A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation

Benjamin Thielmann (Tech. Univ. Braunschweig)
Hagen Gädke-Lütjens (Tech. Univ. Braunschweig)
Andreas Koch (Tech. Univ. Darmstadt)
Content

- Introduction
- Modlib - a Modular Approach
- Performance Analysis
- Speculative Predicated Execution
- LMEM (Local Memory Infrastructure)
- Experimental Results
- Summary
Introduction

- Adaptive Computing System (ACS)
  - Central Processing Unit (CPU)
  - Reconfigurable Computation Unit (RCU)

- Characteristics
  - Compute-intense / frequent execution → Hardware (HW)
  - Less compute-intense / single execution / OS Calls → Software (SW)

- Application
  - High-performance computing
  - Low-power computing

- Programming Reconfigurable HW
  - HDL → optimum results @ high development costs
  - Synthesis from high-level languages → not mature / in development
Introduction

- COMRADE compiler
  - Covers most ANSI-C operators
  - Creates dynamically scheduled hardware
  - Efficient handling of control-intensive irregular code

- Implementation
  - Classic data flow model
  - Control flow predicating the execution of operators (optionally)
    - Activate Tokens (ATs) indicate the presence of data
    - Cancel Tokens (CTs) eliminate a miss-speculated computation

- Building blocks
  - Generic Library for Adaptive Computing Environments (GLACE)
GLACE

- GLACE characteristics
  - Hand-optimized operator library
  - Highly technology dependent

- Interface between compiler and module library
  - Request preplaced netlist
  - Obtain design parameters (area, throughput, latency, delay)

- Pros / Cons
  + Compact
  - Must be manually maintained
  - No pipelining
  - No unified interface (wrappers required)

- Hand-optimization worthwhile the effort?
  + Advanced synthesis tools
Wouldn't it be easier if we had primitives that...
- ... provided a universal handshaking interface
- … would be configurable to the application’s and compiler’s needs
- ... acted mostly autonomously

Modlib characteristics
- Primitives in portable RTL descriptions
- Extended operator support (ANSI-C)
- Easily extendable (IP block insertion support)
- Pipelining
- Dynamic and static scheduling models
- Speculative predicated execution (optionally)
- Unified flexible parameterized interface
Internal View (Generic Operator)

- Shift Register is used for...
  - Balancing pipeline paths with unequal latency
  - Retiming registers to improve synthesis results

- Output Queue is used to...
  - Decouple the execution from stalled data successors

- Token Queue is used to...
  - Reduce backpressure on the token delivering node

- Fully parameterized
Modlib Interface

- Actual Computation
  - Operands and result signal

- Dynamic Scheduling
  - Validation handshake for the operands
  - Validation handshake for the result

- Incoming Control Edge
  - Activate token intake via handshake
  - Cancel token intake via handshake

- Dynamic Cancel Tokens
  - Cancel forwarding handshake (for data)
  - Cancel forwarding handshake (for control)
Modlib Interface

- Actual Computation
  - Operands and result signal

- Dynamic Scheduling
  - Validation handshake for the operands
  - Validation handshake for the result

- Incoming Control Edge
  - Activate token intake via handshake
  - Cancel token intake via handshake

- Dynamic Cancel Tokens
  - Cancel forwarding handshake (for data)
  - Cancel forwarding handshake (for control)
Modlib Interface

- Actual Computation
  - Operands and result signal

- Dynamic Scheduling
  - Validation handshake for the operands
  - Validation handshake for the result

- Incoming Control Edge
  - Activate token intake via handshake
  - Cancel token intake via handshake

- Dynamic Cancel Tokens
  - Cancel forwarding handshake (for data)
  - Cancel forwarding handshake (for control)
Modlib Interface

- Actual Computation
  - Operands and result signal

- Dynamic Scheduling
  - Validation handshake for the operands
  - Validation handshake for the result

- Incoming Control Edge
  - Activate token intake via handshake
  - Cancel token intake via handshake

- Dynamic Cancel Tokens
  - Cancel forwarding handshake (for data)
  - Cancel forwarding handshake (for control)
Modlib Interface

- Actual Computation
  - Operands and result signal

- Dynamic Scheduling
  - Validation handshake for the operands
  - Validation handshake for the result

- Incoming Control Edge
  - Activate token intake via handshake
  - Cancel token intake via handshake

- Dynamic Cancel Tokens
  - Cancel forwarding handshake (for data)
  - Cancel forwarding handshake (for control)
Speculative Predicated Execution

- **Condition**
  - Data Predecessors are ready (ATs available)
  - Control Condition is not resolved yet

- **Control Flow Speculation**
  - Execute all alternatives in parallel
  - Delete the results of the miss-speculated branches (by insertion of CTs)

- **Advantages**
  - Faster than pure lenient execution

```
if (i < n)
a = x + y;
else
a = x / y;
```

![Diagram](attachment:image.png)
Static Cancel Tokens vs. Dynamic Cancel Tokens

(sCTs) Control condition result = 1

CT is stored and waits for incoming data

(dCTs) Control condition result = 1

CT is propagated along data predecessors
Performance Analysis
Modlib vs. GLACE I

Total Resources [Virtex 5 Slices]

- vecmult_10
- memcpy_8
- fcdf22_2
- sha_3
- susan
- gfMultiply

GLACE (dCTs)
Modlib (sCTs)
Modlib (dCTs)
Sequencer Resources [Virtex 5 Slices]

- vecmult_10
- memcpy_8
- fcdf22_2
- sha_3
- susan
- gfMultiply

GLACE (dCTs)
Modlib (sCTs)
Modlib (dCTs)
Modlib vs. GLACE III

Max. Frequency [MHz]

- vecmult_10
- memcpy_8
- fcdf22_2
- sha_3
- susan
- gfMultiply

GLACE (dCTs)
Modlib (sCTs)
Modlib (dCTs)
Modlib vs. GLACE IV

Cycletime [Cycles]

vecmult_10
memcpy_8
fcdf22_2
sha_3
susan
gfMultiply
synthetic_dct

GLACE (dCTs)
Modlib (sCTs)
Modlib (dCTs)
Example - Static CTs vs. Dynamic CTs

```
for (i=0; i<6; ++i)
  if (i != 5) {
    s += i;
  } else {
    a = i + 10;
    b = a*90+1+i;
    c = b / 89;
    s += c;
  }
```

(a) for (i=0; i<6 ; ++i)
  if ( i != 5 ) {
    s += i;
  } else {
    a = i  +  10;
    b = a*90+1+i;
    c = b  /  89;
    s += c;
  }

(b) LESS THAN

ADD
DIV
ADD
MULTIPLEXER

data edge control edge
Example - Static CTs vs. Dynamic CTs

(a)  
for (i=0; i<6; ++i) 
  if ( i != 5 ) { 
    s += i; 
  } else { 
    a = i + 10; 
    b = a*90+1+i; 
    c = b / 89; 
    s += c; 
  }

(b)  
data edge control edge
(a) (b)
==1 ==0
s
b
i 6
89
ADD
DIV
ADD
MULTIPLEXER
Example - Static CTs vs. Dynamic CTs

(a)

```c
for (i=0; i<6 ; ++i)
    if ( i != 5 ) {
        s += i;
    } else {
        a = i  +  10;
        b = a*90+1+i;
        c = b  /  89;
        s += c;
    }
```

(b)

```
# data edge
# control edge

---

```

Benjamin Thielmann

A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation
for (i=0; i<6; ++i) {
  if (i != 5) {
    s += i;
  } else {
    a = i + 10;
    b = a*90+1+i;
    c = b / 89;
    s += c;
  }
}
Example - Static CTs vs. Dynamic CTs

(a) for (i=0; i<6; ++i) 
   if (i != 5) {
     s += i;
   } else {
     a = i + 10;
     b = a*90+1+i;
     c = b / 89;
     s += c;
   }

(b) LESS THAN 

==1 ==0

s
b
i 6
89
ADD
DIV
ADD
MULTIPLEXER

data edge control edge
Example - Static CTs vs. Dynamic CTs

for (i=0; i<6; ++i) {
    if (i != 5) {
        s += i;
    } else {
        a = i + 10;
        b = a * 90 + 1 + i;
        c = b / 89;
        s += c;
    }
}

data edge control edge

(b) i 6
    LESS THAN

== 1 == 0

s

b

i 6

89

ADD

DIV

ADD

MULTIPLEXER

Benjamin Thielmann
A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation
2010-09-01
Discussion

- Static CTs vs. Dynamic CTs
  - Static CTs mostly more efficient than Dynamic CTs
    - Less HW resources
    - Higher frequency
    - Choice for low complexity code
  - Dynamic CTs able to improve cycle efficiency
    - More HW resources
    - Lower frequency
    - For high complexity code (no need to wait for canceled branches)
    - Highly dependent on algorithm
Discussion

- Modlib vs. GLACE
  - Modlib has slightly lower quality of result
  - Modlib is more flexible
  - Modlib is easier to maintain
  - Modlib supports pipelining
  - GLACE performance gain is not worthwhile the effort

- Performance optimization beyond arithmetic functions
  - Inevitable memory bottleneck
  - Take advantage of on chip memory resources
Local Memory Architecture (LMEM)

- Local Memory
  - BlockRAM implementation
  - Direct access via Power PC (PPC)
  - High bandwidth
  - Parallel access

- Local Paging Unit (LPU)
  - Configured via PPC
  - Initiates and supervises transfer
  - Notifies PPC on completion

- MARC Stream Port
  - Direct Memory Access (DMA)
Experimental Results
Benchmark LMEM / Pipelining

- MD5 (Message Digest Algorithm 5)
  - Cryptographic hash function
  - Computes a 128 bit hash

- Characteristics
  - Handles a message in 512 bit chunks
  - Each chunk processed in four rounds
  - Each round has 16 computation steps
  - Block chained algorithm

- Challenges
  - Difficult to pipeline  →  Data Parallelism (multiple messages)
  - Memory bandwidth  →  LMEM
- Loop unrolling
  - 64 function blocks
  - Three cycles each

- LMEM insertion
  - Messages in BlockRAM
  - Hash in BlockRAM

- Output Queue configuration
  - Avoid pipeline stalls
Latency vs. max. Frequency

- Commercial MD5 IP [Heliontech]: Latency 66 Clock Cycles, Frequency 174 MHz
- High Throughput MD5 [Y. Wang, HPCA]: Latency 34 Clock Cycles, Frequency 192 MHz
- MD5 sequential [COMRADE]: Latency 66.48 Clock Cycles, Frequency 144.86 MHz
- MD5 pipelined [COMRADE]: Latency 128.89 Clock Cycles, Frequency 192 MHz
Throughput and HW Resources

- Commercial MD5 IP [Heliontech]:
  - Throughput: 1,349 Mbps
  - Slices: 279

- High Troughput MD5 [Y. Wang, HPCA]:
  - Throughput: 32,035 Mbps
  - Slices: 11,957

- MD5 sequential [COMRADE]:
  - Throughput: 362 Mbps
  - Slices: 1,913

- MD5 pipelined [COMRADE]:
  - Throughput: 20,642 Mbps
  - Slices: 5,631
Throughput / HW Resource

- Commercial MD5 IP [Heliontech]: 4.84 Mbps/Virtex-5 Slice
- High Troughput MD5 [Y. Wang, HPCA]: 2.68 Mbps/Virtex-5 Slice
- MD5 sequential [COMRADE]: 0.19 Mbps/Virtex-5 Slice
- MD5 pipelined [COMRADE]: 3.67 Mbps/Virtex-5 Slice
MD5 Conclusion

- High Performance
  - Throughput comparable to highly optimized manual implementations
  - Full pipelining
  - LMEM access

- High Efficiency
  - Excellent performance per area
  - Efficient pipelining (queues)

- High Latency
  - Due to three cycle function pipeline
  - Improvement possible
Summary

- Modlib has advantages over GLACE
  - Similar quality of result
  - Enhanced application area
    - Target technology independent
    - Pipelining support
    - Support for different scheduling models
    - Support for different execution models

- Dynamic CTs can improve cycle count
  - Highly dependent on algorithm
  - More costly than static CTs

- LMEM
  - Integrates well in the architecture (lightweight)
  - Significantly speeds up bandwidth intensive applications
The Modlib and its documentation is available
@ http://www.esa.cs.tu-darmstadt.de/modlib
Benchmarks Modlib vs. GLACE

- Vecmult 10
  - vector multiplication, unrolled x10 [Synthetic]

- Memcpy 8
  - array copy, unrolled x8 [Synthetic]

- Fcdf22 2
  - 2-D wavelet transform for image compression [Kumar, Pires]

- Sha 3
  - Secure Hash Algorithm [CHStone]

- Susan
  - Edge detection on gray scale image [MiBench]

- GfMultiply
  - Pegwit elliptic curve cryptography application.