Software Pipelining of Nested Loops

Kalyan Muthukumar and Gautam Doshi

Intel Corporation
2200 Mission College Blvd., Santa Clara, CA 95052, U.S.A.
{kalyan.muthukumar, gautam.doshi}@intel.com

Abstract. Software pipelining is a technique to improve the performance of a loop by overlapping the execution of several iterations. The execution of a software-pipelined loop goes through three phases: prolog, kernel, and epilog. Software pipelining works best if most of the time is spent in the kernel phase rather than in the prolog or epilog phases. This can happen only if the trip count of a pipelined loop is large enough to amortize the overhead of prolog and epilog phases. When a software-pipelined loop is part of a loop nest, the overhead of filling and draining the pipeline is incurred for every iteration of the outer loop. This paper introduces two novel methods to minimize the overhead of software-pipeline fill/drain in nested loops. In effect, these methods overlap the draining of the software pipeline corresponding to one outer loop iteration with the filling of the software pipeline corresponding to one or more subsequent outer loop iterations. This results in better instruction-level parallelism (ILP) for the loop nest, particularly for loop nests in which the trip counts of inner loops are small. These methods exploit Itanium^{TM} architecture software pipelining features such as predication, register rotation, and explicit epilog stage control, to minimize the code size overhead associated with such a transformation. However, the key idea behind these methods is applicable to other architectures as well. These methods have been prototyped in the Intel optimizing compiler for the Itanium^{TM} processor. Experimental results on SPEC2000 benchmark programs are presented.

1 Introduction

Software pipelining [1,2,4,6,7,10,13,14,15,16] is a well known compilation technique that improves the performance of a loop by overlapping the execution of independent instructions from several iterations. The execution of a software-pipelined loop goes through three phases: **prolog**, when the pipeline is filled - i.e. new iterations are commenced and no iterations are completed, **kernel**, when the pipeline is in steady state - i.e. new iterations are commenced and older iterations are completed, and **epilog**, when the pipeline is drained - i.e. no new iterations are commenced and older iterations are completed. See Fig.1(a).

Since maximum instruction-level parallelism (ILP) is obtained during the kernel phase, software pipelining works best if most of the execution time is spent in kernel phase rather than in prolog or epilog phases. This can happen only if the trip count of the pipelined loop is large enough to amortize the necessary overhead of the prolog and epilog phases. In practice, there are nested loops, in which the outer loop(s) have high trip counts, and the inner loop has a low trip count. In such cases, the inner loop's software pipeline fill/drain overhead is incurred for every outer loop iteration. This overhead is then amortized over only a few inner loop iterations, hence relatively less time is spent in the kernel phase. This results in poor ILP for the loop nest. See Fig.1(b).

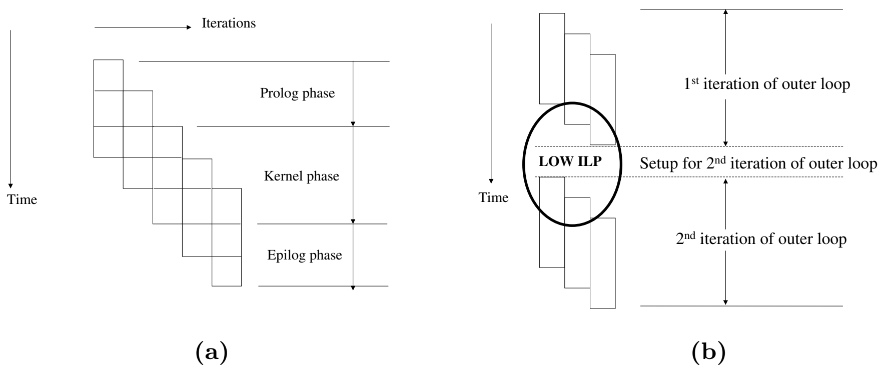

Iterations
Time
Prolog phase
Kernel phase
Epilog phase
(a)
Time
LOW ILP
1st iteration of outer loop
Setup for 2nd iteration of outer loop
2nd iteration of outer loop
(b)

Fig. 1. (a) Phases of a software-pipelined loop, (b) Prolog/Epilog overhead for inner loops with short trip counts.

This paper presents two novel methods to address this problem. These methods perform **Outer Loop Pipelining (OLP)**, by overlapping the epilog of the software pipeline corresponding to one outer loop iteration with the prolog of the software pipelines corresponding to one or more subsequent outer loop iterations. Thus, the software pipeline for the inner loop is filled just once for the loop nest, when the first iterations for all the loops in the loop nest are executed. The software pipeline is also drained just once for the loop nest, when the last iterations for all the loops in the loop nest are executed.

Using Itanium^{TM} architecture [9,8] software pipelining features (such as predication [12], register rotation [5], and epilog stage count register), these methods can be implemented with minimal code changes. The inner loop schedule remains unchanged and only a few additional instructions are added to the outer loop. Hence these methods work well even when the trip counts of inner loops are large. In such cases, the performance improvement due to OLP is not as large as when the inner loops have shorter trip counts. Since there is either a small or a large performance gain, and negligible performance penalty for using this technique, it is especially useful for loop nests whose trip counts are not known at compile-time.

These methods have been prototyped in the Intel optimizing compiler for the Itanium^{TM} processor. Experimental results for kernels derived from workstation applications indicate good speedups for loop nests that have short trip count inner loops. Results on SPECfp2000 and SPECint2000 suites of benchmarks also validate the applicability of this technique for a number of key loops.

The key idea behind these methods can also be applied to architectures that do not have support for rotating registers and other features for software pipelining. In such cases, limited overlap can be achieved between the execution of two successive outer loop iterations, at the expense of some increase in code size.

The rest of this paper is organized as follows. Section 2 describes the Itanium^{TM} architecture features and the software-pipelining schema using these features. Sections 3 presents two OLP methods that use Itanium^{TM} architecture features to achieve pipelining of nested loops. Section 4 discusses how OLP can be applied to traditional architectures as well. Section 5 presents experimental results of these methods on the SPECint2000 and SPECfp2000 suites of benchmarks. Finally, Section 6 provides a summary and directions for future work.

Background and Terminology: We use the term *source loop* to refer to the original source code loop, and the term *kernel loop* to refer to the code that implements the software-pipelined version of the source code. Iterations of the source loop are called *source iterations* and iterations of the kernel loop are called *kernel iterations*. Instructions corresponding to a single source iteration are executed in *stages*. A single source iteration spans multiple kernel iterations. The number of cycles between the start of successive kernel iterations is called the **Initiation Interval (II)**. Figure 1(a) shows the execution of five source iterations of a software-pipelined loop with three pipeline stages.

2 Software Pipelining in the Itanium^{TM} Architecture

The Itanium^{TM} architecture provides many features to aid the compiler in enhancing and exploiting instruction level parallelism (ILP) [9,8]. These include an explicitly parallel (EPIC) instruction set, large register files, register renaming, predication [12], speculation [11], and special support for software pipelining.

The special support for software-pipelined loops includes register rotation, loop branches and loop control registers. Such features were first seen in the Cydrome Cydra-5 [5]. Register rotation provides a renaming mechanism that eliminates the need to unroll loops for the purpose of software renaming of registers. Registers are renamed by adding the register number to the value of a register rename base (RRB) modulo the size of the rotating register file. The RRB is decremented when a software-pipelined loop branch is executed at the end of each kernel iteration. Decrementing the RRB makes the value in register X, during one kernel iteration, appear to move to register X+1, in the next kernel iteration. If X is the highest numbered rotating register, its value wraps to the lowest numbered rotating register. General registers r32-r127, floating-point registers f32-f127, and predicate registers p16-p63 can rotate. Registers r0-r31, f0-r31 and p0-p15 do not rotate and are referred to as *static* registers.

Below is an example of register rotation.

```
L1:     ld4     r32 = [r4],4     // post increment r4 by 4
        add     r34 = r34,r9
        st4     [r5] = r35,4     // post increment r5 by 4
        swp_branch L1 ;;         // software pipeline branch
```

Each stage of the software pipeline is one cycle long (II = 1), a load latency of 2 cycles and an add latency of 1 cycle is assumed. The value that the load writes to r32 is read by the add, two kernel iterations (and hence two rotations) later, as r34. In the meantime, two more instances of the load are executed. However, because of register rotation, those instances write to different registers and do not destroy the value needed by the add.

Predication refers to the conditional execution of an instruction based on a boolean source operand called the qualifying predicate. If the qualifying predicate is True (one), the instruction is executed. If the qualifying predicate is False (zero), the instruction generally behaves like a no-op. Predicates are assigned values by compare, test-bit, or software-pipelined loop branch instructions. Compare instructions generally write two complementary destination predicate registers based on the boolean evaluation of the compare condition.

The rotation of predicate registers serves two purposes. The first, similar to the rotating general and floating-point registers, is to avoid overwriting a predicate value that is still needed. The second purpose is to control the filling and draining of the software pipeline. To do the latter, a predicate is assigned to each stage of the software pipeline to control the execution of the instructions in that stage. This predicate is called a stage predicate. For counted loops, p16 is architecturally defined to be the stage predicate for the first stage, p17 is defined to be the stage predicate for the second stage, etc. A register rotation takes place at the end of each stage (when the swp_branch is executed in the kernel loop). When p16 is set to 1, it enables the first stage for a given source iteration. This value of p16 is rotated to p17 when the swp_branch is executed, to enable the second stage for the same source iteration. Each 1 written into p16, sequentially enables all the stages for a given source iteration. This behavior is used to enable (propagate 1s) or disable (propagate 0s) the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases.

Itanium^{TM} architecture provides special software-pipeline loop branches for counted (br.ctop, br.cexit) and while (br.wtop, br.wexit) loops and software-pipeline loop control registers that maintain the loop count (LC) and epilog count (EC). During the prolog and kernel phases, a decision to continue kernel loop execution means that a new source iteration is started. For example, for a counted loop, LC (which is > 0) is decremented to update the count of remaining source iterations. EC is not modified. P63 is set to one. Registers are rotated (so now p16 is set to 1) and the branch (ctop) is taken so as to continue the kernel loop execution.

Once LC reaches zero, all required source iterations have been started and the epilog phase is entered. During this phase, a decision to continue kernel loop execution means that the software pipeline has not yet been fully drained. P63 is now set to zero because there are no more new source iterations to start and the instructions that correspond to non-existent source iterations must be disabled. EC is decremented to update the count of the remaining stages for the last source iteration. Registers are rotated and the branch is executed so as to continue the kernel loop execution. When EC reaches one, the pipeline has been fully drained, and the branch is executed so as to exit the kernel loop execution.

A pipelined version of the example counted loop, using Itanium^{TM} architecture software pipelining features, is shown below assuming an II of 1 cycle and a loop count of 2 source iterations:

```
           mov     pr.rot = 0 ;;     // Clear rotating preds (16-63)
           mov     LC = 1            // LC = loop count - 1
           mov     EC = 4            // EC = loop stage count
           cmp.eq p16,p0 = r0,r0 ;; // Set p16 = 1
L1:  (p16) ld4     r32 = [r4],4      // Stage1:
     (p18) add     r34 = r34,r9      // Stage3:
     (p19) st4     [r5] = r35,4      // Stage4:
           br.ctop L1 ;;
```

Thus the various Itanium^{TM} architectural features of register rotation, predication and software-pipelined loop branches and registers, enable extremely compact and efficient software-pipelined loop sequences.

3 Software Pipelining of Nested Loops

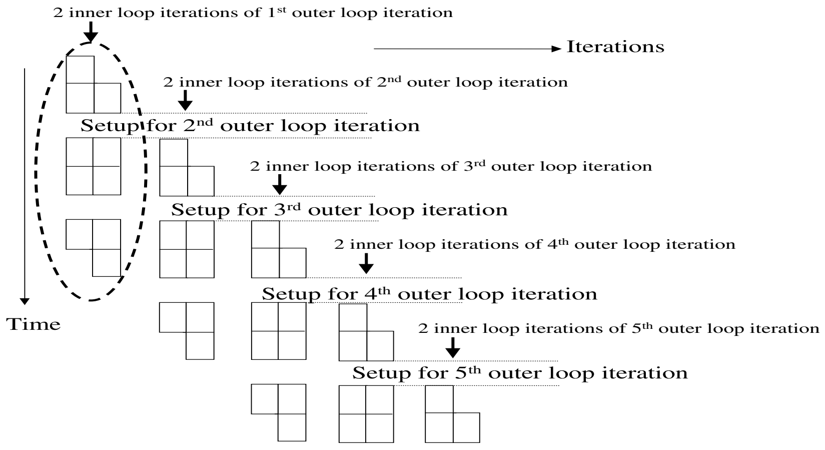
This section presents two new methods for outer loop pipelining (OLP), when the innermost loop is software pipelined. OLP is achieved by overlapping the epilog phase of the inner loop pipeline corresponding to one outer loop iteration, with the prolog (and possibly epilog) phases of the inner loop pipelines corresponding to subsequent outer loop iterations. In doing so, the cost associated with filling and draining the inner loop pipeline is incurred only once during the execution of the entire loop nest rather than during every outer loop iteration. As a result, the performance of the loop nest is improved, especially for inner loops with short trip counts.

Consider a loop nest with an inner loop that is software-pipelined with 5 stages and has a trip count of 2. Figure 2 illustrates how these methods overlap the inner loop computations across multiple outer loop iterations.

During OLP, the inner loop's software pipeline is not drained after all the inner loop source iterations for a given outer loop iteration have been started. Rather, the pipeline is frozen, and set-up for the next outer loop iteration is done. The draining continues during the prolog phase of the next outer loop iteration. Eventually, when all the loops in the enclosing loop nest are in their last iterations, the inner loop software pipeline is drained. Note that computations can be overlapped across more than two outer loop iterations. This occurs when the inner loop pipeline never reaches kernel phase. When inner loop pipeline stages are more than twice the number of inner loop iterations, overlap is achieved across three outer loop iterations. Figure 2 shows inner loop computations overlapped across three outer loop iterations.

Two key features of the Itanium^{TM} architecture enable efficient OLP. They are: (1) rotating registers, and (2) explicit epilog stage control. Rotating registers enable holding the intermediate results of the inner loop computations corresponding to one outer loop iteration, and at the same time, start the inner loop computations for another outer loop iteration. Epilog stage control is achieved via the EC register. Normally EC would have been initialized such that the inner loop is drained completely for each outer loop iteration. In these OLP methods, EC is set to 1 at the start of an inner loop for all outer loop iterations, except for the very last iterations of the outer loops, in which case it is set appropriately so that the inner loop pipeline is completely drained.

The two methods differ in their schemas as to how this is accomplished. The first method does not make copies of the kernel code for the inner loop, but the second method does. The first method works only for counted outer loops, while the second method is more general and does not impose this restriction. Except for the copy of the kernel code in the case of the second method, both methods add very few (static and dynamic) instructions to perform OLP. Since these methods overlap computations across outer loop iterations, they must preserve the register and memory dependences of the original loop nest. This is explained later in this section.



2 inner loop iterations of 1st outer loop iteration
Iterations
2 inner loop iterations of 2nd outer loop iteration
Setup for 2nd outer loop iteration
2 inner loop iterations of 3rd outer loop iteration
Setup for 3rd outer loop iteration
2 inner loop iterations of 4th outer loop iteration
Setup for 4th outer loop iteration
2 inner loop iterations of 5th outer loop iteration
Setup for 5th outer loop iteration
Time

Fig. 2. Overlapping of inner loop pipelines across outer loop iterations.

These methods can be generally applied to perfect or imperfect loop nests of arbitrary depth. For illustration, a running example of a two level perfect loop nest with a counted outer loop is used.

Consider the following loop nest (an excerpt from a critical loop in an important proprietary workstation application), which has a high trip count for the outer loop but a low trip count for the inner loop:

```
REAL*4  A(10,100), B(10,100), C(10,100)
DO  J = 1, 100
  DO  I  = 1, 3
    A(I,J) = B(I,J) / C(I,J)
  ENDDO
ENDDO
```

Suppose that the inner loop is pipelined in 12 stages and has an II of 5 cycles[1]. Furthermore, suppose that it takes 5 cycles to setup the pipeline (i.e. reset array addresses, reset rotating predicate registers, reset EC/LC, etc). Then, this loop requires roughly:

[1] Divides in Itanium^{TM} architecture are implemented as a sequence of instructions [9].

```
Cycles for prolog and kernel stages of inner loop = 100*( 3*5)
Cycles for epilog stages of inner loop            = 100*(11*5)
Cycles to reset for the next outer loop iteration = 100*(   5)
-------------------------------------------------------------
TOTAL CYCLES                                      = 7500 cycles
```

The overhead for draining the epilog stages is very high (5500 cycles i.e. 73% of the total cycles). Traditional techniques of loop collapsing, loop interchange and loop unrolling can be used to address this problem but they each have their own costs. Loop collapsing adds predicated address computations that could affect the inner loop scheduled II. Loop interchange could adversely affect (as it does in this example) the memory access pattern. Loop unrolling increases code size and cannot be easily applied if the inner loop trip count is not a compile-time constant.

Without OLP, the generated code is as follows:

```
           mov     r14 = 99         // Outer loop count - 1
           mov     r2 = 1           // Outer loop index
Outer_loop:
           mov     pr.rot = 0       // P16-P63=0
           mov     EC = 12          // EC = Stage count
           mov     LC = 2 ;;        // LC = Trip count - 1
           cmp.eq  p16,p0 = r0,r0   // P16 = 1
           .....
Inner_loop:
           [inner loop code]
           br.ctop Inner_loop       // Inner loop branch
           .....
           cmp.le  p7,p6 = r2,r14   // Test for outer loop count
           add     r2 = r2,1        // Increment outer loop index
     (p7)  br.cond Outer_loop       // Outer loop branch
```

Fig. 3. Non OLP code for the Running Example.

The computations in the inner loop (loads of B and C, the FP divide sequence, and the store of A) have been omitted to simplify the example. Since the inner loop is a counted loop, it uses the br.ctop instruction. The LC register is initialized to 2 for every iteration of the inner loop, since the inner loop has a trip count of 3 (for a trip count of N, LC is initialized to (N-1)). The EC register is initialized to 12, the stage count of the pipeline.

3.1 Method 1 for Pipelining of Nested Loops

In this method, EC is initialized (to 1) so that the inner loop pipeline is not drained. A test for the final iteration is inserted in the outer loop to conditionally set EC to completely drain the pipeline. The predicate registers (that control the staging of the pipeline) are preserved across outer loop iterations by not clearing them at the start of each inner loop pipeline. The resultant code is as follows:

```
          mov     r14 = 99
          mov     r2 = 1
          mov     pr.rot = 0 ;;        // (1)
          mov     EC = 1               // (2)
Outer_loop:
          mov     LC = 2
          cmp.eq  p16,p0 = r0,r0
          .....
Inner_loop:
          [inner loop code]
          br.ctop Inner_loop
          .....
          cmp.eq  p8,p9 = r2,r14 ;;     // (3)
     (p9) mov     EC = 1                // (4)
     (p8) mov     EC = 12               // (5)
          cmp.le  p7,p0 = r2,r14
          add     r2 = r2,1
     (p7) br.cond Outer_loop
```

Fig. 4. Code After Pipelining of Nested Loops

Instructions to clear the rotating predicates (1) and to set EC (2), have been moved to the preheader of the loop nest. Also, EC has been initialized to 1 instead of 12 to prevent the draining of the inner loop software pipeline. Since the clearing of the rotating predicates is now done outside the loop nest, the rotating predicates p17-p63 retain the values between outer loop iterations. This enables the freezing and restarting of the inner loop pipeline.

Instruction (3) checks for the start of the last iteration of the outer loop. If this is the case, then EC is set to 12 by instruction (5). Otherwise, instruction (4) resets EC to 1. Thus, the inner loop pipeline is drained only during the last iteration of the outer loop.

Thus this method requires the addition of just three new instructions to the outer loop:

- Instruction (3) to check for the start of the last iteration of outer loop,
- Instruction (5) to set EC to stage count, for the last iteration of outer loop,
- Instruction (4) to reset EC to 1, for all other iterations of outer loop.

Assume that the addition of these instructions increases the time required to set-up for the next outer loop iteration from 5 to 6 cycles. However, the pipeline is drained only once, so the time required to execute the loop nest is:

```
Cycles for prolog and kernel stages of inner loop = 100*( 3*5)
Cycles for epilog stages of inner loop            =     (11*5)
Cycles to reset for the next outer loop iteration = 100*(   6)
---------------------------------------------------------------
TOTAL CYCLES                                      = 2155 cycles
```

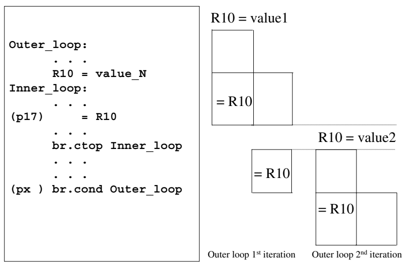
Thus, this method leads to a significant performance improvement (71%) over that of the original code sequence, with hardly any increase in the static or dynamic code size.

Conditions required for this method: OLP essentially involves hoisting inner-loop code associated with later outer loop iterations, so as to overlap them with the current outer loop iteration. As with all code motion, this hoisting must honor the register and memory data dependences of the original code sequence. This section details the conditions that must be satisfied for this method to work correctly. The following are the key ideas behind these conditions, which ensure the correctness of the OLP transformation:

- We should be able to predict the last iteration(s) of the outer loop(s) and ensure that the pipeline of the inner loop is finally drained.
- Register values that are in flight across inner loop kernel iterations, must not be clobbered due to OLP.
- Live-out register values must be computed correctly, even though the pipeline has not been completely drained.
- All memory-dependences across outer-loop iterations must be obeyed.

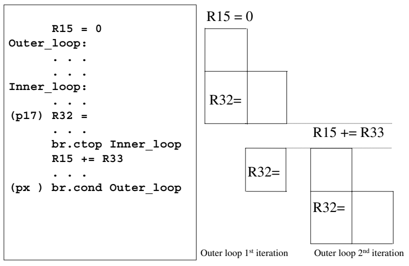
Here are the conditions that ensure the correctness of OLP for a loop nest:
1. **Counted Outer Loops**: All the outer loops must be counted loops. This is needed to set EC to drain the software pipeline only when all the outer loops are in their last iterations.
2. **Single Outer Loop Exit**: Each loop in the loop nest should have only one exit. If this is not satisfied, an early exit in one of the loops would transfer control and skip the instructions in the post-exit of the inner loop that set EC to drain the pipeline of the inner loop.
3. **Live-in Values**: If a value is live-in to the inner loop, it should either be in a rotating register or used only in the first stage of the software pipeline. This condition ensures that register and memory anti-dependences across outer loop iterations in the original loop nest are honored.

Figure 5(a) shows a scenario in which R10 is loop-invariant in the inner loop and is used in the second stage, but changes its value in the immediate outer loop. The inner loop has 2 iterations. After OLP, the second iteration of the inner loop still expects to see “value1” in R10. However, this is clobbered with “value2” when the second iteration of the outer loop is started.



```
Outer_loop:
      . . .
      R10 = value_N
Inner_loop:
      . . .
(p17)     = R10
      . . .
      br.ctop Inner_loop
      . . .
      . . .
(px ) br.cond Outer_loop
```

(a)



```
      R15 = 0
Outer_loop:
      . . .
      . . .
Inner_loop:
      . . .
(p17) R32 =
      . . .
      br.ctop Inner_loop
      R15 += R33
      . . .
(px ) br.cond Outer_loop
```

(b)

Fig. 5. (a) Register Anti-dependences for live-in values (b) Register dependences for live-out values

Note that this problem will not arise if either (a) the values that are live-in are used in the first stage of the pipeline or (b) the live-in values are assigned to rotating registers. In such cases, the live-in values are not clobbered by their subsequent redefinitions in the outer loops.

4. **Live-out Values**: If a value is live-out of the inner loop, and is used within the loop nest in a subsequent outer iteration, it should be defined in the first stage of the software pipeline. This condition ensures that register and memory dependences that exist from the inner loop to an outer loop computation are honored. Consider the loop nest in Fig. 5(b), in which the inner loop has 2 stages and a trip count of 2.

The value defined in R32 in the second iteration of the inner loop is live-out of the inner loop. It gets rotated into R33 and is used to update R15. However, if OLP is done, the second stage of the second iteration of the inner loop does not get executed until after the next iteration of the outer loop is started. The result is that the value that is stored in R15 at the start of the second iteration of the outer loop is the value of R32 that is defined in the first iteration of the inner loop. This would obviously be incorrect.

5. **Rotating Register Values**: Code in the outer loop(s) should not clobber the values held in rotating registers used in the kernel of the inner loop. The register allocator ensures that the rotating registers are not used for outer loop live ranges.

6. **Loop-carried Memory Dependence**: If there is a loop-carried memory dependence for inner loop computations carried by the outer loop, the Point of First Reference (P_{r1}) of a memory location should precede its Point of Second Reference (P_{r2}) in the execution time-line of these instructions. The memory dependence that exists for the pair (P_{r1}, P_{r2}) can be a *flow*, *anti* or *output* dependence. Depending on the number of inner loop iterations that elapse between P_{r1} and P_{r2}, and the stages in which P_{r1} and P_{r2} occur, it may or may not be legal to perform OLP.

Consider the following:

```
DO J = 1, 100
  DO I = 1, 3
           = A(I, J-1)
    A(I, J) =
  ENDDO
ENDDO
```

For this program, there is a memory flow dependence carried by the outer loop for the location A(I, J). P_{r1} is the definition of the value in A(I,J) and P_{r2} is the subsequent use of that value (referenced as A(I,J-1)). P_{r1} and P_{r2} are defined in terms of number of stages of the software pipeline of the inner loop. In general, let P_{r1} and P_{r2} occur in stages D and U, respectively. If M inner loop iterations elapse between P_{r1} and P_{r2}, then :

$P_{r1} = D$, $P_{r2} = U + M$ and OLP is legal iff $P_{r1} < P_{r2}$ (i.e. iff $D < U + M$).

For the above loop nest, M = 3 since three inner loop iterations separate P_{r1} and P_{r2}. If P_{r1} occurs in the 4th stage (D = 4) and P_{r2} occurs in the 2nd stage ($U = 2$), then $D < U + M$, so OLP can be done (Fig. 6(a)). However, if $D = 5$ and $U = 1$, then $D > U + M$, and OLP cannot be done (Fig. 6(b)).

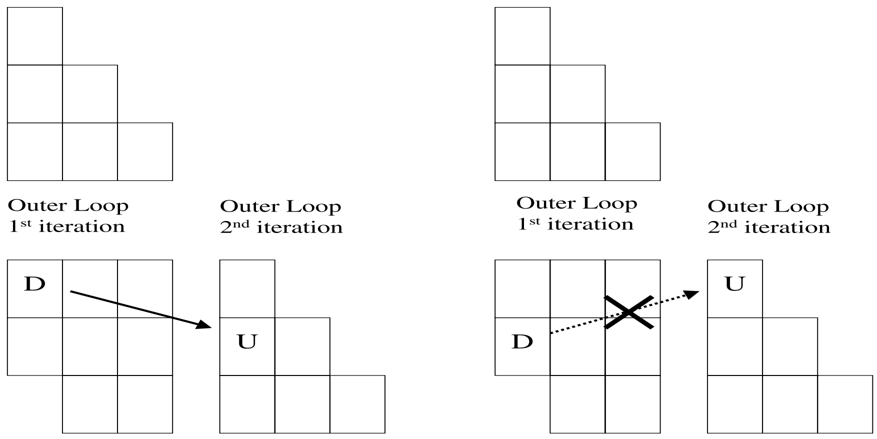


Fig. 6. (a) OLP can be done (b) OLP cannot be done

The value of M is calculated using (a) the number of outer loop iterations that separate P_{r1} from P_{r2}, which is obtained from data dependence analysis [3,17], and (b) the number of inner loop iterations that are executed per outer loop iteration. If the number of inner loop iterations is a compile-time constant, then the legality check for loop-carried memory dependence can be performed at compile-time. Otherwise, based on a run-time check of the number of inner loop iterations, control can either go to a loop nest that has OLP or to another loop nest in which only the inner loop is pipelined.

The conditions described above for the live-in and live-out values and the loop-carried memory dependence in the inner loop are applicable only if we do not want any draining of the software pipeline before starting the next outer loop iterations. These conditions can be relaxed if partial draining is allowed to happen at the end of execution of every iteration of the immediate outer loop. This can be formulated as follows:

- Let $S_{live-in}$ be the maximum stage in which a live-in register value that is in a static register is used in the pipelined inner loop.
- Let $S_{live-out}$ be the maximum stage in which a live-out value that is used in the loop nest, is defined in the pipelined inner loop.
- Let $S_{loop-carried-mem-dep}$ denote the maximum of $(P_{r1} - P_{r2})$ over all memory references that have loop-carried dependence across the outer loop(s). Note that the maximum value of $S_{loop-carried-mem-dep}$ is the number of epilog stages of the pipelined inner loop.

So, the value that EC should be set to before starting the next outer loop iteration is: $ES = Max(1, S_{live-in}, S_{live-out}, S_{loop-carried-mem-dep} + 2)$. If ES is the same as the number of stages required to completely drain the pipeline of the inner loop, then we do not perform OLP, since there is no performance gain from doing it.

Algorithm for Pipelining of Nested Loops: This method can be integrated with modulo scheduling [14] and rotating register allocation in the software pipeliner. The algorithm for this method consists of two steps. The first step, **IsLoopNestEligibleForOLP**, checks to see if the pipelined inner loop and the loop nest satisfy the conditions listed in the previous section. The second step, **PerformOLPForLoopNest**, performs OLP by suitably adding and moving the instructions so that the pipeline of the inner loop is not completely drained. These two functions are invoked by the main function **OLPForLoopNest**.

Algorithm 1 Algorithm for Method 1

```
Bool IsLoopNestEligibleForOLP (Loop_nest, unsigned int *pES)
{
  if (any outer loop is not a counted loop) return False;
  if (any outer loop has more than one exit) return False;
  Compute $S_{live-in}$, $S_{live-out}$, and $S_{loop-carried-mem-dep}$;
  *pES = Max (1, $S_{live-in}$, $S_{live-out}$, $S_{loop-carried-mem-dep} + 2$);
  if (*pES == number of pipeline stages) return False;
  else return True;
}
Void PerformOLPForLoopNest (Loop_nest, unsigned int ES)
{
  Move init of pr.rot from preheader of inner loop to preheader of loop nest;
  Delete the initialization of EC in the preheader of the inner loop;
  Initialize EC = ES in the preheader of the loop nest;
  Add a compare instruction in the post-exit of inner loop to set $p_{Last}$;
  $p_{Last}$ = (last iteration(s) of outer loop(s)) ? 1 : 0;
  Let $p_{NotLast}$ = the predicate register that is complementary to $p_{Last}$;
  Add an instruction "($p_{NotLast}$) EC = ES" to post-exit of inner loop;
  Add an instruction "($p_{Last}$) EC = (Epilog Count to completely drain inner
                          loop)" to post-exit of inner loop;
}
Void OLPForLoopNest (Loop_nest)
{
  unsigned int ES;
  if (IsLoopNestEligibleForOLP (Loop_nest, &ES))
    PerformOLPForLoopNest (Loop_nest, ES);
}
```

3.2 Method 2 for Pipelining of Nested Loops

This method is similar in principle to method 1. However, there are three key differences: (a) it does not require that the outer loops in the loop nest be counted loops, (b) it allows outer loops to have multiple exits, and (c) it makes a copy (or copies if there are multiple exits for any outer loop) of the pipelined inner loop and inserts it after the code for the loop nest. Other conditions that were required for method 1 apply to this method as well. This method is conceptually simpler, less restrictive, and allows more loop nests to be pipelined. However, it comes with the cost of expanded code size and the attendant problems of possibly increased I-cache misses, page faults, instruction TLB misses, etc. This method transforms the code for the running example as follows:

```
       mov     pr.rot = 0 ;;   // (1)
Outer_loop:
       mov     EC = 1          // EC = 1 (no drain)
       mov     LC = 2          // LC = Trip count - 1
       cmp.eq  p16 = r0,r0     // P16 = 1
       ...
Inner_loop:
       [inner loop code]
       br.ctop Inner_loop      // Inner loop branch
       ...
 (p7)  br.cond Outer_loop      // Outer loop branch
       mov     EC = 11         // (2)
Inner_loop_copy:
       [inner loop code]
       br.ctop Inner_loop_copy
       <MOV instructions for values that are live-out of Inner_loop_copy>
```

This method consists of the following steps:

- The pr.rot instruction that is in the preheader of the inner loop in the non-OLP code is moved out of the loop nest.
- EC is now initialized to 1 instead of 12 in the preheader of the inner loop. Immediately following the loop nest, EC is set to 11 (number of epilog stages).
- A copy of the pipelined inner loop is placed following this instruction. This serves to drain the inner loop completely.
- Following this copy of the inner loop, MOV instructions are added for those rotating register values that are live out of the inner loop. These values are not used inside the loop nest, but outside.

4 Pipelining of Nested Loops for Other Architectures

The key idea behind these methods can be applied for nested loops in other architectures as well. However, the speedup that can be achieved in such architectures is potentially smaller than in architectures that have rotating registers and predication. Without these features, this technique can be implemented as follows: The inner loop is peeled such that the code for the prolog phase of the pipelined inner loop is peeled out of the inner loop and placed in the preheader of the loop nest. The code for the epilog phase of the pipelined inner loop is peeled as well and placed after the code for the inner loop. This is intertwined with a copy of the prolog phase of the inner loop for the next outer loop iteration.

This method does achieve overlap between the epilog phase of the pipelined inner loop for one iteration of the outer loop with the prolog phase of the pipelined inner loop for the next iteration of the outer loop. However, it may be

Algorithm 2 Algorithm for Method 2

```
Bool IsLoopNestEligibleForOLP (Loop_nest, unsigned int *pES)
{
  Compute $S_{live-in}$, $S_{live-out}$, and $S_{loop-carried-mem-dep}$;
  *pES = Max (1, $S_{live-in}$, $S_{live-out}$, $S_{loop-carried-mem-dep} + 2$);
  if (*pES == number of pipeline stages) return False;
  else return True;
}
Void PerformOLPForLoopNest (Loop_nest, unsigned int ES)
{
  Move the init of pr.rot in preheader of inner loop to outside the loop nest;
  Replace the ‘mov EC’ instruction in the preheader of the inner
            loop with a MOV instruction that initializes EC to ES;
  At each exit of the loop nest, append an instruction “EC = (Stage count - ES)”
            and a copy of the pipelined inner loop. ;
  Following this, for all rotating register values that are live-out of the inner
            loop to outside the loop nest, add appropriate MOV instructions;
}
```

very difficult to overlap inner loop computations across multiple outer loop iterations. Also, if the inner loop trip count is fewer than the number of prolog stages, exits out of the prolog phase will further add to the code size and complexity. It leads to increased code size that is caused by copies of prolog and epilog phases of the inner loop. Also, modulo scheduling for traditional architectures requires kernel unrolling and/or MOV instructions that also lead to increased code size. However, it may be still profitable to do OLP using this technique for loop nests that have small trip counts for inner loops.

5 Experimental Results

We have prototyped both the OLP methods in the Intel optimizing compiler for the Itanium^{TM} processor. The performance of the resulting OLP code was measured on the Itanium^{TM} processor. The prototype compiler was used to produce code for a critical loop nest (similar to the running example) of an important workstation application. This application was the motivation for developing and implementing this technique in the compiler. This (and similar) kernels showed large (71%) speedups using OLP. Benefits largely accrue from the inner loop trip counts being small and the outer loop trip counts being large. Next, the applicability of this technique was validated using the SPECfp2000 and SPECint2000 benchmark suites (see Table 1). The first column lists the 14 SPECfp2000 benchmarks followed by 12 SPECint2000 benchmarks. The second column shows the total number of loops in each benchmark that are pipelined. This includes loops that are singly nested as well as those that are in loop nests. The third column in the table shows the number of pipelined loops that are in loop nests - these are the loop nests that are candidates for OLP. The fourth column shows

Table 1. Results of Method 2 on SPECfp2000 and SPECint2000

Benchmark	**# of pipelined innermost loops**	**# of pipelined loops within loop nests**	**# of pipelined loops with sibling loops**	**# of OLP loop nests**	**Code size increase due to OLP**
168.wupwise	7	0	0	0	0.0%
171.swim	23	20	15	4	1.5%
172.mgrid	12	10	2	6	0.3%
173.applu	38	38	27	7	0.0%
177.mesa	175	75	52	12	0.4%
178.galgel	258	180	52	50	1.7%
179.art	23	21	16	2	0.8%
183.equake	14	11	7	1	0.0%
187.facerec	59	47	5	38	0.6%
188.ammp	93	38	24	5	2.9%
189.lucas	22	14	2	3	2.7%
191.fma3d	128	50	44	2	0.1%
200.sixtrack	302	262	216	12	0.0%
301.apsi	132	83	40	35	0.8%
164.gzip	52	21	17	0	0.0%
175.vpr	53	29	15	8	1.1%
176.gcc	205	58	31	1	0.0%
181.mcf	20	6	5	0	0.0%
186.crafty	39	14	7	2	0.0%
197.parser	47	21	15	0	0.0%
252.eon	94	72	4	64	0.4%
253.perlbmk	81	44	22	0	0.0%
254.gap	302	102	76	1	0.0%
255.vortex	17	5	1	0	0.0%
256.bzip2	32	13	11	0	0.0%
300.twolf	223	113	72	9	0.8%

the number of pipelined inner loops that have sibling loops in loop nests. Such loops are not candidates for OLP. The fifth column shows the number of loop nests for which OLP was successfully done using Method 2. Unfortunately, the performance gains due to OLP on these benchmarks were negligible, because: (a) Many critical loop nests have large trip counts for the innermost loop. In such cases, the draining of the software pipeline for the innermost loop is not a high overhead, and therefore reducing the cost of draining the pipeline does not contribute to a significant performance gain. (b) Many critical pipelined loops were ineligible for OLP since they had sibling loops in their loop nests. (c) Live values between stages of the software pipelined loop are exposed by OLP. As a result, the register pressure for the code sections outside the innermost loop increases, causing spills in some cases.

These results validate that the overhead of OLP due to adding instructions in post-exits of inner loops is miniscule (column 6). Even the small performance gain possible in loop nests with large trip counts of inner loops was realized.

6 Conclusion

We have presented two methods for Outer Loop Pipelining (OLP) of loop nests that have software-pipelined inner loops. These methods overlap the draining of the software pipeline corresponding to one outer loop iteration with the filling or draining of the software pipeline corresponding to another outer loop iteration. Thus, the software pipeline for the inner loop is filled and drained only once for the loop nest. This is efficiently implemented using the Itanium^{TM} architecture features such as predication, rotating registers, and explicit epilog stage control. This technique is applicable, in a limited sense, to other architectures as well.

Both methods are applicable to perfect as well as imperfect loop nests. The first method does OLP with minimal code expansion, but requires all the outer loops be counted loops with no early exits. The second method does not place such restrictions on the loop nest, but does duplicate the kernel code of the pipelined inner loop where required. These methods have been prototyped in the Intel Optimizing Compiler for Itanium^{TM} architecture. Experimental results indicate good speedups for loop nests with short trip counts for inner loops in an important workstation application. The speedups observed for the SPECfp2000 and SPECint2000 suites of benchmarks were small - this is because their critical loop nests have large trip counts for inner loops.

With the continuing trend of wider and deeply pipelined processors, the availability of parallel execution resources and the latency of instructions will increase. As a result, the prolog/epilog overhead (as a fraction of the execution time of the loop) will increase as well. OLP will be increasingly important as a means to maximize the performance of loop nests.

Acknowledgements. Comments from Dan Lavery, Dong-Yuan Chen, Youfeng Wu, Wei Li, Jean-Francois Collard, Yong-Fong Lee, Sun Chan, and anonymous reviewers helped improve the presentation of this paper. Dan Lavery provided the description of the Itanium^{TM} architecture software pipelining features. Were it not for the plentiful LaTeX support of Kishore Menezes, we'd still be struggling with Word.

References

1. Aiken, A., Nicolau, A.: Optimal Loop Parallelization. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, June, (1988), 308–317
2. Allan, Vicki H., Jones, Reese B., Lee, Randall M., Allan, Stephen J. : Software Pipelining. ACM Computing Surveys, **27**, No. 3, September (1995) 367–432
3. Banerjee, U.: Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Boston, MA, (1993)

4. Charlesworth, A.: An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. IEEE Computer, Sept. (1981).
5. Dehnert, J. C., Hsu, P. Y., Bratt, J. P.: Overlapped Loop Support in the Cydra 5. Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April, (1989), 26–38
6. Ebcioglu, K.: A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture", Dec. (1987), 69–79
7. Eisenbeis, C., et. al: A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops. INRIA TR-RR3337, January (1998)
8. Huck, J., et al: Introducing the IA-64 Architecture. IEEE Micro, **20**, Number 5, Sep/Oct (2000)
9. Intel Corporation: IA-64 Architecture Software Developer's Manual. Santa Clara, CA, April 2000
10. Lam, M. S.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, June, 1988, 318–328
11. Mahlke, S. A., Chen, W. Y., Hwu,W. W., Rau, B. R., Schlansker, M. S.: Sentinel Scheduling for Superscalar and VLIW Processors. Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct, (1992), 238–247
12. Mahlke, S. A., Hank, R. E., McCormick, J.E., August, D. I., Hwu, W. W.: A Comparison of Full and Partial Predicated Execution Support for ILP Processors. Proceedings of the 22nd International Symposium on Computer Architecture, June, (1995), 138–150
13. Rau, B. R., Glaeser, C. D.: Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture, Oct, (1981), 183–198
14. Rau, B. R.: Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. MICRO-27, (1994), 63–74
15. Rau, B. R, Schlansker, M. S., Tirumalai, P. P.: Code Generation Schema for Modulo Scheduled Loops. MICRO-25, (1992), 158–169
16. Ruttenberg, J., Gao, G. R., Stoutchinin, A., Lichtenstein, W. : Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation, May, (1996), 1–11
17. Wolfe, M.: High-Performance Compilers for Parallel Computing. Addison-Wesley, Redwood City, CA, (1996)