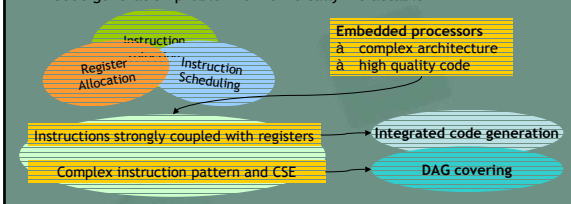
Cases Studies on Automatic Extraction of Target-specific Architectural Parameters in Complex code generation

Seoul National University
So&R Laboratoty
Yunheung Paek, Minuk Ahn, Soonho Lee

Contents

Motivation in complex code generation

• Code generation problem is intrinsically intractable



Reduce compilation time using various heuristics
à slashing DAG search space

Heuristics are target specific à XR² architectural description language

Motivation in complex code generation

• XR² gives a way of characterizing target architecture

Storage
Addressing modes
Instructions
Structure of XR²

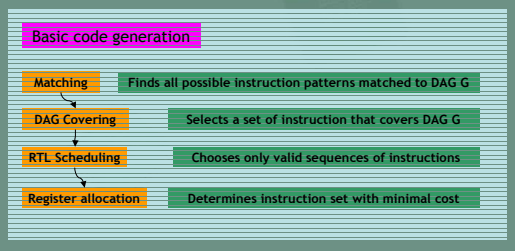
Ex) a part of ARM9 description

```
HiISA ARM9x {
        Storage_Set arm9x_Storage
        AddreMode_Set arm9x_AddrMode
        Instruction_Set arm9x_InstructionSet
}
Storage arm9x_Storage {
        Memory dmem {
        }
        RegisterF
        }
}
AddrMode ImmediateOffset : AddressingMode1
operands {
}
}
}
instruction add_dst_src1_imme : ADD_Rd_Rn_Oprnd2 {
        operands {
                GRClass Rd
                GRClass Rn
                IMM(32) imm32
                CPSR cpsr
        }
        syntax {
                "ADD " Rd ", " Rn ", #" imm32
        }
        action {
                Rd = add(Rn, imm32, cpsr);
        }
}
```


Contents

Basic code generation : naïve approach

• Ignore compilation time and focus only on finding optimal code for a given basic block

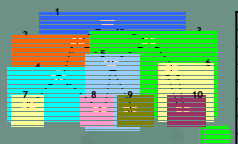


Basic code generation : Matching

Model target processor with a set of RTLs using XR² and transform every RT into tree

↓

Match above RT trees against subgraphs of DAG using signature for fast processing



windows	signature
1	+ + + << << +
2	* + << i * * i
3	* << + * i * *
4	* i * _ _ _ _
5	<< * i _ _ _ _
6	* * _ _ _ _
7	* _ _ _ _ _ _
8	* _ _ _ _ _ _
9	* _ _ _ _ _ _
10	* _ _ _ _ _ _

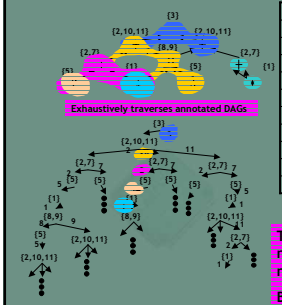
No.	RT	signature
1	mul	*rr
2	add	+rr
3	sub	-rr
4	load	M
5	loadi	i
6	negate	-
7	mac	+r*__rr +*rrr__
8	shift	<<rr
9	shift-imm	<<ri
10	add-shift	+r<<__rr +<<rrr__
11	add-shift-imm	+r<<__ri +<<rri

To minimize overhead of tree walk RT tree is converted into signature

- Shared nodes may appear multiple times

Basic code generation : DAG covering

- Enumerate every possible cover by exhaustively searching DAG



Exhaustively traverses annotated DAGs

windows	signature
1	+ + + << << +
2	* + << i * * i
3	* << + * i * *
4	* i * _ _ _ _
5	<< * i _ _ _ _
6	* * _ _ _ _
7	* _ _ _ _ _ _
8	* _ _ _ _ _ _
9	* _ _ _ _ _ _
10	* _ _ _ _ _ _

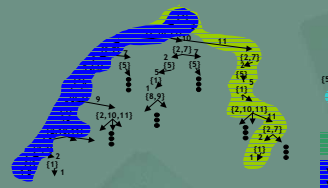
No.	RT	signature
1	mul	*rr
2	add	+rr
3	sub	-rr
4	load	M
5	loadi	i
6	negate	-
7	mac	+r*__rr +*rrr__
8	shift	<<rr
9	shift-imm	<<ri
10	add-shift	+r<<__rr +<<rrr__
11	add-shift-imm	+r<<__ri +<<rri

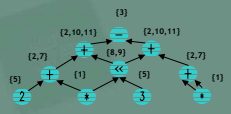
To avoid generating code for shared node more than once, we mark every node we meet.

But it is such a huge search space

Basic code generation : RTL scheduling

- Compute the minimal cost instructions for the selected DAG cover





RTL codes from DAG covering

$v1_{AB} = 2$
$v2_{AB} = v1_{AB} + x1_M * x2_M$
$v3_{AB} = x1_M + x2_M$
$v4_{AB} = v2_{AB} + v3_{AB} << 3$
$v5_{AB} = x3_M * x4_M$
$v6_{AB} = v5_{AB} + v5_{AB}$
$v7_{AB} = v6_{AB} + v3_{AB} << 3$
$v10_{AB} = v4_{AB} - v7_{AB}$

RTL scheduling construct a valid sequence of RTLs by mapping the RTLs in the cover to time slot {1,…,n} without violating data dependence constraints

Basic code generation : Register allocation

Register allocation assigns registers to the operands of scheduled RTLs

RTL codes from DAG covering

$v1_{AB} = 2$
$v2_{AB} = v1_{AB} + x1_M * x2_M$
$v3_{AB} = x1_M * x2_M$
$v4_{AB} = v2_{AB} + v3_{AB} << 3$
$v5_{AB} = x3_M * x4_M$
$v6_{AB} = v5_{AB} + v5_{AB}$
$v7_{AB} = v6_{AB} + v3_{AB} << 3$
$v10_{AB} = v4_{AB} - v7_{AB}$

$v1_A = 2$
$v2_A = v1_A + x1_M * x2_M$
$v3_B = x1_M * x2_M$
$v3_M = v3_B$
$v4_A = v2_A + v3_M << 3$
$v5_B = x3_M * x4_M$
$v5_M = v5_B$
SL[0] = $v4_A$
$v6_A = v5_B + v5_M$
$v3_B = v3_M$
$v7_A = v6_A + v3_B << 3$
$v4_B$ = SL[0]
$v7_M = v7_A$
$v8_B = v4_B - v7_M$

Basic code generation : Determining optimal code

- Exhaustive comparison

Minimal cost schedule is determined only after traversing all the paths in a search tree considering following cost function

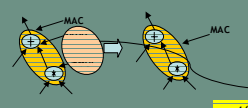
$cost(code_{RTL}) = \sum c(r_i) + \sum c(m_i) + \sum c(s_i)$ — RTL cost | Extra move cost | Spill cost

Contents

Target specific speeding-up code generation

- Concept
  - Faster compilation with little degradation of code quality
- Basic strategy
  - To reduce the number of covers by slashing the DAG search space
  - To reduce the running time of RTL scheduling and register allocation
- Clustering
  - Clustering would eliminate a half of the search space below the clustered node according to cost function
  - When we have several candidates of DAG covers provided by XR² (architecture description language), we determine the cover with minimal cost, using the cost function previously shown

Target specific speeding-up code generation
Clustering



This part of search space need not to be considered!

$cost(code_{RTL}) = \sum c(r_i) + \sum c(m_i) + \sum c(s_i)$

Major component

$c(r_{MAC}) < c(r_{ADD}) + c(r_{MUL})$

$c(m_{MAC}) >= c(m_{ADD+MUL})$

$c(s_{MAC}) <= c(s_{ADD+MUL})$

In most architecture, this condition holds

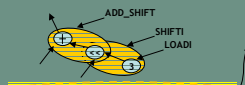
Most ALU instructions operate on the same functional unit using the registers in the same class

Homogeneous architectures use the same registers used in MAC for ADD and MUL

Heterogeneous architectures require dedicated registers for different instructions, so they use more registers and increase register spill

Target specific speeding-up code generation
Clustering

- Another case of clustering



$c(r_{ADD\_SHIFT}) + c(r_{LOADI}) = c(r_{ADD}) + c(r_{SHIFTI})$

$c(m_{ADD\_SHIFT+LOADI}) = c(m_{ADD+SHIFTI})$

$c(s_{ADD\_SHIFT+LOADI}) = 2/3 \sim c(s_{ADD+SHIFTI})$

$cost(code_{RTL}) = \sum c(r_i) + \sum c(m_i) + \sum c(s_i)$

Composite of fewer cycle clusters are better.

In many architecture each cost equals.

For homogeneous and most heterogeneous machines, each cost equals

For homogeneous machines, each cost equals.

But heterogeneous machines, it varies on the type of instructions.

For homogeneous machines à first condition.

For heterogeneous machines à third condition.

Slashing the search space à reduce compilation time

Determination of cluster depends on

- Architectural feature provided by XR²
- Cost function chosen by compiler developer.

Target specific speeding-up code generation
Register allocation

- Only register allocation for heterogeneous register architecture is considered.
- A few choices in allocation
- only a small number of registe
- fast on-chip software-controllable memory
- The speed of register allocator is very important

One cycle latency : spill cost can be ignored

Issue here : Shared Node

Localization : split the life span of a register assigned to shared node

Contents

- Motivation in complex code generations
- Basic code generation
- Target specific speeding-up code generation
- Experimental result
- Conclusion

Experimental result : ARM9

- Basic blocks from MediaBench and DSPstone

The growth rate of compilation time with the increase of code size

	Tried DAG cover		Compile time(sec)		Execution time(cycle)		Code size(words)	
code	basic	improved	basic	improved	basic	improved	basic	Improved
convolution	21952	14	209	0.1	65	65	11	11
Fir	115112	84	∞	0.66	132	114	18	16
lms block 1	129802	84	∞	0.63	124	110	17	15
lms block 2	100128	14	∞	0.41	111	98	14	13
lms block 3	18816	6	342	0.01	68	68	11	11
n_complex_updates	61234	16	∞	0.55	466	328	67	47
adcpm init	96002	5	∞	0.13	224	149	32	22
idctrow 1	97664	6	∞	0.15	241	169	38	26
idctrow 2	10231	18200	∞	∞	1452	964	274	172

Experimental result : TI320c54x

	Tried DAG cover		Compile time(sec)		Execution time(cycle)		Code size(words)	
code	basic	improved	basic	improved	basic	improved	basic	improved
convolution	174500	432	∞	2	46	29	28	15
Fir	215100	224	∞	2	74	63	39	21
lms block 1	159700	192	∞	12	67	46	40	20
lms block 2	147300	128	∞	0.8	57	28	34	14
lms block 3	224200	80	∞	1	49	37	28	16
n_complex_updates	126700	78400	∞	∞	204	145	131	75
adcpm init	140900	16	∞	0.2	141	45	73	35
idctrow 1	67400	79600	∞	∞	152	66	92	36
idctrow 2	16700	19600	∞	∞	603	450	462	270

- In all programs, heuristics reduce the compilation time significantly
- TI320c54x has CISC style instruction set which causes the increase of search space and compilation time

Contents

Conclusion

- Our work is

Integrated code generation based on DAG covering for optimal code → Too much computational overhead

Our strategy :
- Reduce the number of covers by slashing the DAG search space
  Clustering
- Reduce the running time of register allocation

With architectural parameters provided by
XR² architecture description language

Future Work

- We will

Investigate many kinds of architectures.

Experiences from
- custom-fit heuristics affected by target specific info.
- target parameters from architectural parameters

à solution of code generation for embedded processors