Generation of Efficient C Compilers for VLIW-DSPs

Gert Goossens, Werner Geurts, Dirk Lanneer, Sabine Van Cromphaut, Steven Van Rompaey

Target Compiler Technologies

gert.goossens@retarget.com
http://www.retarget.com
mAgicV in the Shapes multi-tile platform
Chess/Checkers - a retargetable tool suite for ASIP design
Targeting the Chess compiler to mAgicV
VLIW challenges
  ▪ Wide instruction word
  ▪ Predicated execution
Conclusions
mAgicV in Shapes multi-tile platform

- Floating-point DSP
- VLIW
- 128 bit uncompressed
- Dual Harvard, quad-mpy
  - 4 multipliers
  - 4 adder-subtractors
  - separate AGUs
  - 2x 128 gen.-purp. registers
  - vector operations
- Deep pipeline
  - IF, ID, ID2, C0, C1, C2, C3, C4, C5, C6, C7
### Instruction word (uncompressed)

<table>
<thead>
<tr>
<th>MUXOUT</th>
<th>PRED</th>
<th>RFA7P</th>
<th>FLOW</th>
<th>RFA7</th>
<th>RFA6</th>
<th>ARFY1</th>
<th>ARFY0</th>
<th>RFA4</th>
<th>RFA3</th>
<th>RFA2</th>
<th>RFA1</th>
<th>RFA0</th>
<th>AGU1</th>
<th>AGU0</th>
<th>DMQADD</th>
<th>DMQADU</th>
<th>RFA0P</th>
<th>RFA1P</th>
<th>RFA2P</th>
<th>RFA3P</th>
<th>RFA4P</th>
<th>RFA5P</th>
</tr>
</thead>
<tbody>
<tr>
<td>119..118</td>
<td>117..114</td>
<td>113..113</td>
<td>112..107</td>
<td>106..99</td>
<td>98..91</td>
<td>90..87</td>
<td>96..83</td>
<td>92..75</td>
<td>74..67</td>
<td>66..59</td>
<td>56..51</td>
<td>50..43</td>
<td>42..39</td>
<td>38..32</td>
<td>31..28</td>
<td>27..21</td>
<td>20..14</td>
<td>13..7</td>
<td>8..6</td>
<td>5..5</td>
<td>4..4</td>
<td>3..3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARF4SRC</th>
<th>ARF2SRC</th>
<th>ARF2SRC</th>
<th>ARF6SRC</th>
<th>ARFAR1</th>
<th>ARFAR0</th>
</tr>
</thead>
<tbody>
<tr>
<td>90..87</td>
<td>88..83</td>
<td>ARF6DST</td>
<td>MM4H</td>
<td>82..43</td>
<td>39..0</td>
</tr>
<tr>
<td>90..87</td>
<td>88..83</td>
<td>ARF4DST</td>
<td>MM10</td>
<td>82..43</td>
<td>39..0</td>
</tr>
<tr>
<td>90..87</td>
<td>88..83</td>
<td>ARF0SRC</td>
<td>MM10L</td>
<td>82..43</td>
<td>39..0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FLOWADDSDTT</th>
<th>JMPIADD</th>
<th>106..99</th>
<th>98..83</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOWADDSSRC</td>
<td>JMPIOFF</td>
<td>106..99</td>
<td>98..83</td>
</tr>
</tbody>
</table>

**MM120** (e.g. for loading start stages of software pipelining 8 groups of 15-bit)

119..0
mAgicV and the Shapes multi-tile platform

Chess/Checkers - a retargetable tool suite for ASIP design

Targeting the Chess compiler to mAgicV

VLIW challenges
  - Wide instruction word
  - Predicated execution

Conclusions
Chess/Checkers

Retargetable tool-suite
nML: processor description language

- Programmer’s model (cf. manual)
- Few 1000 lines — Learning: few weeks

### Structural skeleton
Storages & connectivity

```plaintext
mem DM[1024]<num, addr>;
reg R[4]<num>;
pipe C<num>;
trn A<num>; trn B<num>;
fu alu;
...
```

### Instruction-set grammar
- Or-rule = alternatives
- And-rule = parallelism
- Instruction behaviour in register-transfer model
- User-defined primitive functions

```plaintext
opn my_core (alu_inst | mac_inst | shift_inst);
...

opn alu_inst (op:opcod, x:c2u, val:c16s, y:c2u) {
    action {
        stage EX1:
            A = R[x];
            B = val;
            switch (op) {
                case add : C = add(A, B) @alu;
                case sub : C = sub(A, B) @alu;
                case and : C = and(A, B) @alu;
                case or : C = or(A, B) @alu;
            }
        stage EX2:
            R[y] = C @alu;
    }
    syntax : op " R" y ", R" x ", " val;
    image : "0"::op::x::y::val;
}
...
```
Chess: graph-based C compilation

Front end
- C \rightarrow \text{Control-Data Flow Graph}
- nML \rightarrow \text{Instruction-Set Graph}

Compilation phases
- Map CDFG onto ISG
- Graph algorithms

ISG contains structural info
- Hardware resources, data-types, connectivity, instr. encoding, pipelining, instr.-level parallelism, pipeline hazards
- Much closer to hardware than conventional compilers (e.g. gcc)
- Enables efficient compilation for “irregular” architectures
- Patented
mAgicV and the Shapes multi-tile platform

Chess/Checkers - a retargetable tool suite for ASIP design

Targeting the Chess compiler to mAgicV

VLIW challenges
  - Wide instruction word
  - Predicated execution

Conclusions
Chess/Checkers for mAgicV

Work done

- Development of nML processor model
- Development of simulation models for ISS
- Generation of C compiler and ISS
- Development of on-chip debug interface

(Target, Atmel) (Atmel) (automAgic) (Target - ongoing)
Chess compiler for mAgicV

ChessDE integrated development environment
## Chess compiler for mAgicV

### Results: cycle count and program code size

<table>
<thead>
<tr>
<th>function</th>
<th>Theoretical Performance on mAgicV (1)</th>
<th>Hand-optimised assembly code</th>
<th>Chess compiler for mAgicV</th>
<th>TI C67 family (from web site)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Sum</td>
<td>0,875 × N + K</td>
<td>0,875 × N + 14 256 bytes</td>
<td>0,875 × N + 12 240 bytes</td>
<td>1 × N + 8 192 bytes (based on vecmul size)</td>
</tr>
<tr>
<td>Vector Complex Product</td>
<td>1,5 × N + K</td>
<td>1,5 × N + 14</td>
<td>1,5 × N + 12</td>
<td>2 × N + 18 (2)</td>
</tr>
<tr>
<td>Complex FIR</td>
<td>(K' + M) × L + K + 416 bytes</td>
<td>(23 + M) × L + 25 + 24 560 bytes</td>
<td>(24 + 1,25 × M) × L + 24 560 bytes</td>
<td>(2 × M + 14) × L + 17 + (L-1) 640 bytes</td>
</tr>
<tr>
<td>Vector Division</td>
<td>3 × N + 18 (2)</td>
<td>3 × N + 40 240 bytes</td>
<td>3 × N + 27 416 bytes</td>
<td>8 × ((N-1)/4) + 53 512 bytes</td>
</tr>
<tr>
<td>Vector SQRT</td>
<td>7 × N + 8</td>
<td>7 × N + 80</td>
<td>7 × N + 39</td>
<td>32 bit precision Computing 1/x</td>
</tr>
<tr>
<td>Vector Sin</td>
<td>9,5 × N + K</td>
<td>10 × N + 124</td>
<td>10 × N + 71</td>
<td></td>
</tr>
<tr>
<td>IIR</td>
<td>28 + ((5 × (Stages_Nr - 3) + 32) × Ch_Nr + 16) Samples_Nr/2</td>
<td>4 × Stages_Nr + 42 (3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) K represents the function prologue cycle count
(2) Estimate
(3) Only inner loop; 4 × Stages_Nr + 42 is the 5 multiply implementation; (5 × Stages_Nr + 32)/2 includes the 4 multiply biquad with global gain extraction
### FIR filter

```
for(k=0; k<hLEN; k+=4) chess_loop_range(4,){
    acc = acc + *pp-- * *pp1++;
    acc1 = acc1 + *pp-- * *pp1++;
    acc2 = acc2 + *pp-- * *pp1++;
    acc3 = acc3 + *pp-- * *pp1++;
}
```

**Output ASM**

```
CMUL 22 14 12 | VM2RF0_U-- 1e 1 | CADD 10 22 10 | VM2RF1_U++ 20 2
CMUL 24 18 16 | VM2RF0_U-- 12 1 | CADD 8  24 8  | VM2RF1_U++ 14 2
CMUL 26 1c 1a | VM2RF0_U-- 16 1 | CADD e  26 e  | VM2RF1_U++ 18 2
CMUL 28 20 1e | VM2RF0_U-- 1a 1 | CADD c  28 c  | VM2RF1_U++ 1c 2
```

84% performance*

### Vector div

```
for (int i=0;i<len;i++) chess_loop_range(1,){
    *result++ = *dividend++ / *divisor++;
}
```

**Output ASM**

```
VFMUL 10 18 12 | VM2RF0_U++ 8 0 | - | VM2RF1_U++ 12 1
VFMUL e 10 16 | - | - | -
VFMUL a 8 18 | - | - | -
VINVSEED 18 12 | - | VFADD 16 6 -10 | -
VFMUL c a 16 | - | VFADD 14 6 -e | -
VFMUL 1a 14 c | VRF2M0_U++ 2 | - | -
```

100% performance*
Chess compiler for mAgicV

Resolved challenges – Compiler

- Improved phase coupling: register assignment and software pipelining
- Support of low-overhead loops, including software pipelining
- In-flight (aggressive) scheduling of memory loads

Resolved challenges – Assembler and linker

- Caching of nML analysis to speed up assembler/disassembler
- Linker to generate relocatable ELF executables, for code compression
 Agenda

- mAgicV and the Shapes multi-tile platform
- Chess/Checkers - a retargetable tool suite for ASIP design
- Targeting the Chess compiler to mAgicV
- VLIW challenges
  - Wide instruction word
  - Predicated execution
- Conclusions
Wide instruction word

- Instruction word contains orthogonal fields (issue slots)
- Restricted parallelism in source program may result in many NOP fields
  - E.g. vector-div on mAgic

```
VFMUL 10 18 12 | VM2RF0_U++ 8 0 | - | VM2RF1_U++ 12 1
VFMUL e 10 16 | - | - | -
VFMUL a 8 18 | - | - | -
VINVSEED 18 12 | - | VFADD 16 6 -10 | -
VFMUL c a 16 | - | VFADD 14 6 -e | -
VFMUL 1a 14 c | VRF2M0_U++ 2 | - | -
```

- How to reduce program code space?
  - Encoded instruction set → OK for ASIPs, less for gen.-purp. processor
  - Code compression by NOP elimination
  - Redundancy-based code compression
  - Variable-length instruction set

→ Loss of program space
→ Loss of power in instruction fetch
Code compression by NOP elimination

PC value

Instruction bits

Instruction A

Instruction B

format bits

NOP bits

opcode bits

“i”

“i+1”

 HW

- I/S contains both uncompressed (A, B) and compressed (AB) instructions
- After decoding AB, A and B are issued sequentially

Compiler

- Compiler replaces instruction sequence by compressed instruction
- Compiler favours compressible sequences

Processor model

- nML: describe uncompressed instructions + compression rule
Redundancy-based code compression

- Exploit redundancy in instruction word
  - Often proprietary, processor-specific schemes
  - mAgiCV’s “DyProDe” scheme: compression factor up to 3

- HW: decompression
  - Decompression logic, typically in separate pipeline stage(s)
  - HDL generator and ISS generator provide API to integrate custom decompression logic

- SW: compression
  - Custom compression routine, post-processing of ELF executable
  - Linker can generate relocatable ELF executables
  - Successfully applied to mAgiCV
Variable-length instruction set

- Processor model
  - Multiple instruction lengths can be specified in nML
  - Compiler favours short instructions

- Instruction fetch
  - No. of bits fetched may equal maximum instruction length, or less
  - Fetch buffer and stall cycles may be required
    - HDL generator and ISS generator provide API to specify buffer and stall behaviour

- Jump targets
  - If unaligned, need stall cycles to compose & issue target instruction
  - “Elongation” option in compiler
    - Replaces some short instructions by longer equivalent, to ensure aligned jump target
    - Avoids stall cycles
Agenda

- mAgiсV and the Shapes multi-tile platform
- Chess/Checkers - a retargetable tool suite for ASIP design
- Targeting the Chess compiler to mAgiсV
- VLIW challenges
  - Wide instruction word
  - Predicated execution
- Conclusions
**Predicated execution**

**What?**
- Operation executes or idles, depending on Boolean guard
- Avoids (long) latencies of conditional branches

**Tradeoff**
- Predicated operations consume instr. cycles

**Especially useful for**
- Conditional blocks with few operations
- VLIW: creates larger DFGs, more inherent ILP
Predicated execution

Compiler support

- Predication-vs.-jump selection
- Speculative execution of operations that cannot be predicated
- Code selection and reg. allocation done within conditional blocks, however taking limitations on predication into account
- Final if-else conversion (flattening) and scheduling

Extensions

- Vector predication (SIMD)
  - `vector_if, vector_else` construct in C code
- Nested predication (under development)
Agenda

- mAgicV and the Shapes multi-tile platform
- Chess/Checkers - a retargetable tool suite for ASIP design
- Targeting the Chess compiler to mAgicV
- VLIW challenges
  - Wide instruction word
  - Predicated execution
- Conclusions
Conclusions

- Chess/Checkers - a retargetable tool suite for ASIP design
- VLIW - increasingly popular architecture for massive parallel computing
- Chess compiler successfully targeted to mAgicV
- Compiler extensions for improved support of VLIWs