Software Pipelining

Ruud van der Pas

SiliconGraphics

European High Performance Computing Team

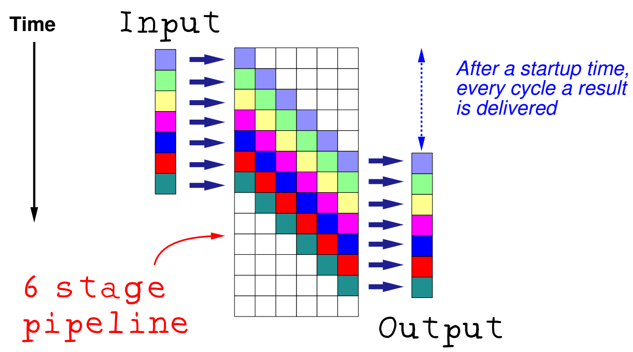
Software Pipelining

- Software pipelining (SWP for short), is the most powerful compiler technique

  To Obtain Maximum Performance

- The principles are based upon hardware pipelining
- It can be seen as a *generalisation* of loop unrolling
- Loop iterations are executed in an interleaved way
- SWP is used to:
  - *Hide instruction latencies*
  - *Exploit the superscalar nature of the microprocessor*

Pipelining – Assembly Line Idea



Software Pipelining – Without

Example: madd instruction (a=a+b*c)

latency is 4 cycles, repeat rate 1 cycle

Cycle	FP Instructions	Int instructions
1	madd a1,b1,c1	---
2	---	---
3	---	---
4	---	---
5	---	store a1

This scheme produces 1 result/4 cycles

Software Pipelining – With

Example: madd instruction (a=a+b*c)

latency is 4 cycles, repeat rate 1 cycle

Cycle	FP Instructions	Int instructions
1	madd a5,b5,c5	store a1
2	madd a6,b6,c6	store a2
3	madd a7,b7,c7	store a3
4	madd a8,b8,c8	store a4
5	madd a9,b9,c9	store a5

?

Where do these numbers come from?

This scheme produces 1 result/cycle

So, we get a 4x speed-up!

Software Pipelining – With

Cycle	FP Instructions		Int instructions	
1	madd	a1,b1,c1	---	Prologue
2	madd	a2,b2,c2	---	
3	madd	a3,b3,c3	---	
4	madd	a4,b4,c4	---	
i	madd	a(i) ,b(i) , c(i)	store a(i–4)	Steady State
i+1	madd	a(i+1),b(i+1), c(i+1)	store a(i–3)	
i+2	madd	a(i+2),b(i+2), c(i+2)	store a(i–2)	
i+3	madd	a(i+3),b(i+3), c(i+3)	store a(i–1)	
i+4	madd	a(i+4),b(i+4), c(i+4)	store a(i)	
N–3	---		store a(N–3)	Epilogue
N–2	---		store a(N–2)	
N–1	---		store a(N–1)	
N	---		store a(N)	

Software Pipelining (SWP)

As we have seen, with SWP we distinguish 3 phases:

prologue phase	**prepare the "pipeline"**
steady state phase	***execute the "pipeline"***
epilogue phase	**shutdown of the "pipeline"**

Consequence:

A certain *minimum length* is needed to compensate for the cost of executing the prologue/epilogue phase

This lengths depends on the actual (loop) characteristics

SWP – Compiler Options

Compile with:

–O3	**Lower optimisation levels don't invoke SWP by default !**
–SWP:=on	**Activate SWP at a lower optimisation level**
–S	**Generates an assembly file with SWP messages Search for:**

- ***swps* (=steady state phase)**
- ***swpf* (=SWP failed)**

Software Pipelining – Example #1

```
RESIDUAL = 0.0
DO J = START_Y, END_Y, 4
   RESIDUAL = RESIDUAL + RESIDUALS(J)
END DO
```

Compiled with -O3

```
#<swps>
#<swps> Pipelined loop line 72 steady state
#<swps>
#<swps>    12 estimated iterations before pipelining
#<swps>     8 unrollings before pipelining
#<swps>     9 cycles per 8 iterations
#<swps>     8 flops      ( 44% of peak) (madds count as 2)
#<swps>     8 flops      ( 44% of peak) (madds count as 1)
#<swps>     0 madds      (  0% of peak)
#<swps>     9 mem refs   (100% of peak)
#<swps>     3 integer ops  ( 16% of peak)
#<swps>    20 instructions ( 55% of peak)
#<swps>     2 short trip threshold
#<swps>     5 integer registers used.
#<swps>    24 float registers used.
#<swps>
```


Software Pipelining – Assembly

```
lwc1    $f22,92($4)                      # [0]
add.s   $f21,$f20,$f21                   # [1]
lwc1    $f19,76($4)                      # [1]
add.s   $f18,$f17,$f18                   # [2]
lwc1    $f16,60($4)                      # [2]
add.s   $f15,$f14,$f15                   # [3]
lwc1    $f13,44($4)                      # [3]
add.s   $f12,$f11,$f12                   # [4]
lwc1    $f10,28($4)                      # [4]
add.s   $f9,$f8,$f9                      # [5]
daddiu  $3,$4,128                        # [5]
lwc1    $f7,12($4)                       # [5]
add.s   $f6,$f5,$f6                      # [6]
slt     $2,$1,$3                         # [6]
lwc1    $f4,-4($4)                       # [6]
add.s   $f3,$f2,$f3                      # [7]
pref    0,252($3)                        # [7]
add.s   $f1,$f0,$f1                      # [8]
bne     $5,$0,.BB45.update_blocks_       # [8]
lwc1    $f20,108($3)                     # [8]
```

Note that several registers are used to accumulate the results

Software Pipelining – Messages

Compiled with -O2

50	estimated iterations before pipelining	
2	unrollings before pipelining	
4	*cycles per 2 iterations*	
2	flops	(25% of peak) (madds count as 2)
2	flops	(25% of peak) (madds count as 1)
0	madds	(0% of peak)
2	mem refs	(50% of peak)
4	integer ops	(50% of peak)
8	instructions	(50% of peak)
1	short trip threshold	
4	integer registers used.	
4	float registers used.	
2	min cycles required for resources	
4	cycles required for recurrence(s)	
2	operations in largest recurrence	

Compiled with -O3

12	estimated iterations before pipelining	
8	unrollings before pipelining	
9	*cycles per 8 iterations*	
8	flops	(44% of peak) (madds count as 2)
8	flops	(44% of peak) (madds count as 1)
0	madds	(0% of peak)
9	mem refs	(100% of peak)
3	integer ops	(16% of peak)
20	instructions	(55% of peak)
2	short trip threshold	
5	integer registers used.	
24	float registers used.	

```
RESIDUAL = 0.0
DO J = START_Y, END_Y, 4
   RESIDUAL = RESIDUAL + RESIDUALS(J)
END DO
```


Software Pipelining

Software pipelining observations:

- *Needs a certain level of independence*
- *This technique is by now well understood*
- *It is always good to check the SWP messages*
- *SWP cycle estimates assume an ideal world*
- *Typical assumption is that all data is in the L1 cache*

Software Pipelining – Independence

```
void update(int n, float *a, float *b, int *idx, float s)
{
  int i;
  for (i=0; i<n; i++)
     a[idx[i]] += s*b[i];
}
```

Comments:

- *The compiler (or more accurately: the software pipeliner), will have to assume that for a certain combination of (i1,i2):*

  idx[i1] = idx[i2]

- *This means that there is a dependency for i=i1 and i=i2*
- *As a result, the compiler will generate a correct, but non–optimal, instruction scheduling scheme:*

Software Pipelining – Independence

Compiled with -O3

```
#<swps>    25 estimated iterations before pipelining
#<swps>     4 unrollings before pipelining
#<swps>    72 cycles per 4 iterations
#<swps>     8 flops       (  5% of peak) (madds count as 2)
#<swps>     4 flops       (  2% of peak) (madds count as 1)
#<swps>     4 madds         (  5% of peak)
#<swps>    17 mem refs      ( 23% of peak)
#<swps>    11 integer ops   (  7% of peak)
#<swps>    32 instructions  ( 11% of peak)
#<swps>     1 short trip threshold
#<swps>     7 integer registers used.
#<swps>     3 float registers used.
#<swps>
#<swps>    18 min cycles required for resources
#<swps>    72 cycles required for recurrence(s)
#<swps>    28 operations in largest recurrence
#<swps>
```


Software Pipelining – Independence

```
void update(int n, float *a, float *b, int *idx, float s)
{
  int i;
#pragma ivdep

  for (i=0; i<n; i++)
     a[idx[i]] += s*b[i];
}
```

Comments:

- *The "#pragma ivdep" informs the compiler that there is no dependency*
- *As a result, it will now generate an optimal, instruction scheduling scheme:*

Notes:

IVDEP = Ignore Vector DEPendency

Option -OPT:alias=restrict can also be used for this

Software Pipelining – Independence

Compiled with -O3

```
#<swps>    25 estimated iterations before pipelining
#<swps>     4 unrollings before pipelining
#<swps>    18 cycles per 4 iterations
#<swps>     8 flops       ( 22% of peak) (madds count as 2)
#<swps>     4 flops       ( 11% of peak) (madds count as 1)
#<swps>     4 madds         ( 22% of peak)
#<swps>    16 mem refs      ( 88% of peak)
#<swps>    11 integer ops   ( 30% of peak)
#<swps>    31 instructions  ( 43% of peak)
#<swps>     1 short trip threshold
#<swps>    14 integer registers used.
#<swps>    10 float registers used.
```

Software Pipelining – Example #2

```
for (idx=0;idx<list[ptr]-1; idx++){
    delta[src[idx]] += cou[idx] * delta[ptr];
    wed[idx] = neu[src[idx]] * c1 + c2 * wed[idx];
    cou[idx] += wed[idx];
}
```

Observations:

- *In total, 8 memory references (ld+st) are required per iteration*
- *We perform 6 floating point operations (2 madds) per iteration*
- *Therefore, we can (theoretically) overlap all floating point with the memory operations => absolute minimum cycle count for this loop is 8*
- *In 8 cycles, we can perform a maximum of 16 flops, but our operation only does 6 flops => 38% of peak is best case*

Software Pipelining – Assembly

Compiled with -O3:

```
#<loop> Not unrolled: analysis found no benefit (disable
        w/-CG:unroll_analysis=no)
#<sched>
#<sched> Loop schedule length:33 cycles (ignoring nested
         loops)
#<sched>
#<sched>     6 flops      (  9% of peak) (madds count as 2)
#<sched>     4 flops      (  6% of peak) (madds count as 1)
#<sched>     2 madds       (  6% of peak)
#<sched>    12 mem refs    ( 36% of peak)
#<sched>    11 integer ops ( 16% of peak)
#<sched>    27 instructions ( 20% of peak)
```

- *This loop is not software pipelined because the loop end value `list[ptr]-1` may change while the loop is executed*
- *It needs to be "a constant variable"*

Software Pipelining – Modified #1

```
int vlen = list[ptr]-1;
for (idx=0;idx<vlen; idx++){
    delta[src[idx]] += cou[idx] * delta[ptr];
    wed[idx] = neu[src[idx]] * c1 + c2 * wed[idx];
    cou[idx] += wed[idx];
}
```

```
#<swps>     51 cycles per iteration
#<swps>      6 flops     (  5% of peak) (madds count as 2)
#<swps>      4 flops     (  3% of peak) (madds count as 1)
#<swps>      2 madds          (  3% of peak)
#<swps>     11 mem refs       ( 21% of peak)
#<swps>      8 integer ops    (  7% of peak)
#<swps>     23 instructions ( 11% of peak)
#<swps>      1 short trip threshold
#<swps>     10 integer registers used.
#<swps>      5 float registers used.
#<swps>     14 min cycles required for resources
#<swps>     50 cycles required for recurrence(s)
#<swps>     19 operations in largest recurrence
#<swps>     51 cycles in minimum schedule found
```


Software Pipelining – Modified #2

```
int vlen = list[ptr]-1;
#pragma ivdep
for (idx=0;idx<vlen; idx++){
    delta[src[idx]] += cou[idx] * delta[ptr];
    wed[idx] = neu[src[idx]] * c1 + c2 * wed[idx];
    cou[idx] += wed[idx];
}
```

```
#<swps>     Not unrolled before pipelining
#<swps>     12 cycles per iteration
#<swps>      6 flops        ( 25% of peak) (madds count as 2)
#<swps>      4 flops        ( 16% of peak) (madds count as 1)
#<swps>      2 madds        ( 16% of peak)
#<swps>      9 mem refs     ( 75% of peak)
#<swps>      8 integer ops  ( 33% of peak)
#<swps>     21 instructions ( 43% of peak)
#<swps>      5 short trip threshold
#<swps>     23 integer registers used.
#<swps>      8 float registers used.
```

Software Pipelining – Assembly –O3

Addressing causes the extra cycles:

```
madd.d $f0,$f0,$f2,$f4        # [0]   multiply and add
sdc1 $f1,0($7)                # [0]   store double word
ldc1 $f4,0($5)                # [1]   load double word
dsll $1,$4,3                  # [2]   left shift by 3
lw $4,0($8)                   # [2]   load word
sdc1 $f0,0($3)                # [3]   store double word
daddu $1,$1,$9                # [3]   double word add unsigned
ldc1 $f1,0($1)                # [4]   load double word
mul.d $f0,$f3,$f5             # [6]   multiply
madd.d $f0,$f0,$f1,$f6        # [7]   multiply and add
ldc1 $f3,0($24)               # [7]   load double word
ldc1 $f2,0($10)               # [8]   load double word
dsll $1,$2,3                  # [9]   left shift by 3
daddiu $31,$24,8              # [9]   double word add immediate unsigned
daddu $3,$1,$11               # [10]  double word add unsigned
daddiu $7,$6,8                # [10]  double word add immediate unsigned
lw $2,0($19)                  # [10]  load word
add.d $f1,$f0,$f0             # [11]  add
daddiu $17,$19,4              # [11]  double word add immediate unsigned
beq $18,$12,.BB23.swp3_4      # [11]  branch on equal
ldc1 $f0,0($3)                # [11]  load double word
```


Software Pipelining – Modified #3

```
int vlen = list[ptr]-1;double rvdp=delta[ptr];
#pragma ivdep
for (idx=0;idx<vlen; idx++){
    delta[src[idx]] += cou[idx] * rvdp;
    wed[idx] = neu[src[idx]] * c1 + c2 * wed[idx];
    cou[idx] += wed[idx];
}
```

```
#<swps>    50 estimated iterations before pipelining
#<swps>     2 unrollings before pipelining
#<swps>    19 cycles per 2 iterations
#<swps>    12 flops        ( 31% of peak) (madds count as 2)
#<swps>     8 flops        ( 21% of peak) (madds count as 1)
#<swps>     4 madds        ( 21% of peak)
#<swps>    16 mem refs     ( 84% of peak)
#<swps>    12 integer ops  ( 31% of peak)
#<swps>    36 instructions ( 47% of peak)
#<swps>     3 short trip threshold
#<swps>    20 integer registers used.
#<swps>    11 float registers used.
```


Software Pipelining – Assembly –O3

19/2=9.5 cycles/iteration is more or less optimal !

```
sdc1 $f0,0($7)          # [0]
sdc1 $f1,8($7)          # [1]
dsll $1,$6,3            # [2]
ldc1 $f3,0($16)         # [2]
daddu $2,$1,$8          # [3]
dsll $1,$3,3            # [4]
ldc1 $f0,0($2)          # [4]
madd.d $f1,$f0,$f5,$f6  # [5]
daddu $3,$1,$8          # [5]
ldc1 $f0,0($3)          # [6]
madd.d $f0,$f0,$f4,$f6  # [7]
dsll $1,$5,3            # [7]
lw $6,4($13)            # [7]
sdc1 $f1,0($2)          # [8]
daddu $2,$1,$9          # [8]
dsll $1,$4,3            # [9]
ldc1 $f1,0($2)          # [9]
```

```
sdc1 $f0,0($3)              # [10]
daddu $1,$1,$9              # [10]
mul.d $f0,$f2,$f7           # [11]
ldc1 $f2,0($1)              # [11]
madd.d $f1,$f0,$f1,$f8      # [12]
ldc1 $f5,8($15)             # [12]
mul.d $f0,$f3,$f7           # [13]
lw $3,0($13)                # [13]
madd.d $f0,$f0,$f2,$f8      # [14]
lw $5,4($14)                # [14]
ldc1 $f4,0($15)             # [15]
daddiu $24,$31,16           # [16]
add.d $f1,$f1,$f1           # [17]
daddiu $7,$15,16            # [17]
ldc1 $f2,8($31)             # [17]
add.d $f0,$f0,$f0           # [18]
lw $4,0($14)                # [18]
beq $16,$10,.BB25.swp3_5    # [18]
daddiu $14,$13,8            # [18]
```


Software Pipelining – Example #2

To summarize:

Version	*Memory refs./iteration*	*Cycles/iteration*
Original	*12*	*33.0*
+ Fixed loop length	*11*	*51.0*
+ pragma ivdep	*9*	*12.0*
+ temp scalar	*8*	*9.5*

Software Pipelining – Summary

- *Software pipelining is crucial (R8K) and important (R10K) for good performance*
- *SWP requires a certain independence between iterations*
- *Compile with optimisation level –O3*
- *Lower optimisation level, try –SWP:=on*
- *You may have to help the compiler (especially with C programs) in case of possible dependencies*
- *It is recommended to check the software pipelining messages for the relevant loops*