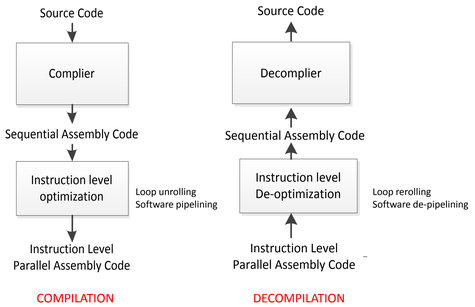
INSTRUCTION LEVEL LOOP DE-OPTIMIZATION
LOOP REROLLING AND SOFTWARE DE-PIPELINING

Erh-Wen Hu and Bogong Su, Dept. of Computer Science, William Paterson University Wayne NJ. USA
Jian Wang, Mobile Broadband Software Design, Ericsson, Ottawa, ON, Canada

DECOMPILATION

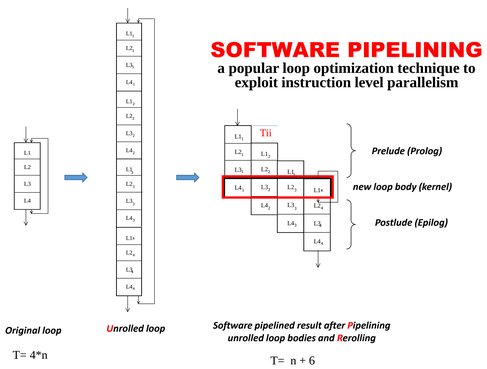
Important issue in software reverse engineering
- Porting legacy software
- Re-optimizing assembly code
- Detecting bugs
- Detecting malware
- Decompilation for modern high performance processors needs instruction level de-optimization

COMPILATION/DECOMPILATION for Instruction Level Parallel Assembly Code



SOFTWARE PIPELINING

a popular loop optimization technique to exploit instruction level parallelism



LOOP UNROLLING

another popular loop optimization technique

- Reduce loop overhead
- Loop unrolling provides more opportunities to increase instruction level parallelism

Original loop	After loop unrolling

int x;
for (x = 0; x < 100; x++)
{
delete(x);
}
	int x;
	for (x = 0; x < 100; x += 5)
	{
	delete(x);
	delete(x + 1);
	delete(x + 2);
	delete(x + 3);
	delete(x + 4);
	}

LOOP UNROLLING + SOFTWARE PIPELINING

- further enhances the performance
- code is very complicated

Assembly code of flt_filterap function from SMV benchmark after unrolling and software pipelining

- Two level nested loop
- Optimization
  - inner unrolling x10
  - outer unrolling x2
  - then software pipelining
- Each row represents one *instruction group* containing several instructions executed in parallel

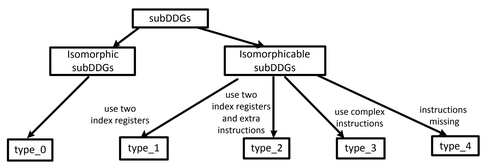
OBSERVATION

We observed eight optimized assembly code with unrolled and/or software pipelined loops generated by TI C6 compiler
- Three from EEMBC Telecommunication benchmarks
- Five from SMV benchmarks

Our observation:
In real DSP code most subDDGs are not isomorphic

- Reasons
  - TI compiler replaces single-word instructions (e.g. LDW) with more efficient complex double-word instructions (LDDW)
  - Peephole optimization removes some instructions in some subDDGs
- Causing difficulties in loop rerolling

CATEGORY OF SUBDDGS

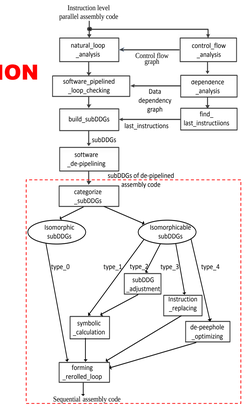


0. subDDGs are all isomorphic: use the same index register and have the same operations on their corresponding nodes.
1. contains some memory fetch instructions using an additional index register for accessing the same array due to limitation of instruction format when unrolling too many times.
2. uses two index registers to access the same array and an additional instruction in some subDDGs to move data across datapath, because memory fetch instruction must use the index register from its own datapath in the TI processor.
3. uses complex instructions to replace some simple instructions.
4. some of its instructions missing in its subDDGs due to peephole optimization.

METHODOLOGIES

1. Perform software de-pipelining first, then perform rerolling if the loop has been software pipelined after unrolling
2. Build data dependency graphs of subDDGs based on the analysis of innermost loops in assembly code. The process begins from the *last_instructions* to reduce the search space
3. Find clusters of potential unrolled copies including all isomorphic subDDGs and isomorphicable subDDGs
4. Convert all isomorphicable subDDGs to isomorphic subDDGs using symbolic calculation, instruction replacing, de-peephole optimization and other techniques
5. Use single loop to represent all isomorphic subDDGs, which is the rerolled loop.

FLOW CHART OF LOOP DE-OPTIMIZATION TECHNIQUE



type_0: all subDDGs are isomorphic

type_1: use different index registers

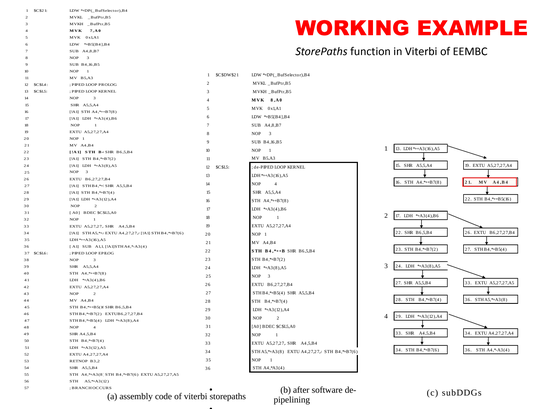
type_2: uses two index registers and additional instruction

type_3: uses complex instructions

type_4: some of its instructions missing due to peephole optimization

WORKING EXAMPLE

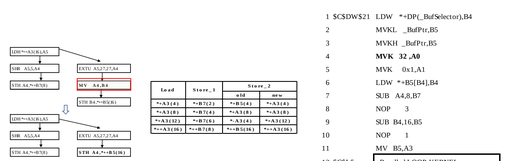
StorePaths function in Viterbi of EEMBC



(a) assembly code of viterbi storepaths

(b) after software de-pipelining

(c) subDDGs



(d) subDDGs adjustment
the mv instruction be removed, now we have 4 type_1 subDDGs

Load	Store_1	Store_2	
		old	new
*+A3(4)	*+B7(2)	*+B5(4)	*+A3(4)
*+A3(8)	*+B7(4)	*+A3(8)	*+A3(8)
*+A3(12)	*+B7(6)	*+A3(4)	*+A3(12)
*++A3(16)	*+B7(8)	*++B5(16)	*++A3(16)

(e) indexes of memory load and store instructions

- Each subDDG has one Load and two Store instructions, the second store instructions of the four unrolled copies use different index registers
- After the execution of *symbolic_calculation* function, all unrolled copies use the same index register for the second store instruction
- Now we have four isomorphic subDDGs (type_0)

```
1  $C$DW$21    LDW   *+DP(_BufSelector),B4
2              MVKL  _BufPtr,B5
3              MVKH  _BufPtr,B5
4              MVK   32 ,A0
5              MVK   0x1,A1
6              LDW   *+B5[B4],B4
7              SUB   A4,8,B7
8              NOP   3
9              SUB   B4,16,B5
10             NOP   1
11             MV    B5,A3
12 $C$L5:      ; Rerolled LOOP KERNEL
13             LDH   *++A3(4),B6
14             [ A0] BDEC $C$L5,A0
15             NOP   3
16             SHR   B6,5,B4       EXTU B6,27,27,B4
17             STH   B4,*+B7(2)    STH B4,*+A3(4)
18 $C$L6:      RETNOP  B3,2
```

(f) rerolled loop of viterbi storepaths

- After the execution of forming_rerolled_loop function, the rerolled loop is semantically equivalent to the original assembly code shown in (a)

Characteristics of Unrolled Loops

No	Function Name	source code: levels of nest	source code: loop count: outer	source code: loop count: mid	source code: loop count: inner	asm code: levels of nested	asm code: loop count: outer	asm code: loop count: mid	asm code: loop count: inner	Loop Optimization Applied	subDDGs: Numbers	subDDGs: Type	Features of Isomorphicable subDDGs	Solution of Loop De-optimization
1	Dot product	1	-	-	100	1	-	-	50	unroll x2 then s/w pipelining	2	0		software de-pipelining
2	Viterbi Decoder	1	-	-	31	0	-	-	0	unroll x31	31	1	two index registers	symbolic calculation
3	Viterbi StorePaths	1	-	-	32	1	-	-	7	unroll x4, then s/w pipelining	4	2	one extra MV instruction and two index registers	1. software de-pipelining 2. subDDG adjustment 3. symbolic calculation
4	SMV LSF_1	1	-	-	7	0	-	-	0	unrolling x7	7	3	use complex intructions	use simple instructions to replace complex instruction
5	SMV LSF_2	3	9	128	10	2	9	128	0	innermost unrolling x10	10	3	use complex intructions	use simple instructions to replace complex instruction
6	SMV LSF_3	3	9	128	7	2	9	128	0	innermost unrolling x7	7	4	one less instruction due to peephole optimization	de-peephole optimization
7	SMV LSF_4	2	7	-	10	1	7	-	0	inner unrolling x10 then outer s/w pipelining	10	2	one extra MV instruction and two index registers	1. software de-pipelining 2. subDDG adjustment 3. symbolic calculation
8	SMV FLT	2	170	-	9	1	85	-	0	1. inner unrolling x10 2. outer unrolling x2 3. s/w pipelining	outer _0	10	3	use complex intructions	1. software de-pipelining 2. use simple instructions to replace complex instruction
											outer _1	10	4	1. use complex instructions 2. two subDDGs have no load instruction due to peephole optimization 3. some subDDGs have extra MV instructions	1. software de-pipelining 2. use simple instructions to replace complex instruction 3. de-peephole optimization 4. subDDG adjustment

EXPERIMENTAL DATA

No.	name	Original: # Inst.	Original: # IG	Original: Clock Cycles	Original: Loop Count	After De-pipelining: # Inst.	After De-pipelining: # IG	After De-pipelining: Clock Cycles	After De-pipelining: Loop Count	After Rerolling: # Inst.	After Rerolling: # IG	After Rerolling: Clock Cycles	After Rerolling: Loop Count
1	Dot product	44	19	113	50	14	12	552	50	11	8	802	100
2	Viterbi FindMetrics	50	38	38	0	/	/	/	/	21	15	204	32
3	Viterbi StorePaths	65	54	208	7	40	35	211	8	18	16	243	32
4	SMV LSF_1	13	13	13	0	/	/	/	/	6	6	37	7
5	SMV LSF_2	86	43	40	0	/	/	/	/	13	11	120	10
6	SMV LSF_3	37	19	19	0	/	/	/	/	21	11	48	6
7	SMV LSF_4	144	35	125	7	71	38	146	7	10	8	630	70
8	SMV FLT	237	92	2997	85	171	60	5100	85	29	38	13260	outer 170, inner 9

- # Inst. denotes the number of instructions representing the code size
- # IG denotes the number of instruction groups
- Clock Cycles represent the execution time of specific code used in the experiment
- Three sections
  - the leftmost one is **original** assembly code
  - the rightmost section is the final result of the semantically equivalent sequential code after loop de-optimization (re-rolling and/or software de-pipelining)
  - The second section lists certain kernels after software de-pipelining
- In the final result, numbers of instructions, code sizes, and instruction groups, are reduced while the number of clock cycles is increased

EXPERIMENT

Select eight loop examples to conduct experiments manually
- Dot product, Viterbi Decoder and Viterbi StorePaths are from Viterbi function of EEMBC Telecommunication benchmark
- other five kernels are from SMV benchmark

Original sets of assembly code are generated by TIC64 compiler
- TIC64 is a digital signal processor with two datapaths, each consists of four function units and one memory port; May issue up to eight instructions including two memory fetches at same time
- All examples have loop unrolling
- Some involve both loop unrolling and software pipelining

Results

Dot product is the simplest example
- all its subDDGs are isomorphic subDDGs using the same index register
- The *categorize_subDDGs* function determines it is type_0
- The *forming_rerolled_loop* function can thus be called immediately

SMV FLT is the most complicated case
- the compiler unrolls the inner loop first, and then unrolls the code of outer loops, and finally software pipelines them
- peephole optimization is used to reduce some instructions, which further complicates the rerolling process

SUMMARY

- Our instruction level loop de-optimization algorithms involve software de-pipelining and loop rerolling
- Instruction level loop de-optimization can be very complicated, particularly when the assembly code after loop unrolling is combined with software pipelining and peephole optimization
- Although different compilers may generate different optimized assembly code, our approach can be a useful technique for the difficult tasks of loop rerolling and software de-pipelining necessary to decompile loops at instruction level
- In this paper, we consider only loop independent dependency and plan to extend it to handle loop carried dependency in the future.