Challenges of mixed-width vector code generation and static scheduling in LLVM (for VLIW Architectures)

*Erkan Diken, **Pierre-Andre Saulais, ***Martin J. O'Riordan
(*) Eindhoven University of Technology, Eindhoven
(**) Codeplay Software, Edinburgh
(***) Movidius Ltd., Dublin

Euro LLVM 2015
London, England

April 14, 2015

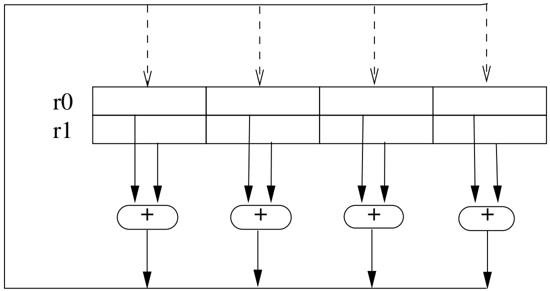
PART I

"Background: SIMD / Vector Instruction / VLIW"

Erkan Diken (e.diken@tue.nl)

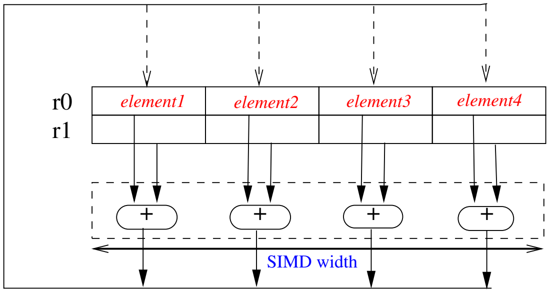
SIMD

- Single-instruction multiple-data (SIMD) hardware
- The same operation on multiple data lanes (in parallel)



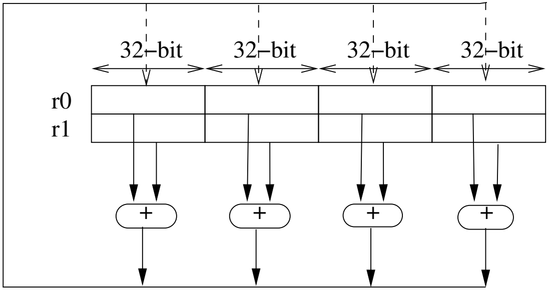
SIMD

- SIMD (vector) width
- Vector data = $< \#ofelements > x < elementtype >$



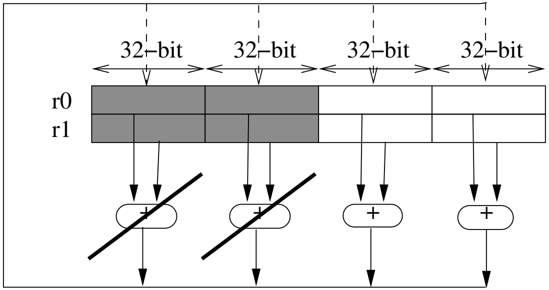
128-bit Vector Instruction

- ADD.128 r0, r0, r1
- 128-bit = (4 x i32, 4 x f32, 8 x i16, 8 x f16, 16 x i8 ...)



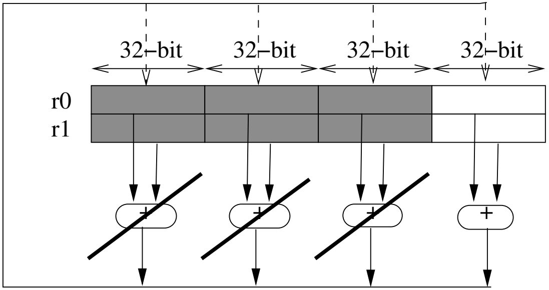
64-BIT VECTOR INSTRUCTION

- ADD.64 r0, r0, r1
- 64-bit = (2 x i32, 2 x f32, 4 x i16, 4 x f16, 8 x i8 ...)



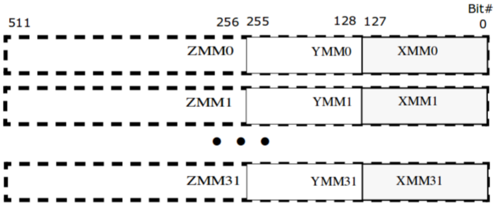
32-bit Vector Instruction

- ADD.32 r0, r0, r1
- 32-bit = (2 x i16, 2 x f16, 4 x i8 ...)



EXAMPLE: INTEL AVX-512 ARCHITECTURE

- The vector processing unit (VPU) in Xeon Phi coprocessor
- ZMM (512-bit), YMM (256-bit), XMM (128-bit) registers



References: ”Intel Architecture Instruction Set Extensions Programming Reference”, ”Intel Xeon Phi Coprocessor Vector Microarchitecture”

OBSERVATIONS

- SIMD units get wider and wider
- When a part of SIMD unit is not used for a shorter vector processing:
  1. Ignore the results of some SIMD lanes through masking
  2. Disable SIMD lanes through hardware reconfiguration (e.g. clock/power gating)
- Both result in performance and/or energy waste
- Can we:
  1. Introduce more SIMD heterogeneity into processor (and)
  2. Tackle the introduced complexity (problem) in the compiler

VLIW WITH MULTIPLE NATIVE SIMD WIDTHS

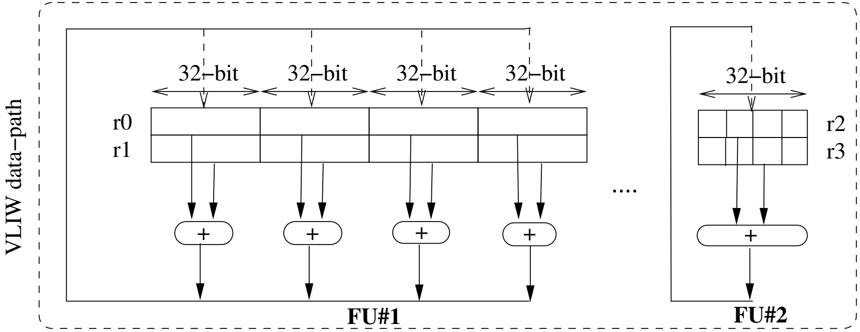


Figure : VLIW data-path with 128-bit and 32-bit native SIMD widths

VLIW with multiple native SIMD widths

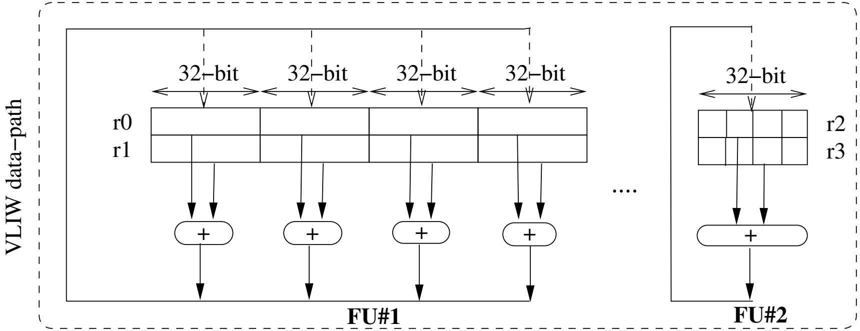


Figure : VLIW data-path with 128-bit and 32-bit native SIMD widths

Mixed-width vector code:

- FU#1.ADD.128 r0, r0, r1 || FU#2.ADD.32 r2, r2, r3
- FU#1.ADD.64 r0, r0, r1 || FU#2.ADD.32 r2, r2, r3
- FU#1.ADD.32 r0, r0, r1 || FU#2.ADD.32 r2, r2, r3

CHALLENGES OF ...

1. Mixed-width vector code generation support (and)
2. Static scheduling

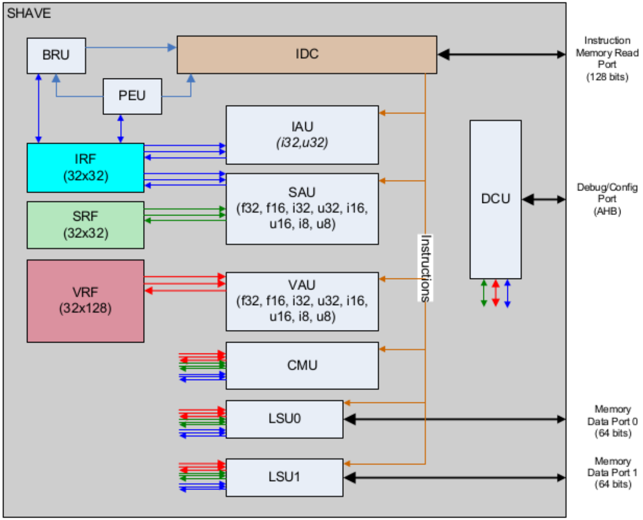
in LLVM for such VLIW architectures

PART II

"Mixed-width vector code generation in LLVM for VLIW Architectures"

Erkan Diken (e.diken@tue.nl)

SHAVE Vector Processor*


SHAVE
BRU
IDC
PEU
IAU
(i32,u32)
IRF
(32x32)
SAU
(f32, f16, i32, u32, i16, u16, i8, u8)
SRF
(32x32)
DCU
Instructions
VRF
(32x128)
VAU
(f32, f16, i32, u32, i16, u16, i8, u8)
CMU
LSU0
LSU1
Instruction Memory Read Port (128 bits)
Debug/Config Port (AHB)
Memory Data Port 0 (64 bits)
Memory Data Port 1 (64 bits)

(*) SHAVE is part of the Movidius Myriad 1 and Myriad 2 Vision Processor Platform of Movidius Ltd. (www.movidius.com)

More Details

Architecture:

- VAU is designed to support 128-bit vector arithmetic
- VAU accepts operands from 32 x 128 VRF registers
- SAU is designed to support 32-bit vector arithmetic
- SAU accepts operands from 32 x 32 IRF and SRF registers

More Details

Architecture:

- VAU is designed to support 128-bit vector arithmetic
- VAU accepts operands from 32 x 128 VRF registers
- SAU is designed to support 32-bit vector arithmetic
- SAU accepts operands from 32 x 32 IRF and SRF registers

Compiler:

- The original compiler supports 128-bit and 64-bit vector code generation.
- 128-bit legal vector types: 16 x i8, 8 x i16, 4 x i32, 8 x f16, 4 x f32
- 64-bit legal vector types: 8 x i8, 4 x i16, 4 x f16
- What about 32-bit vector types: 4 x i8, 2 x i16, 2 x f16 ?

MORE DETAILS

Architecture:

- VAU is designed to support 128-bit vector arithmetic
- VAU accepts operands from 32 x 128 VRF registers
- SAU is designed to support 32-bit vector arithmetic
- SAU accepts operands from 32 x 32 IRF and SRF registers

Compiler:

- The original compiler supports 128-bit and 64-bit vector code generation.
- 128-bit legal vector types: 16 x i8, 8 x i16, 4 x i32, 8 x f16, 4 x f32
- 64-bit legal vector types: 8 x i8, 4 x i16, 4 x f16
- What about 32-bit vector types: 4 x i8, 2 x i16, 2 x f16 ?

Contribution:

- Implementing 32-bit vector code generation for SAU units in the compiler back-end

EXAMPLE: MIXED-WIDTH VECTOR CODE

Listing 1: LLVM IR code with two different vector types

```
define <4 x i8> @main(<4 x i8> %a, <4 x i8> %b,
                      <8 x i8> %x, <8 x i8> %y,
                      <8 x i8>* %zptr){
entry:
        %c = add <4 x i8> %a, %b
        %z = add <8 x i8> %x, %y
        store <8 x i8> %z, <8 x i8>* %zptr
        ret <4 x i8> %c
}
```

EXAMPLE: MIXED-WIDTH VECTOR CODE

Listing 3: LLVM IR code with two different vector types

```
define <4 x i8> @main(<4 x i8> %a, <4 x i8> %b,
                      <8 x i8> %x, <8 x i8> %y,
                      <8 x i8>* %zptr){
entry:
        %c = add <4 x i8> %a, %b
        %z = add <8 x i8> %x, %y
        store <8 x i8> %z, <8 x i8>* %zptr
        ret <4 x i8> %c
}
```

Listing 4: Mixed-width vector assembly code

```
main:
        BRU.JMP i30
        CMU.CPVI.x32 i9 v22.0
        CMU.CPVI.x32 i10 v23.0
        VAU.ADD.i8 v15 v21 v20       //64-bit add (8 x i8)
            || SAU.ADD.i8 i10 i10 i9 //32-bit add (4 x i8)
        NOP
        CMU.CPIV.x32 v23.0 i10
            || LSU1.ST64.l v15 i18
```

Implementation Details

- Type legalization: New legal vector types for the target: 4 x i8, 2 x i16, 2 x f16

Implementation Details

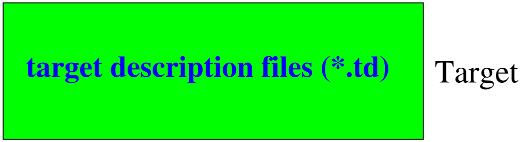
- Type legalization: New legal vector types for the target: 4 x i8, 2 x i16, 2 x f16
- Register class association: Which register file class is available for which vector type
  - SRF: 2 x f16
  - IRF: 4 x i8, 2 x i16
  - Quarter of VRF: 4 x i8, 2 x i16, 2 x f16

IMPLEMENTATION DETAILS

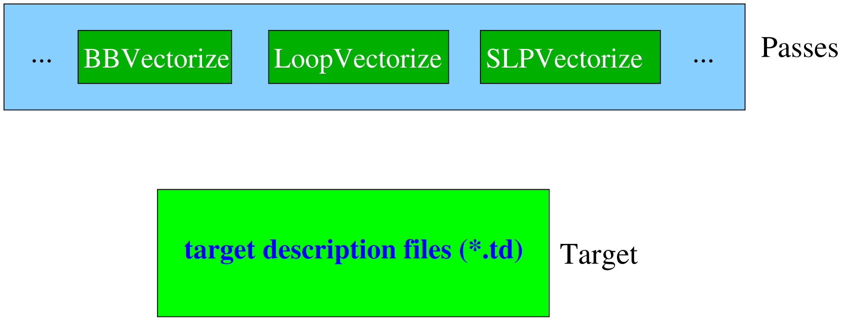
- Type legalization: New legal vector types for the target: 4 x i8, 2 x i16, 2 x f16
- Register class association: Which register file class is available for which vector type
  - SRF: 2 x f16
  - IRF: 4 x i8, 2 x i16
  - Quarter of VRF: 4 x i8, 2 x i16, 2 x f16
- Operation lowering for ISel: Add records to back-end for matching IR operations with MI
  - Natively supported operations: load/store, add, sub, mul, shift etc.
  - Custom lowering, expansion, promotion

For more implementation details: ”moviCompile: An LLVM based compiler for heterogeneous SIMD code generation” FOSDEM’15

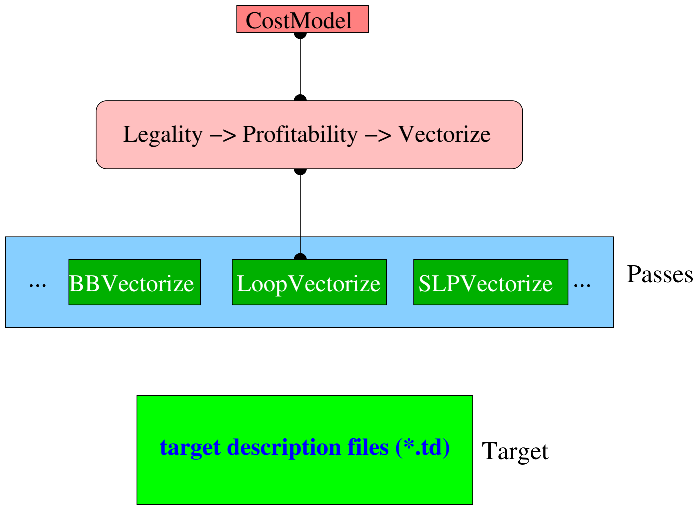
Overall Picture (Target)



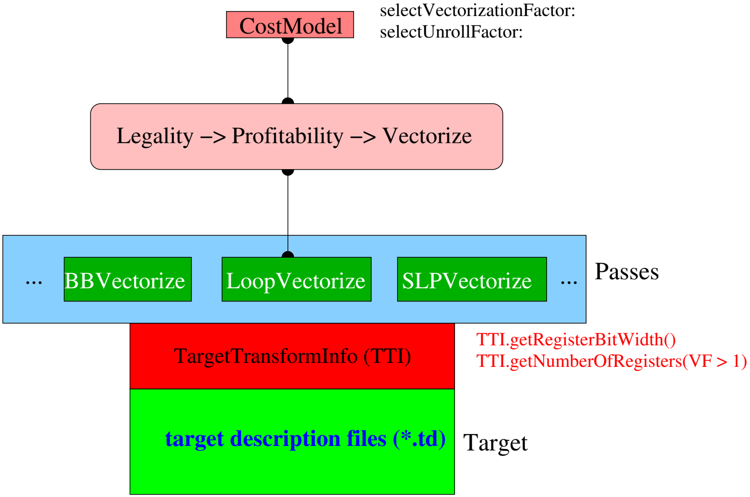
OVERALL PICTURE (TARGET, PASSES)



OVERALL PICTURE (TARGET, PASSES, COST MODEL)


CostModel
Legality –> Profitability –> Vectorize
...
BBVectorize
LoopVectorize
SLPVectorize
...
Passes
target description files (*.td)
Target

Overall Picture (Target, Passes, Cost Model, TTI)


CostModel
selectVectorizationFactor:
selectUnrollFactor:
Legality –> Profitability –> Vectorize
...
BBVectorize
LoopVectorize
SLPVectorize
...
Passes
TargetTransformInfo (TTI)
TTI.getRegisterBitWidth()
TTI.getNumberOfRegisters(VF > 1)
target description files (*.td)
Target

TARGET TRANSFORM INFO (TTI)

Listing 5: SHAVE

```
unsigned SHAVETTI::getNumberOfRegisters(bool Vector) const {
  if (Vector) {
    // 32 VRF registers.
    return 32;
  }

  if (ST->isMyriad1()) {
    // 32 IRF registers, 32 SRF registers.
    return 64;
  }

  // 32 IRF registers.
  return 32;
}

unsigned SHAVETTI::getRegisterBitWidth(bool Vector) const {
  if (Vector) {
    // 128-bit VRF registers.
    return 128;
  }

  // 32-bit IRF/SRF registers.
  return 32;
}
```

Target Transform Info (TTI)

Listing 6: X86

```
unsigned X86TTIImpl::getNumberOfRegisters(bool Vector) {
  if (Vector && !ST->hasSSE1())
    return 0;

  if (ST->is64Bit()) {
    if (Vector && ST->hasAVX512())
      return 32;
    return 16;
  }
  return 8;
}

unsigned X86TTIImpl::getRegisterBitWidth(bool Vector) {
  if (Vector) {
    if (ST->hasAVX512()) return 512;
    if (ST->hasAVX()) return 256;
    if (ST->hasSSE1()) return 128;
    return 0;
  }

  if (ST->is64Bit())
    return 64;
  return 32;
}
```

LESSONS TAKEN AND DISCUSSION POINTS

- TTI reports only one vector-width for the target, however:
  - Returning a list/set of supported vector-widths
  - Increases flexibility for mixed-width vector code optimisations

LESSONS TAKEN AND DISCUSSION POINTS

- TTI reports only one vector-width for the target, however:
  - Returning a list/set of supported vector-widths
  - Increases flexibility for mixed-width vector code optimisations
- Even though compiler back-end supports mixed-width vector code generation, LLVM will always:
  - Place the 32-bit vectors in the 32-bit vector registers
  - Place 128/64-bit vectors in the 128-bit vector registers

Lessons taken and discussion points

- TTI reports only one vector-width for the target, however:
  - Returning a list/set of supported vector-widths
  - Increases flexibility for mixed-width vector code optimisations
- Even though compiler back-end supports mixed-width vector code generation, LLVM will always:
  - Place the 32-bit vectors in the 32-bit vector registers
  - Place 128/64-bit vectors in the 128-bit vector registers
  - Affinity between a vector-type and a particular register-class
  - Vector type could be associated with a set of register classes, but with a preferred affinity to one class

LESSONS TAKEN AND DISCUSSION POINTS

- TTI reports only one vector-width for the target, however:
  - Returning a list/set of supported vector-widths
  - Increases flexibility for mixed-width vector code optimisations
- Even though compiler back-end supports mixed-width vector code generation, LLVM will always:
  - Place the 32-bit vectors in the 32-bit vector registers
  - Place 128/64-bit vectors in the 128-bit vector registers
  - Affinity between a vector-type and a particular register-class
  - Vector type could be associated with a set of register classes, but with a preferred affinity to one class
- This would allow operations on the shorter vector type to migrate to a larger vector register type
  - In case of register or FU pressure made such migration produce better code
  - This is especially true in a VLIW architecture where two or more FUs can perform the same task

PART III

”Static Scheduling in LLVM for VLIW Architectures”

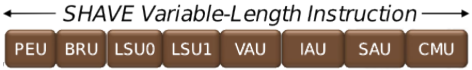
Pierre-Andre Saulais (pierre-andre@codeplay.com)

1. Scheduling challenges with VLIW architectures

- Important optimisations
- Scheduling hazards
- Example schedule

Important optimisations

- Maximising Instruction-Level Parallelism
  - VLIW processors usually have many functional units
  - Keep FUs as busy as possible
  - *Software pipelining and loop unrolling can have a huge impact*
- Filling branch delay slots
  - Instructions can be executed while a branch is 'pending'
  - *Fill these slots first using bottom-up scheduling*
- Breaking dependencies between instructions
  - Dependencies prevent instructions from being executed in parallel
  - *Rename registers*
  - *Perform early scheduling, before register allocation*



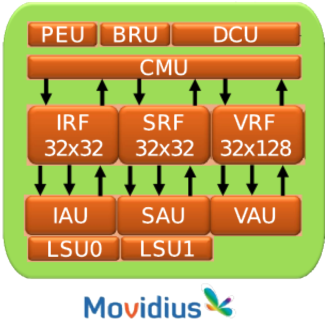
Movidius

SCHEDULING HAZARDS

- Static scheduling for VLIW architectures
  - **Not** just to achieve optimal performance
  - **Required** for correct execution
- No instruction interlocking / pipeline bubbles
  - To reduce power consumption
  - Can lead to conflicts between instructions (i.e. hazards)
  - Hazards must be handled by the scheduler
- Common hazards to avoid
  - Operand not ready
  - Port conflicts

SCHEDULING HAZARDS

- Operand not ready
  - Instruction latency must be taken into account
  - Otherwise the previous register value will be used
  - *Enforce 'cycle-distance' dependencies between instructions*
- Register port conflicts
  - Cannot write two values using the same port in a given cycle
  - One value will 'win' and be written to both registers
  - *Track conflicts and schedule instructions in different cycles*



EXAMPLE SCHEDULE

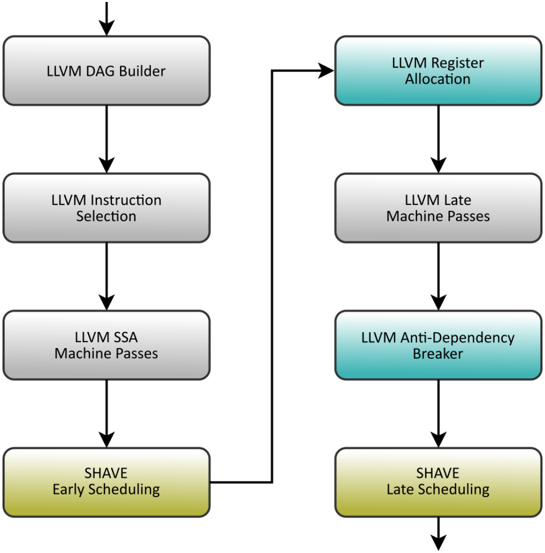
- Cycle table
  - Instruction executed by each FU
  - For each cycle in a basic block or function

#	CMU	IAU	VAU	LSU0	LSU1	PEU	BRU
0				LDIH i17 0x0b0a	LDIL i17 0x0908		
1	CPIV v12.0 i17			LDIH i16 0x0f0e	LDIL i16 0x0d0c4		
2	CPIV v13.0 i16			LDIH i9 0x0706	LDIL i9 0x0504		
3	CPIV v14.0 i9			LDIH i10 0x0302	LDIL i10 0x0100		
4	CPIV v15.0 i10						
5	CP.i8.i32 v12						
6	CP.i8.i32 v13						
7	CP.i8.i32 v14						
8	CP.i8.i32 v15	ADD i8 i18 32		LDIL i10 0x0100	LDIL i9 0		
9	CPIVR v11 i9	ADD i9 i9 16					
10	CMII i9 i10		ADD v22 v11 v13				
11			ADD v21 v11 v12			PCXX NEQ	BRA #9
12			ADD v20 v11 v14				delay slot
13			ADD v10 v11 v15				del. slot 2
14				ST.l v21 i8	STO.l v22 i8 16		del. slot 3
15				STO.l v10 i8 -32	STO.l v20 i8 -16		del. slot 4
16				STO.h v21 i8 8	STO.h v22 i8 24		del. slot 5
17		ADD i8 i8 64		STO.h v10 i8 -24	STO.h v20 i8 -8		del. slot 6

2. IMPLEMENTATION WITHIN LLVM

- Scheduling passes in the backend
- SHAVE MI scheduler
- SHAVE hazard recognizer
- Moving instructions across FUs

Scheduling passes in the backend



SHAVE MI SCHEDULER

- Based on the list scheduling algorithm
  - Assigns costs to each instruction
  - Schedules instructions one by one, with decreasing cost
- Enforces dependencies between instructions
  - To avoid ’operand not ready’ hazards
  - Consumes ISA scheduling information (e.g. latency)
- Bundles multiple instructions into a cycle, to maximise ILP
  - Moves instructions across FUs as needed
  - Uses a hazard recognizer to avoid conflicts

SCHEDULING INFORMATION

- Describes which resources are used by an instruction and when
  - Per-operand latency
  - Per-operand port list
  - Defined with TableGen (using the Processor Resource Model)
- Used extensively in the scheduler
  - When creating the instruction dependency graph
  - When packing instructions into bundles

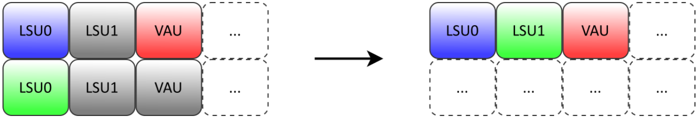
```
struct SHAVEResUse {
  unsigned Latency; // Cycle where the resources are used
  BitField ResourceMask; // FUs and ports
};

// Resources used by MI for each cycle of operation
bool GetSchedResources(MachineInstr *MI, vector<SHAVEResUse> &Uses);
```

SHAVE HAZARD RECOGNIZER

- Answers queries from the scheduler
  - "Can this instruction be scheduled in that cycle?"
  - "Will these two instructions conflict?"
- Keeps track of already scheduled instructions
  - Scoreboard approach
  - Cycle table describes which resources (FUs, ports, ...) are used
  - Two instructions in the same cycle cannot share resources

MOVING INSTRUCTIONS ACROSS FUS

- Some FUs have overlapping functionality
  - Memory instructions: LSU0 ↔ LSU1
  - Some arithmetic instructions: IAU ↔ SAU
  - Some copy instructions: CMU.CP* ↔ LSU.CP
- Exploit this overlap to improve ILP
- Transform an instruction to another equivalent instruction
  - Mutate instruction in-place
  - Calculate new scheduling cost
  - Revert changes if no scheduling improvement



FU PARAMETERISATION

- Each LSU instruction has a 'FU' operand
    - Avoids duplicating instruction definitions in TableGen
    - Scheduling information now depends on the value of this operand
    - Default value, no need to specify it in DAG patterns
- Simplifies instruction mutation
    - Change the operand value to move instruction across FUs
- Makes scheduling policy easier

```
class FUnitOp<int num> : OperandWithDefaultOps<i8, (ops (i8 num))> {
    let PrintMethod = "printFUnitOperand";
}

def lsu_id : FUnitOp<8>; // Defaults to LSU1

class SHAVE_LSUInstr<dag OOL, dag IOL, string asmstr, list<dag> pat> :
    SHAVEInstr<OOL, !con(IOL, (ins lsu_id:$funit)),
               !strconcat("$funit", asmstr), LSU1> {
    let Pattern = pat;
}
```

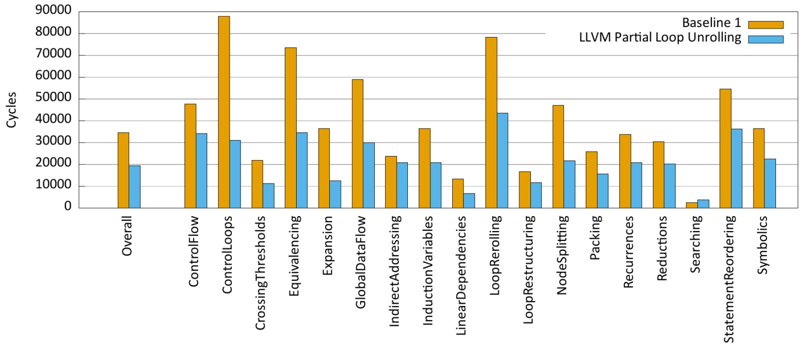
3. OPTIMISATION RESULTS

- TSVC Benchmark
- Optimisations
  - LLVM Partial Loop Unrolling
  - Branch Delay Slot Filling
  - Unified Early/late Scheduling
- Overall results

TSVC Benchmark

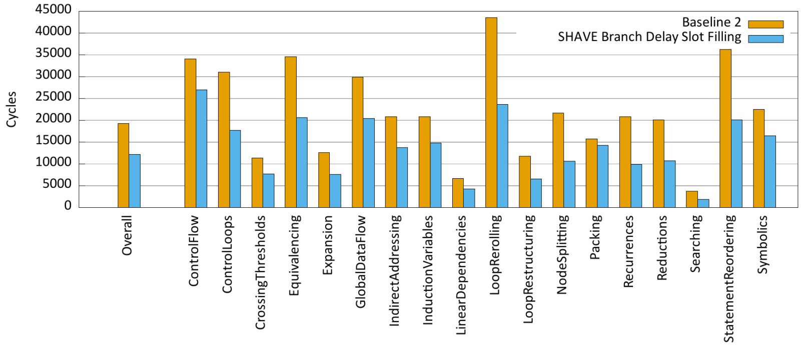
- Designed to exercise vectorisation in a compiler
- Each test is a loop
    - Tests are grouped into categories
    - Each category exercises a different kind of loop pattern
- Result caveat
    - No per-category analysis done here
    - Per-category results included to show that optimisation impact differs between patterns

LLVM Partial Loop Unrolling



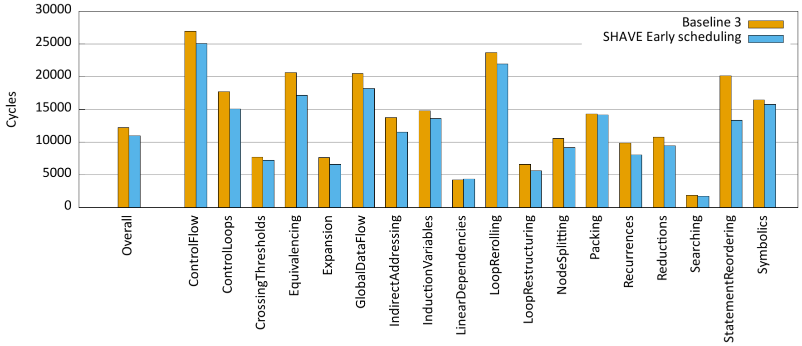
- Runs multiple loop iterations 'at a time'
  - Introduces opportunities for ILP between loop iterations
  - Many FUs: significant improvements on VLIW architectures
- Pass `-mllvm -unroll-allow-partial` to clang
  - Requires a cost model for your target

Branch Delay Slot Filling



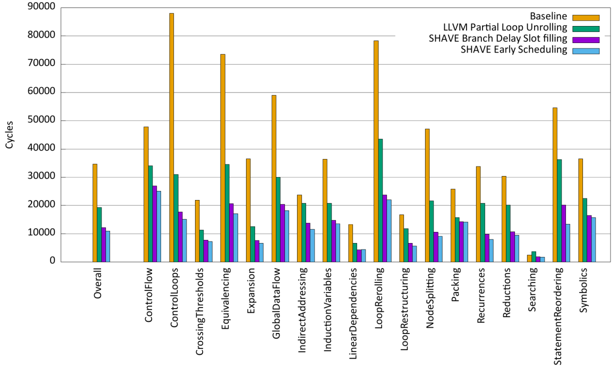
- Branches have 6 delay slots on this ISA
  - Delay slots are filled with `NOP`s
  - Cost applies to every iteration of a loop
- This optimisation reduces the cost of branching
  - Biggest impact on small loops with high number of iterations

Unified Early/late Scheduling



- Uses the same scheduler for early and late scheduling
  - Tends to cluster higher-latency instructions like loads
  - Gives the register allocator a better idea of register usage
- Works well in combination with loop unrolling
  - Avoids dependencies between iterations, improving ILP

Overall results



- Observed geomean speedup on TSVC tests:
  - LLVM Partial Loop Unrolling: 1.793x
  - Branch Delay Slot Filling: 1.578x
  - Unified Early/late Scheduling: 1.114x
  - Overall: 3.151x

Thank you for your attention!

Questions ?

Contacts:
Erkan Diken: e.diken@tue.nl
Pierre-Andre Saulais: pierre-andre@codeplay.com
Martin J. O'Riordan: martin.oriordan@movidius.com

Technische Universiteit **Eindhoven** University of Technology