LOOP level parallism and dependence

23-11-2022

- To keep a pipeline full, parallelism amomg instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.
- To avoid a pipeline stall, the execution of dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction.

Loop Level Parallelism and Dependence

Q: What is Loop Level Parallelism?
A: ILP that exists as a result of iterating a loop.

→ Two types of dependencies limit the degree to which Loop Level Parallelism can be exploited.

Two types of dependencies

Loop Carried

A dependence, which only applies if a loop is iterated.

Loop Independent

A dependence within the body of the loop itself (i.e. within one iteration).

An Example of Loop Level Dependences

- Consider the following loop:

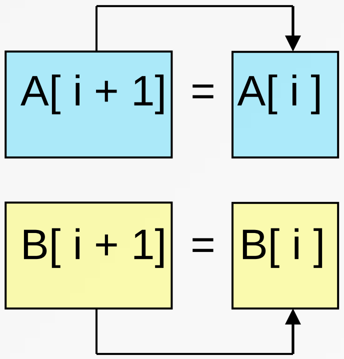
```
for (i  = 0; i <= 100; i++) {

        A[ i + 1]  =  A[ i ]  +  C [ i ] ;                // S1

        B[ i + 1]  =  B[ i ]  +  A [i + 1] ;              // S2

}
```

A Loop Independent Dependence

N.B. how do we know A[i+1] and A[i+1] refer to the same location? In general by performing pointer/index variable analysis from conditions known at compile time.

An Example of Loop Level Dependences

- Consider the following loop:



```
for (i  = 0; i <= 100; i++) {
        A[ i + 1]  =  A[ i ]  +  C [ i ] ;              // S1
        B[ i + 1]  =  B[ i ]  +  A [i + 1] ;            // S2
}
```

Two Loop Carried Dependences

We'll make use of these concepts when we talk about software pipelining and loop unrolling !

We will look at two local optimizations, applicable to loops:

STATIC LOOP UNROLLING

Loop Unrolling replaces the body of a loop with several copies of the loop body, thus exposing more ILP.

KEY IDEA:
Reduce loop control overhead and thus increase performance

SOFTWARE PIPELINING

Reschedule instructions from a sequence of loop iterations to enhance ability to exploit more ILP.

KEY IDEA:
Reduce stalls due to data dependencies.

These two are usually complementary in the sense that scheduling of software pipelined instructions usually applies loop unrolling during some earlier transformation to expose more ILP, exposing more potential candidates “to be moved across different iterations of the loop”.

STATIC LOOP UNROLLING

- OBSERVATION: A high proportion of loop instructions executed are loop management instructions (next example should give a clearer picture) on the induction variable.
- KEY IDEA: Eliminating this overhead could potentially significantly increase the performance of the loop:
- We'll use the following loop as our example:

```
for (i = 999 ; i >= 0 ; i -- )  {
            x[ i ] = x[ i ] + constant;
}
```

STATIC LOOP UNROLLING (continued) – a trivial translation to MIPS

```
for (i = 999 ; i >= 0 ; i -- )  {
        x[ i ] = x[ i ] + constant;
}
```

Our example translates into the MIPS assembly code below (**without any scheduling**).

Note the loop independent dependence in the loop ,i.e. x[i] on x[i]

R1 is initially the address of the element in the array with the highest address

F2 CONTAINS THE SCALAR VALUE

R2 is precomuted. So 8(R2) is the last element to oerate on. Or 1st element in array.

```
Loop :  L.D      F0,0(R1)       ; F0 = array element.
        ADD.D    F4,F0,F2       ; add scalar in F2
        S.D      F4,0(R1)       ; store result
        DADDUI   R1,R1,#-8      ; decrement ptr
        BNE      R1,R2,Loop     ; branch if R1 !=R2
```

STATIC LOOP UNROLLING (continued)

- Let us assume the following latencies for our pipeline:

INSTRUCTION PRODUCING RESULT	INSTRUCTION USING RESULT	LATENCY (in CC)*
FP ALU op	Another FP ALU op	*3*
FP ALU op	Store double	*2*
Load double	FP ALU op	*1*
Load double	Store double	*0*

- Also assume that functional units are fully pipelined or replicated, such that one instruction can issue every clock cycle (assuming it’s not waiting on a result!)
- Assume no structural hazards exist, as a result of the previous assumption

*** - CC == Clock Cycles**

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Let us issue the MIPS sequence of instructions obtained:

			CLOCK CYCLE ISSUED
Loop :	L.D	F0,0(R1)	*1*
		stall	*2*
	ADD.D	F4,F0,F2	*3*
		stall	*4*
		stall	*5*
	S.D	F4,0(R1)	*6*
	DADDUI	R1,R1,#-8	*7*
		stall	*8*
	BNE	R1,R2,Loop	*9*
		stall	*10*

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Let us issue the MIPS sequence of instructions obtained:

			CLOCK CYCLE ISSUED
Loop :	L.D	F0,0(R1)	*1*
		stall	*2*
	ADD.D	F4,F0,F2	*3*
		stall	*4*
		stall	*5*
	S.D	F4,0(R1)	*6*
	DADDUI	R1,R1,#-8	*7*
		stall	*8*
	BNE	R1,R2,Loop	*9*
		stall	*10*

→Each iteration of the loop takes 10 cycles!

→ We can improve performance by rearranging the instructions, in the next slide.

We can push S.D. after BNE, if we alter the offset!

We can push ADDUI between L.D. and ADD.D, since R1 is not used anywhere within the loop body (i.e. it’s the induction variable)

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Here is the rescheduled loop:

			CLOCK CYCLE ISSUED
→Loop :	L.D	F0,0(R1)	*1*
	DADDUI	R1,R1,#-8	*2*
	ADD.D	F4,F0,F2	*3*
	stall		*4*
	BNE	R1,R2,Loop	*5*
	S.D	F4,8(R1)	*6*

Here we've decremented R1 before we've stored F4. Hence need an offset of 8!

→ Each iteration now takes 6 cycles

→ This is the best we can achieve because of the inherent dependencies and pipeline latencies!

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Here is the rescheduled loop:

			CLOCK CYCLE ISSUED
Loop :	L.D	F0,0(R1)	*1*
	DADDUI	R1,R1,#-8	*2*
	ADD.D	F4,F0,F2	*3*
	stall		*4*
	BNE	R1,R2,Loop	*5*
	S.D	F4,8(R1)	*6*

Observe that 3 out of the 6 cycles per loop iteration are due to loop overhead !

STATIC LOOP UNROLLING (continued)

- Hence, if we could decrease the loop management overhead, we could increase the performance.

- **SOLUTION : Static Loop Unrolling**

  → Make n copies of the loop body, adjusting the loop terminating conditions and perhaps renaming registers (we'll very soon see why!),

  → This results in less loop management overhead, since we effectively merge n iterations into one !

  → This exposes more ILP, since it allows instructions from different iterations to be scheduled together!

STATIC LOOP UNROLLING (continued) – issuing our instructions

- The unrolled loop from the running example with an unroll factor of n = 4 would then be:

```
Loop :   L.D       F0,0(R1)
         ADD.D     F4,F0,F2
         S.D       F4,0(R1)
         ------------------------
         L.D       F6,-8(R1)
         ADD.D     F8,F6,F2
         S.D       F8,-8(R1)
         ------------------------
         L.D       F10,-16(R1)
         ADD.D     F12,F10,F2
         S.D       F12,-16(R1)
         ------------------------
         L.D       F14,-24(R1)
         ADD.D     F16,F14,F2
         S.D       F16,-24(R1)
         ------------------------
         DADDUI    R1,R1,#-32
         BNE       R1,R2,Loop
```

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ The unrolled loop from the running example with an unroll factor of n = 4 would then be:

```
Loop :  L.D      F0,0(R1)
        ADD.D    F4,F0,F2
        S.D      F4,0(R1)
        ...................
        L.D      F6,-8(R1)
        ADD.D    F8,F6,F2
        S.D      F8,-8(R1)
        ...................
        L.D      F10,-16(R1)
        ADD.D    F12,F10,F2
        S.D      F12,-16(R1)
        ...................
        L.D      F14,-24(R1)
        ADD.D    F16,F14,F2
        S.D      F16,-24(R1)
        ...................
        DADDUI   R1,R1,#-32
        BNE      R1,R2,Loop
```

n loop Bodies for n = 4

Adjusted loop overhead instructions

Note the renamed registers. This eliminates dependencies between each of n loop bodies of different iterations.

Note the adjustments for store and load offsets (only store highlighted red)!

STATIC LOOP UNROLLING (continued) – issuing our instructions

- Let's schedule the unrolled loop on our pipeline:

			CLOCK CYCLE ISSUED
Loop :	L.D	F0,0(R1)	*1*
	L.D	F6,-8(R1)	*2*
	L.D	F10,-16(R1)	*3*
	L.D	F14,-24(R1)	*4*
	ADD.D	F4,F0,F2	*5*
	ADD.D	F8,F6,F2	*6*
	ADD.D	F12,F10,F2	*7*
	ADD.D	F16,F14,F2	*8*
	S.D	F4,0(R1)	*9*
	S.D	F8,-8(R1)	*10*
	DADDUI	R1,R1,#-32	*11*
	S.D	F12,16(R1)	*12*
	BNE	R1,R2,Loop	*13*
	S.D	F16,8(R1);	*14*

STATIC LOOP UNROLLING (continued) – issuing our instructions

→ Let's schedule the unrolled loop on our pipeline:

This takes 14 cycles for 1 iteration of the unrolled loop.

Therefore w.r.t. original loop we now have 14/4 = 3.5 cycles per iteration.

Previously 6 was the best we could do!

→ We gain an increase in performance, at the expense of extra code and higher register usage/pressure

→ The performance gain on superscalar architectures would be even higher!

			CLOCK CYCLE ISSUED
Loop :	L.D	F0,0(R1)	*1*
	L.D	F6,-8(R1)	*2*
	L.D	F10,-16(R1)	*3*
	L.D	F14,-24(R1)	*4*
	ADD.D	F4,F0,F2	*5*
	ADD.D	F8,F6,F2	*6*
	ADD.D	F12,F10,F2	*7*
	ADD.D	F16,F14,F2	*8*
	S.D	F4,0(R1)	*9*
	S.D	F8,-8(R1)	*10*
	DADDUI	R1,R1,#-32	*11*
	S.D	F12,16(R1)	*12*
	BNE	R1,R2,Loop	*13*
	S.D	F16,8(R1);	*14*

STATIC LOOP UNROLLING (continued)

However loop unrolling has some significant complications and disadvantages:

- Unrolling with an unroll factor of n, increases the code size by (approximately) n. This might present a problem,

- Imagine unrolling a loop with a factor n= 4, that is executed a number of times that is not a multiple of four:

  → one would need to provide a copy of the original loop and the unrolled loop,

  → this would increase code size and management overhead significantly,

  → this is a problem, since we usually don’t know the upper bound (UB) on the induction variable (which we took for granted in our example),

  → more formally, the original copy should be included if (UB mod n != 0), i.e. number of iterations is not a multiple of the unroll factor

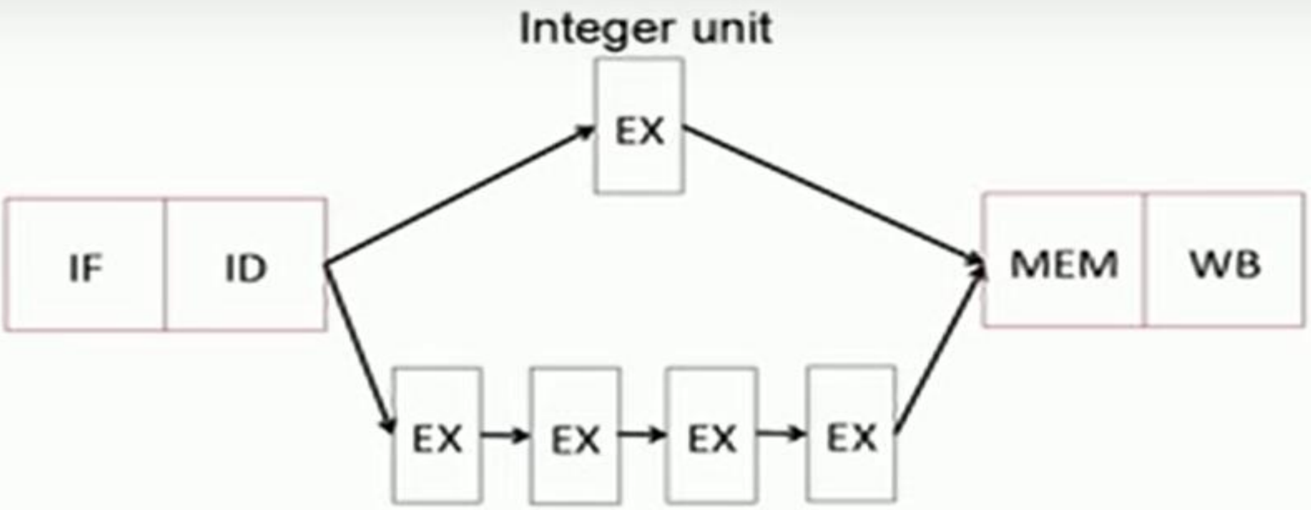
STATIC LOOP UNROLLING (continued)

However loop unrolling has some significant complications and disadvantages:

- We usually *ALSO* need to perform register renaming to reduce dependencies within the unrolled loop. This increases the register pressure!

- The criteria for performing loop unrolling are therefore usually very restrictive!

multi-cycle functional Unit of a RISC processor(MIPS)



Instruction pipeline