

Introduction to Software Pipelining in the IA-64 Architecture

intel. *IA-64 Software Programs*

This presentation is an introduction to software pipelining in the IA-64 architecture.

It is recommended that the reader have some exposure to the IA-64 architecture prior to viewing this presentation.

Agenda

- Objectives
- What is software pipelining?
- IA-64 architectural support
- Assembly code example
- Compiler support
- Summary

Objectives

- Introduce the concept of software pipelining (SWP)
- Understand the IA-64 architectural features that support SWP
- See an assembly code example
- Know how to enable SWP in compiler

What is Software Pipelining?

intel. *IA-64 Software Programs*

Software Pipelining

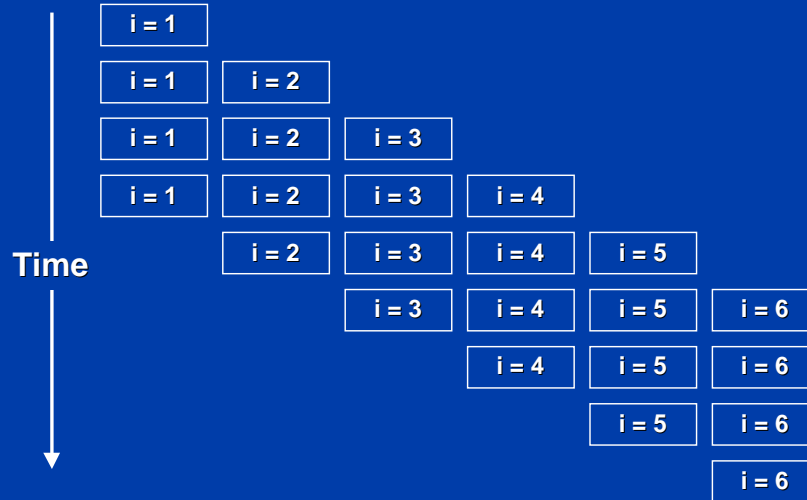
- Software performance technique that overlaps the execution of consecutive loop iterations
- Exploits instruction level parallelism across iterations

intel. IA-64 Software Programs

Software Pipelining (SWP) is the term for overlapping the execution of consecutive loop iterations. SWP is a performance technique that can be done in just about every computer architecture.

SWP is closely related to loop unrolling.

Software Pipelining



intel. IA-64 Software Programs

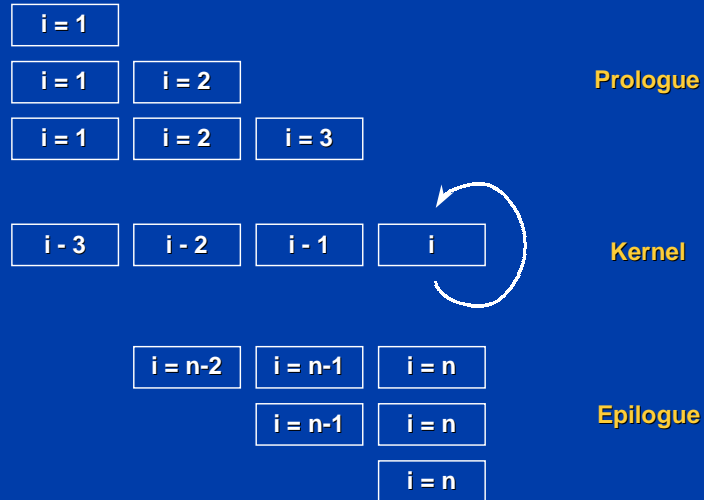
Here is a conceptual block diagram of a software pipeline. The loop code is separated into four pipeline stages.

Six iterations of the loop are shown ($i = 1$ to 6).

Notice how the pipeline stages overlap. The second iteration ($i=2$) can begin while the first iteration ($i=1$) begins the second stage. This overlap of iterations increases the amount of parallel operations that can be executed in the processor. More parallel operations helps increase performance.

Many, but not all, loops may be software pipelined.

Software Pipelining



intel. IA-64 Software Programs

Here is another conceptual block diagram of a software pipeline. This is the same four stage pipeline.

Here we see the loop prolog (the start of the pipeline), the kernel (all pipeline stages active), and the epilog (the completion of the pipeline).

Modulo Loop Scheduling

- One of many software pipelining techniques
- Direct support in IA-64 architecture
- High performance code with minimal code space

intel. IA-64 Software Programs

Modulo Loop Scheduling is just one software pipelining technique.

There is direct support in the IA-64 architecture for modulo loop scheduling: rotating registers, predicates, special branch instructions, and loop count registers.

The architectural support makes it easy to create SWP loops. The resulting code has high performance and is compact.

Modulo Loop Scheduling

- **Basic technique**
 - Create a schedule for one loop iteration that can be repeated at regular intervals
- **Initiation interval (ii)**
 - **Constant interval between iterations**
 - Number of clock cycles between starting iteration i and starting iteration $i+1$
 - **Goal is to minimize ii to increase throughput**

intel. IA-64 Software Programs

Modulo Loop Scheduling involves developing a schedule for one loop iteration such that when the schedule is repeated at regular intervals, no intra- or inter-iteration dependency is violated, and no resource usage conflict arises.

The Initiation Interval (ii) is essentially the length of one pipeline stage - the number of clock cycles it takes to execute one pass through the loop instructions or the number of clock cycles between starting iteration i and starting iteration $i+1$.

When comparing SWP loops, compare the ii and the number of stages. For high loop counts, ii is more important than the number of stages. With high loop counts, more time is spent in the loop's kernel rather than the prolog and epilog. The kernel time is determined by the number of iterations and the ii.

IA-64 Architectural Support

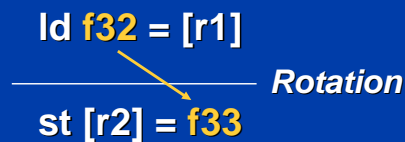
intel. *IA-64 Software Programs*

The IA-64 architecture has special support for software pipelining.

Rotating Registers

- Registers rotate for each loop iteration
- Rotation provides a “new register” for each loop iteration
- General, predicate, FP registers rotate

`ld f32 = [r1]`
----- *Rotation*
`st [r2] = f33`



intel. IA-64 Software Programs

The General registers, Floating Point (FP) registers, and the Predicate registers have the ability to rotate. Conceptually, the data rotates or moves from a register to the adjacent register. For example, if data is written to F32 on one iteration, the data will be in F33 on the next iteration. This allows the instruction to reference a new “register” on each iteration. F32 can be written again, because the value has moved to F33.

General registers R32 and above can rotate, as can Floating Point registers F32-F127 and Predicate registers P16-P63. General registers must be enabled to rotate with the “alloc” instruction (not discussed here). The rotation for General registers is by blocks of 8 registers.

The register rotation is circular - F127 rotates to F32 and P63 rotates to P16. For the general registers, R40 rotates to R32 if one block of 8 registers is enabled to rotate.

Predication

- Predicate registers control the pipeline stages
 - One predicate assigned to each stage
 - Allows the instructions to execute at the correct time
- Rotating predicates advance the pipeline

```
(p16) ld f32 = [r1] // Stage 1  
(p17) st [r2] = f33 // Stage 2
```

intel. IA-64 Software Programs

One predicate register is assigned to each stage in the pipeline. All instructions for that stage will share the same predicate. This allows the instructions for the stage to execute at the correct time (i.e. only when that stage is ready to execute).

The rotating predicate registers act a shift register that starts up the software pipeline (prolog), keeps it moving (kernel), and then shuts down the pipeline at the end of the loop (epilog).

P16 is the first rotating predicate register and it is typically used to control the first stage in the pipeline.

Branch Instructions

- Counted loop branches
 - br.ctop, br.cexit
- While loop branches
 - br.wtop, br.wexit
- Controls the rotation and loop count

p16 = cmp.eq r0, r0

Loop:

(p16) ld f32 = [r1]

(p17) st [r2] = f33

br.ctop Loop

intel. IA-64 Software Programs

“top” branches are used at the bottom of the loop:

if loop is to continue then branch to the top of the loop,
else, fall through to end the loop.

“exit” branches are used if the loop exit is not at the loop bottom:

if loop is to continue, then fall-through,
else, branch out of the loop to end the loop.

Refer to the IA-64 Application Developer’s Architecture Guide to understand the difference between the different branch instruction types.

In the code snippet above, notice that P16 is set before entry into the software pipelined loop.

Count Registers

- **Loop Count (LC) Register**
 - Number of loop iterations
 - For counted loops only
- **Epilog Count (EC) Register**
 - Contains the number of pipeline stages
 - Used to count during the epilogue

intel. IA-64 Software Programs

LC is for Counted loops only, not While loops. It is loaded with the loop count minus one.

EC is used to count the stages during the loop's epilog. EC is used for both Counted loops and While loops.

br.ctop Instruction

- If $LC > 0$
 - Decrement LC, Set P63 = 1
 - Rotate registers
- When $LC == 0$
 - Decrement EC, Set P63 = 0
 - Rotate registers
- Loop exits when EC Register == 1

intel. IA-64 Software Programs

br.ctop is used in counted loops. It is used in the assembly code example that follows.

Here is how the br.ctop instruction executes:

If LC is greater than 0, decrement LC, set P63 to 1, and rotate the registers. Setting P63 to 1 will rotate which sets P16, which continues the pipeline.

If LC equals zero, decrement the EC, set P63 to 0, rotate the registers. Setting P63 to 0 will rotate which clears P16, which starts to empty the pipeline.

Exit the loop when EC register equals 1.

Assembly Code Example

intel. *IA-64 Software Programs*

This is an assembly code example to illustrate how software pipelining works.

DAXPY Example Code

```
double x[N], y[N], da;  
for ( i=0; i<N; i++ ) {  
    y[i] = da*x[i] + y[i];  
}
```

intel. IA-64 Software Programs

DAXPY is the inner loop in many equation solvers. In this example, we will use a counted loop.

Step 1: Dependency Graph

- **Select instructions**
 - ld, fma, st, br.ctop
- **Create dependency graph**
 - No inter-iteration dependencies here



Dependency Graph

intel. IA-64 Software Programs

ld = load (actually ldf is used)

fma = floating point multiply-and-add instruction

st = store (actually stf is used)

br.ctop = branch

There are no inter-iteration dependencies here. Each iteration is can be executed independently. This simplifies the remaining steps. The intra-iteration dependency graph is shown.

Step 2: Map to Resources

- Map instructions to available execution units
 - Disregard instruction latencies and dependencies at this point
 - Determines the initiation interval (ii = 2 cycles)



intel. IA-64 Software Programs

Mapping the instructions to the available processor execution units creates the Modulo Reservation Table (MRT). This table determines the loop's ii.

In this case, we assume a hypothetical machine that has two memory execution units. With two memory units, only two load or store operations can be issued per clock cycle. Therefore, we need a minimum of two clock cycles to issue the 3 memory instructions (2 loads and 1 store).

The FMA instruction could be executed in cycle 1, if needed.

The branch instruction can be executed in the second cycle, with the fma and st instructions. It is not shown in the diagram above.

Why does $ii = 2$ cycles?

- Assume the processor has two memory execution units
- Have three memory instructions to issue: ld, ld, st
- Therefore, need two cycles to issue the instructions
 - fma instruction could be issued in first or second cycle

intel. IA-64 Software Programs

The branch instruction executes in cycle 2.

Code (so far)

Loop:

```
ld      y[i]
ld      x[i] ;;
fma     y[i] ← da * x[i] + y[i]
st      y[i]
br.ctop Loop ;;
```

intel. IA-64 Software Programs

Here is pseudo code for the loop so far. It's not quite a SWP loop yet and it would not execute correctly! The next steps will take care of that.

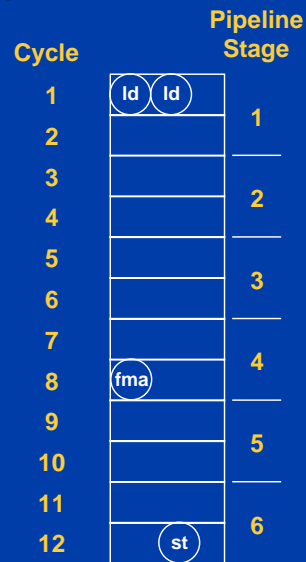
Step 3: Schedule

- Use instruction latencies to schedule the instructions

– Instruction latencies:
ld = 7, fma = 4

(For this example only!)

- 12 cycles and ii of 2 = 6 pipeline stages



intel. IA-64 Software Programs

With the MRT schedule of two cycles, each pipeline stage is therefore two cycles. Here we use instruction latencies of 7 cycles for the loads and 4 cycles for the fma instruction. **These latencies are for the example only!**

Using the instruction latencies, an optimal schedule can be developed and the instructions are assigned to a particular pipeline stage.

With a ld latency of 7, the fma instruction should execute at cycle 8. With a fma latency of 4 cycles, the st should execute at cycle 12.

The optimal schedule in this case is 12 cycles.

Therefore, we need six pipeline stages to fully hide the latency for the load and the fma instruction. The pipeline will have six pipeline stages.

Step 4: Assign Predicates

- Assign predicate register to control each pipeline stage

p16 → Stage 1 (ld, ld)

p17 → Stage 2

p18 → Stage 3

p19 → Stage 4 (fma)

p20 → Stage 5

p21 → Stage 6 (st)

intel. IA-64 Software Programs

Now we can assign a predicate register to each pipeline stage. We start at P16 because this is where predicate register rotation begins. The `br.ctop` instructions sets or clears P63 before the register rotation. After the rotation, P16 will be set or clear depending on whether or not the pipeline is continuing or in the epilog.

The assignment of the predicate registers to the pipeline stages enforces the intra-iteration dependency of `ld, ld → fma → st`. Note that these instructions are assigned to different stages, which enforces the dependency.

Before entry into the loop, the code will need to set register P16 and clear registers P17-P63. This code is not shown.

Code (so far)

Loop:

```
(p16)  ld      y[i]
(p16)  ld      x[i] ;;
(p19)  fma    y[i] ← da * x[i] + y[i]
(p21)  st      y[i]
      br.ctop Loop ;;
```

intel. IA-64 Software Programs

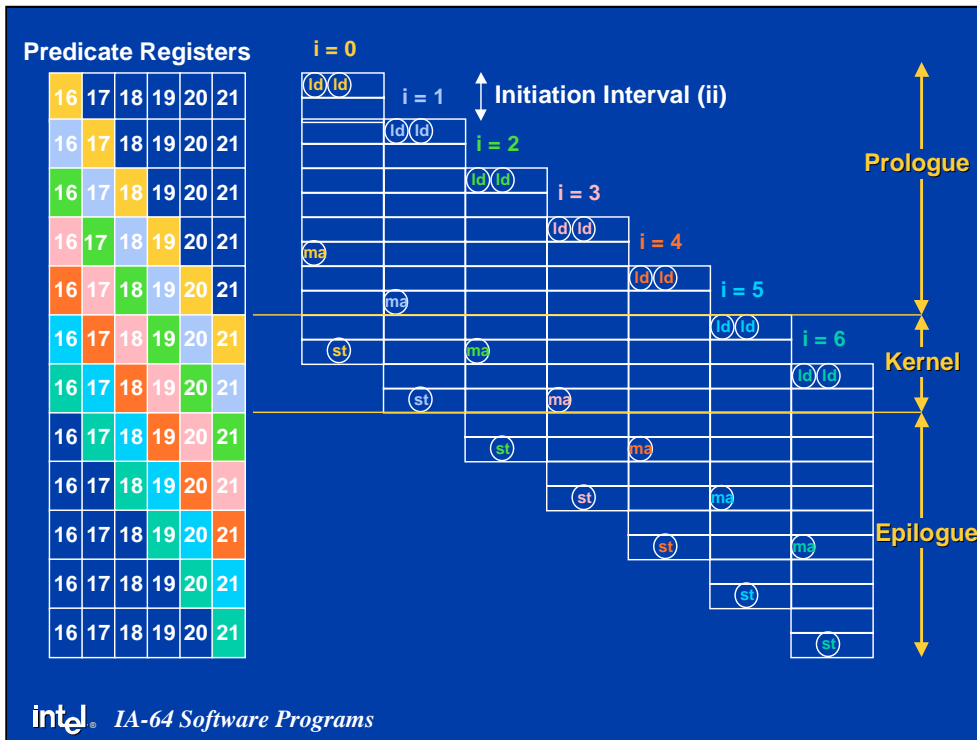
Here is pseudo code for the loop so far. Register assignment follows after the animated slide. The code outside the loop is not shown.

Loop Animation

- 7 loop iterations are shown
 - $i = 0$ to $i = 6$
- Iterations are color-coded

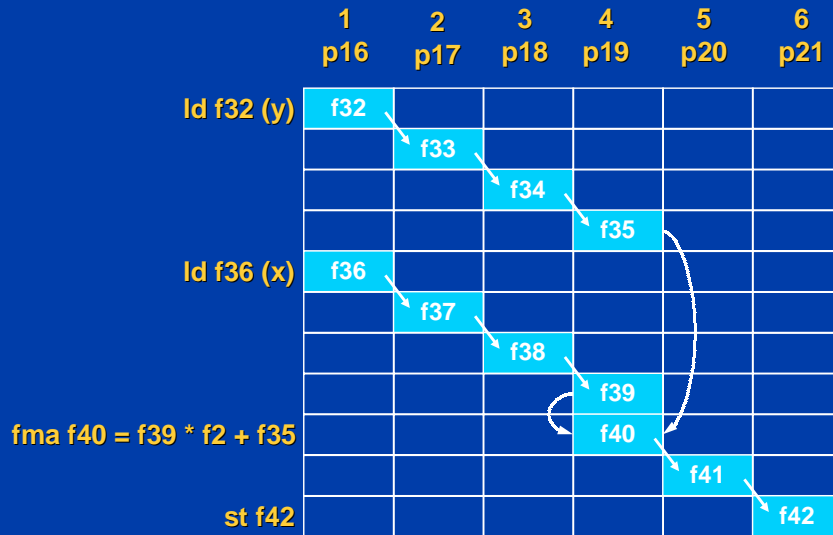
intel. IA-64 Software Programs

The next slide contains an animation of seven loop iterations. The `br.ctop` instruction is not shown in the animation. It executes at the bottom of each iteration (or every two cycles in this case).



This is an animated slide that shows the loop when the loop count is 7. The iterations are color-coded. Predicate register P16 must be set by the code before entry into the loop. The `br.ctop` instruction (not shown in the diagram) keeps the pipeline going and then shuts it down during the epilog.

Step 5: Assign Registers

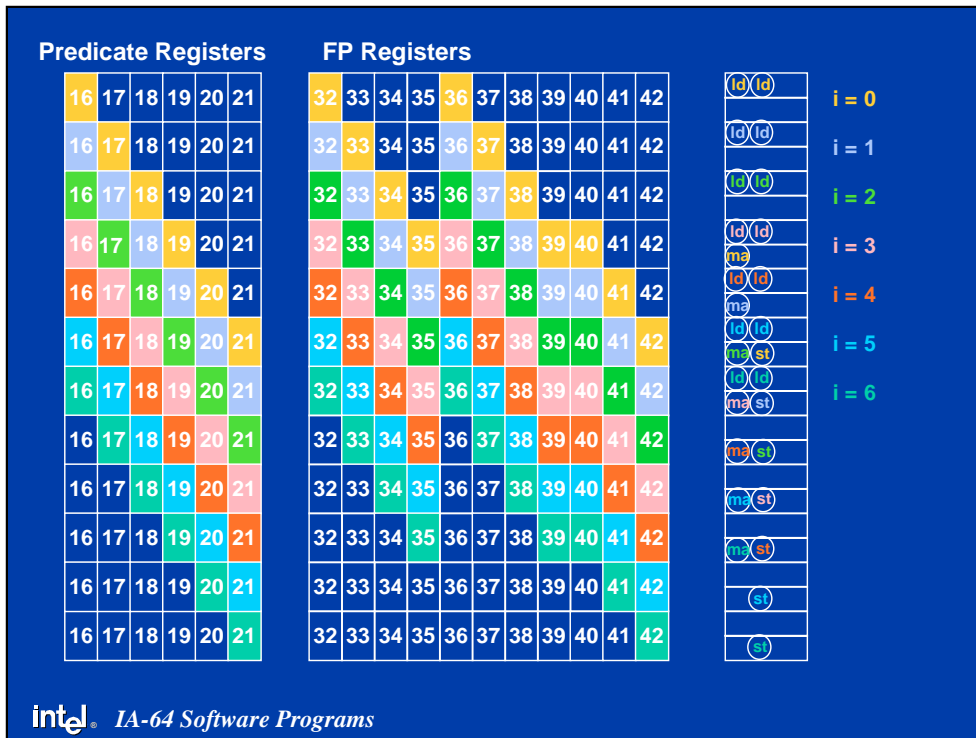


intel. IA-64 Software Programs

Next, we need to assign registers.

The y element needs 4 rotating registers and the x element also needs 4 rotating registers. They need 4 registers because the value is needed for 4 pipeline stages (loaded at stage 1 and used at stage 4). The fma result (the new y) needs 3 rotating registers (from stage 4 to stage 6). The st instruction uses F42.

The loop invariant “da” is loaded into a non-rotating FP register, F2, in this case.



This is an animated slide that shows the same loop when the loop count is 7. The iterations are color-coded. It shows the rotating predicate registers and the rotating floating point registers.

DAXPY Assembly Code

• $y[i] = da * x[i] + y[i];$

```
LOOP: // ii=2, stages=6
      (p16) ldfd      f32=[r34], 8 // stage 1 (Load y)
      (p16) ldfd      f36=[r41], 8 // stage 1 (Load x)
      ;;
      (p19) fma.d     f40=f39,f2,f35 // stage 4 (FMA)
      (p21) stfd      [r39]=f42 // stage 6 (St y)
      br.ctop LOOP
      ;;
```

intel IA-64 Software Programs

Here is the assembly code for the inner loop of the DAXPY example. R34 and R41 contain the addresses of y and x, respectively. The post-increment capability of the load instruction is used to increment the addresses to the next element.

Loop Count

- Example with loop count = 7
 - Set LC = 6
 - Set EC = 6



intel. IA-64 Software Programs

This slide shows the LC and EC registers, again with a loop count of 7. Notice that LC is written with the loop count minus one. EC is set with the number of pipeline stages.

LC counts the prolog and kernel. EC counts the epilog only.

Note that LC could be set to 0, indicating a loop count of one. In this case, the loop never reaches the kernel.

Compiler Support

- **How to enable SWP in Intel® compiler**
 - Enabled with -O2 switch
 - Typical SWP Disqualifiers
 - Function call, complicated If statements, unbalanced If statements, memory disambiguation problems, short loop counts
- **Good candidates for SWP**
 - No first-order recurrences
 - Example: $x[i] = \dots x[i-1] \dots$;
 - Long loop counts

intel. IA-64 Software Programs

Support for software pipelining is included in the Intel® C/C++ compiler.

Summary

- SWP enables compact code
- Removes the need to unroll loops
- Optimized for a particular processor

**IA-64 Modulo Scheduling Provides
the Performance of Loop Unrolling
Without Code Expansion**

References

- “IA-64 Application Developer’s Architecture Guide”, Intel Corp., Doc # 245188
- Rau, B. R. “Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops”, MICRO-27
- Wolfe, Michael “High Performance Compilers for Parallel Computers”, Jan. 1996, Addison Wesley