysis / evaluation
- HDL refinement
- Annotation of power model
- Power estimation
**Application-specific case study**

**Results – summary**

**IPSec (DES)**
- 8.9% *speed-up*
- 8.7% *energy savings*
- 0.07% *area increase* for additional logic

*Furthermore:*
- *code size* reduction of 6%
  - for normal core 5924 Bytes
  - for enhanced core 5572 Bytes
- the *saved 352* Bytes on-chip SRAM correspond to an area of about 9240μm²
- this is about the *75-fold of area* needed for the realization of the *super instruction*
- 8kB SRAM (0.21mm²)@205MHz consume about 9.1mW power which is *comparable* to the power consumption of the N-Core
# Hardware Extensions

## Speed-up and area requirements

<table>
<thead>
<tr>
<th>Extension</th>
<th>Speed-up @200MHz</th>
<th>Area @130nm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net-S-Core architecture extension</td>
<td>1.0</td>
<td>+16%</td>
</tr>
<tr>
<td>XORL/DW instruction set extension</td>
<td>1.096</td>
<td>+ 0.0001mm²</td>
</tr>
<tr>
<td>IXD,IXQ instruction set extension</td>
<td>1.12</td>
<td>+ 0.001mm²</td>
</tr>
<tr>
<td>ldwxorlsl8, andshr, orshl8/16/24, ldwaddi, ixwandshr, ldwixw instruction set extension</td>
<td>1.11+ / 1.25*</td>
<td>+ 0.004mm²</td>
</tr>
<tr>
<td>CRC hardware accelerator</td>
<td>15</td>
<td>+ 0.014mm²</td>
</tr>
<tr>
<td>IP filter (incl. 13kB SRAM )</td>
<td>6</td>
<td>2.86mm² / 0.028mm²</td>
</tr>
<tr>
<td>IP headercheck</td>
<td>13</td>
<td>+ 0.03mm²</td>
</tr>
<tr>
<td>AES encryption HW accelerator</td>
<td>160</td>
<td>+ 0.124mm²</td>
</tr>
<tr>
<td>AES decryption HW accelerator</td>
<td>175</td>
<td>+ 0.197mm²</td>
</tr>
<tr>
<td>Content addressable memory</td>
<td>1230</td>
<td>+ 0.72mm²</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallelism / Multiprocessor Cache</th>
<th>Current research</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-Core I/O</td>
<td></td>
</tr>
<tr>
<td>N-Core I/O</td>
<td></td>
</tr>
<tr>
<td>N-Core I/O</td>
<td></td>
</tr>
<tr>
<td>N-Core I/O</td>
<td></td>
</tr>
<tr>
<td>Net-S-Core</td>
<td></td>
</tr>
<tr>
<td>Net-S-Core</td>
<td></td>
</tr>
<tr>
<td>AES …</td>
<td></td>
</tr>
<tr>
<td>CRC …</td>
<td></td>
</tr>
</tbody>
</table>

Without / * with memory

Current research
GigaNetIC Prototyping Platform
Based on the RAPTOR2000 System

Hardware blocks
- 2 Switch boxes
- 2 Ethernet controller (4 x 10/100 Mbps ports)
- 2 Cluster components
  - 32 KB packet buffer, 8 MB shared SRAM
  - 4 N-Core subsystems (32 KB local memory)
  - 1 serial interface (UART)

FPGA utilization
- 12.5 MHz global system clock
- Xilinx Virtex-II 8000 (70% slices, 81% BlockRAM)

**PE level:**
<table>
<thead>
<tr>
<th>HW block</th>
<th>Slices</th>
<th>RAM16s</th>
</tr>
</thead>
<tbody>
<tr>
<td>address decoder</td>
<td>121</td>
<td>-</td>
</tr>
<tr>
<td>bus controller incl. N-Core</td>
<td>3,206</td>
<td>-</td>
</tr>
<tr>
<td>memory interface incl. 32 KB RAM</td>
<td>71</td>
<td>16</td>
</tr>
<tr>
<td>programmable interrupt controller</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>timer</td>
<td>163</td>
<td>-</td>
</tr>
<tr>
<td>Wishbone bridge</td>
<td>95</td>
<td>-</td>
</tr>
<tr>
<td>∑(N-Core subsystem)</td>
<td>3,662</td>
<td>16</td>
</tr>
</tbody>
</table>

**Cluster level:**
<table>
<thead>
<tr>
<th>HW block</th>
<th>Slices</th>
<th>RAM16s</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 x N-Core subsystem</td>
<td>14,648</td>
<td>64</td>
</tr>
<tr>
<td>packet memory (32 KB)</td>
<td>53</td>
<td>16</td>
</tr>
<tr>
<td>SRAM interface</td>
<td>22</td>
<td>-</td>
</tr>
<tr>
<td>UART</td>
<td>626</td>
<td>-</td>
</tr>
<tr>
<td>Wishbone arbiter</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>∑(cluster)</td>
<td>15,362</td>
<td>80</td>
</tr>
</tbody>
</table>

**SoC level:** Ethernet controller (8,998 slices & 56 RAM16s), switch box (7,946 slices)
GigaNetIC Prototyping Platform

Graphical User Interface

Configuration
- Bitstream upload (FPGA)
- Program upload (N-Core)

Controlling
- N-Core (start, stop, reset)
- UART (input / output)
- Ethernet port configuration

Debugging
- Memory dump
- Status information
- Sending test packets

Self-adapting
- Number of processors
- Number of Ethernet ports
- Size of memory blocks
Evaluation & Verification Environment

- N-Core compiler, profiler and simulator 8 MHz
- SystemC simulation 50 kHz
- VHDL simulation 100 Hz
  - PERFMON environment
- FPGA-based rapid prototyping 20 MHz
Resource-efficient MANET processor
Specialized architecture implementation

technical data
- 180 nm
- UMC standard cell
- 100 MHz @ 25° C
- 750 mW
- 0.7-1.6 Gbps per link
- 0.25 Gbps external
- 2.1 MBit memory
- 1.6 M logic cells
The UPSLA Framework offers

- support for application specific instruction set extensions independent of a specific architecture
- a very fast cycle accurate simulator
Conclusion

- performance of the generated ANSI-C compiler can be close to commercial products

- compared to related approaches:
  
  - feedback driven application specific instruction-set extension method
  
  - we do not analyze data flow graphs, but statistics from cycle accurate simulation

  - there is no need to recode the application in case of new instructions, it is the compiler's burden to make use of the new instruction

  - automatic fine-grained parallelization based on “unstructured” ANSI-C code
Current work

- improving user front end for UPSLA (GUI)
- automatic generation of VHDL out of UPSLA
- improving parallelization capabilities of the compiler
Thank you for your attention!

Ulrich Rückert
M. Grünewald, T. Jungeblut, J.C. Niemann, M. Porrmann, C. Puttmann

Uwe Kastens
O. Bonorden, G. Hagen, D. Khoi Le, M. Thies, A. Slowik,