

# Software Pipelining of Loops with Early Exits for the Itanium™ Architecture

Kalyan Muthukumar<sup>†</sup> Dong-Yuan Chen<sup>‡</sup> Youfeng Wu<sup>‡</sup> Daniel M. Lavery<sup>∨</sup>

<sup>†</sup>Intel Technology India Pvt Ltd  
17 Mahatma Gandhi Road  
Bangalore 560 001, India

<sup>‡</sup>Intel Microprocessor Research Labs  
2200 Mission College Blvd  
Santa Clara, CA 95052, USA

<sup>∨</sup>Intel Compiler Labs  
2200 Mission College Blvd  
Santa Clara, CA 95052, USA

{kalyan.muthukumar, dong-yuan.chen, youfeng.wu, daniel.m.lavery}@intel.com

## ABSTRACT

The Itanium architecture contains many features to enhance parallel execution, such as an explicitly parallel (EPIC) instruction set, large register files, predication, and support for speculation. It also contains features such as register rotation to support efficient software pipelining of loops. Software-pipelining techniques have been shown to significantly improve the performance of loop-intensive scientific programs that have only one exit per loop. However, loops with multiple exits, which are often encountered in control-intensive non-numeric programs, pose a special challenge to such techniques. This paper describes the schema for generating software-pipelined loops using Itanium architecture features. It then presents two new methods for transforming a loop with multiple exits so that it can be efficiently software-pipelined on Itanium. They make control-flow transformations that convert a loop with multiple exits into a loop with a single exit. This is followed by if-conversion to create a loop consisting of a single basic block. These methods are better than existing techniques in that they result in better values of II (Initiation Interval) for pipelined loops and smaller code sizes. One of these methods has been implemented in the Intel optimizing compiler for the Itanium architecture. This method is compared with the technique suggested by Tirumalai *et al* and we show that it performs better on loops of the benchmark programs in SpecInt2000.

## 1. INTRODUCTION

The Itanium architecture contains many features to enhance parallel execution, such as an explicitly parallel (EPIC) instruction set, large register files, predication [18], and support for speculation [19]. It also contains features such as register rotation [7] to support efficient software pipelining without increasing the code size of the loop. Software pipelining [2][1][3][11][21][12][9] tries to improve the performance of a loop by overlapping the execution of several iterations. This improves the utilization of available hardware resources by increasing the instruction-level parallelism (ILP).

Typically, the loop body that is to be software-pipelined is assumed to consist of a single basic block. This simplifies the task of software pipelining. A loop with control flow in it (and hence containing multiple basic blocks) can be converted to a single basic block by the process of *if-conversion* [8], which removes branches by predicating instructions based on the condition of the branch. However, loops that have early exits

(i.e. that have more than one exit branch out of the loop) cannot be converted into a single basic block by this technique alone. Such loops often occur in control-intensive programs (for example a **break** statement in C/C++) and are also created by compiler transformations such as tail duplication that exclude some paths from the pipelined loop [5].

This paper introduces the Itanium™ architecture support for software pipelining and shows how it can be used to efficiently pipeline loops in control-intensive programs, including while loops and loops with early exits. Two new methods for transforming loops with multiple exits to make them pipelineable are also presented. They make control-flow transformations that convert a loop with multiple exits into a loop with a single exit. This is followed by if-conversion to collapse the remaining control flow and create a loop consisting of a single basic block.

The first method is an improvement of the technique by Tirumalai *et al* [15]. It uses a general-purpose register (as in the Tirumalai method) to “remember” which early exit was taken, if any. This register is examined after the loop terminates to determine which exit was taken and where execution should proceed. Our first method takes advantage of parallel compare instructions in the Itanium architecture to reduce the recurrence MII of the pipelined loop. Our second method uses predicate registers to “remember” which early exit was taken, if any. After the loop terminates, the predicate registers are examined and control branches to the target of the appropriate early exit, if necessary. This method reduces both the resource MII and recurrence MII compared to the method proposed by Tirumalai. We have implemented the second method and the technique of Tirumalai in the Intel optimizing compiler for Itanium. We present results of applying these two techniques on the SPEC CINT2000 suite of benchmarks. The method proposed in this paper results in an average 31% improvement in achieved II compared to Tirumalai’s method for the early exit loops in SPEC CINT2000.

The rest of this paper is organized as follows. Section 1.1 provides some background and motivates the need to perform transformations on loops with early exits so that they can be efficiently pipelined. Section 2 describes the Itanium architecture features and Section 3 describes the software-pipelining schema using these features. The existing method of Tirumalai is presented in Section 4. Sections 5 and 6 present the

two methods that improve the performance of software-pipelined loops that have early exits and compares them with Tirumalai’s method. Section 7 presents the experimental results of applying these techniques to the SPEC CINT2000 benchmarks. Section 8 provides a summary and directions for future work.

## 1.1 Background and Motivation for Early Exit Transformations

The number of cycles between the start of successive iterations in a software-pipelined loop is called the Initiation Interval (II). Software-pipelined loops have three execution phases: the prolog phase, in which the software pipeline is filled, the steady-state kernel phase, in which the pipeline is full, and the epilog phase, in which the pipeline is drained. The software pipeline is coded as a loop that is very different from the original source code loop. To avoid confusion when discussing loops and loop iterations, we use the term source loop and source iterations to refer back to the original source code loop, and the term kernel loop and kernel iterations to refer to the loop that implements the software pipeline.

In this paper we discuss early exit transformations in the context of modulo scheduling [17], though the transformations may apply to other software pipelining algorithms as well. In the modulo-scheduling algorithm a minimum candidate II is computed prior to scheduling. This candidate II is the maximum of the resource-constrained minimum II (resource MII) and the recurrence-constrained minimum II (recurrence MII). The resource MII is based on the resource usage of the loop and the processor resources available. The recurrence MII is based on the cycles in the dependence graph for the loop and the latencies of the processor.

Loops that have control flow pose a special challenge for modulo scheduling. This is because the modulo-scheduling algorithm has generally been developed to schedule either a single basic block or a trace consisting of a single path through the loop body. Loops that contain control flow within the loop body have been handled by one of three methods: if-conversion [7][14], hierarchical reduction [12], or combining modulo scheduling with other software-pipelining algorithms [13][20]. In addition, infrequent or hazardous paths can be excluded from modulo-scheduling altogether [5]. In Intel’s Itanium™ compiler, control flow within the loop body is removed using if-conversion, and tail-duplication is used to exclude some paths from scheduling.

If-conversion alone cannot remove early exits from the loop. Figure 1 shows the overlapped execution of several iterations of a pipelined loop that has one early exit.

The figure shows an early exit branch taken in iteration (i+2). However, some instructions in earlier iterations i and (i+1) have not yet been executed. These are shown by the black areas in the figure. Also instructions from iteration (i+2) that were moved from above to below the early exit during scheduling have not yet been executed. These are shown by the grey area in the figure. Using the pipelining method in [5], an explicit epilog can be created that contains the instructions from the black and gray areas. The target of the early exit is changed to this epilog block. Control then branches from this new block to the original target of the early exit. One such explicit epilog is required for each early exit in the loop. The addition of the epilog blocks

increases the code size. Some of the epilogs may be combined, but the largest epilog, that corresponding to the first early exit, always remains. Lavery’s approach works well for architectures that do not have predication. However, the predication and software pipelining support in the Itanium architecture make it desirable to investigate methods that remove branches and generate kernel-only code (code with no explicit prologs or epilogs [16]).

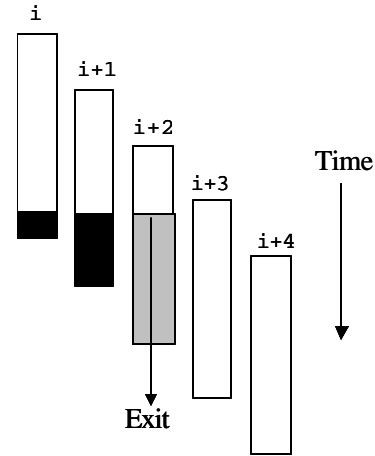


Figure 1. The overlapped execution of a software-pipelined loop with one early exit.

One such method, proposed by Tirumalai et al [15], is to remove the early exit branches but “remember” that the condition for an early exit became true during the execution of the loop. When an early exit occurs, the epilog phase of the pipelined loop is executed in the kernel only code with the appropriate parts of the last source iteration disabled using predication. Upon exit from the loop, control branches to the target of the “remembered” early exit. Thus, the loop with multiple exits is converted to a loop with a single exit. There is still control flow within the loop, but this is removed by the process of if-conversion. Unfortunately, as shown later in this paper, the original transformation method proposed by Tirumalai increases the number of instructions in the loop, increasing the resource MII. The second method proposed in this paper transforms the loop in such a way that it almost always reduces the number of instructions in the original loop. Our method always achieves a resource MII that is equal to or better than the Tirumalai method. It also always achieves a resource MII that is equal to or better than Lavery’s method for architectures like Itanium that have separate compare and branch instructions.

## 2. ITANIUM™ ARCHITECTURE FEATURES

Itanium provides many features to aid the compiler in enhancing and exploiting ILP. These include an explicitly-parallel (EPIC) instruction set, large register files, predication, speculation, and support for software pipelining. An Itanium program must explicitly indicate groups of instructions, called *instruction groups*, that have no register flow or output dependences. Instruction groups are delimited by *architectural stops* in the code. Because instruction groups have no register flow or output dependences, they can be issued in parallel without hardware checks for register dependences between instructions. In the

examples in this paper, architectural stops are indicated by double semicolons (;;). Parallel execution generates many simultaneously live values. The Itanium™ architecture provides 128 general and 128 floating-point registers, so that these results can be kept in registers rather than spilled to memory.

Predication refers to the conditional execution of an instruction based on a boolean source operand called the qualifying predicate. If the qualifying predicate is one, the instruction is executed. If the qualifying predicate is zero, the instruction generally behaves like a no-op. Predicates are assigned values by compare instructions. Predication removes branches and the mispredict penalties associated with them and exposes more ILP [18]. Almost all Itanium instructions have a qualifying predicate. Even conditional branches have qualifying predicates. The branch is taken if the qualifying predicate is one, and not taken otherwise. Itanium provides 64 predicate registers.

Compare instructions in Itanium each have two destination predicate registers. Generally, the first destination is set to one if the compare condition evaluates to true and zero if the condition evaluates to false. The second destination is set to the complement of the first. Three types of compares are used in this paper: conditional, unconditional, and parallel. Conditional compares behave similarly to other predicated Itanium instructions. If the qualifying predicate is zero, the conditional compare leaves both destination predicate registers unchanged. Unconditional compares set both destination predicate registers to zero when the qualifying predicate is zero.

Parallel compares allow the results of two compares to be combined in a way that is analogous to a wired-AND or wired-OR in logic design. Two parallel compares can target the same destination predicate register but still be placed in the same instruction group and issued in parallel. To use the AND-type parallel compare, the destination predicate register is first initialized to one. Then if the condition for a subsequent AND-type compare evaluates to false, the predicate register is set to zero. If the condition evaluates to true, the predicate register is left unchanged. The OR-type parallel compares are similar. The destination predicate register is initialized to zero. Then if the condition for a subsequent OR-type compare evaluates to true, the predicate register is set to one. Parallel compares are useful for implementing the `||` and `&&` logical operators in C and for computing the conditions for the execution of blocks in nested if-then-else constructs. In the following code, the store is executed only if all three conditions are true:

```

cmp.unc p1,p0 = (r1==r2) ;; // unconditional compare
(p1)  cmp.unc p2,p0 = (r3==5) ;; // unconditional compare
(p2)  cmp.unc p3,p0 = (r6>0) ;; // unconditional compare
(p3)  st [r4] = r5

```

Predicate register p0 is hardwired to one and when used as a destination, the result is discarded. Using AND-type parallel compares, all three conditions can be evaluated in parallel:

```

p1 = 1;;
cmp.and p1,p0 = (r1==r2) // AND-type compare
cmp.and p1,p0 = (r3==5) // AND-type compare
cmp.and p1,p0 = (r6>0) ;; // AND-type compare
(p1)  st [r4] = r5

```

Itanium features both control and data speculation. In this paper, we restrict the discussion to control speculation. Control

speculation refers to the execution of an instruction before the branch that guards it. Itanium contains features similar to those first proposed in [19] that allow the compiler to perform speculative code motion. Speculative versions of loads defer exceptions by writing a special token into the destination register to indicate the existence of a deferred exception. Most computation instructions propagate deferred exception tokens from the source to the destination registers. Speculation check instructions are used to test for the presence of a deferred exception token. If a token is found, the check instruction branches to recovery code to re-execute the speculative instruction sequence and handle the exception.

### 3. SOFTWARE PIPELINING IN THE ITANIUM™ ARCHITECTURE

Itanium provides extensive support for software-pipelined loops, including register rotation, and special loop branches and registers. Such features were first seen in the Cydrome Cydra-5 [4][7][6][10]. Register rotation provides a renaming mechanism that eliminates the need for loop unrolling for the purpose of software renaming of registers. Special software-pipelined loop branches support register rotation and, combined with predication, reduce the need to generate separate blocks of code for the prolog and epilog phases. This section describes each of the Itanium™ architecture software pipelining features and shows how they can be used to efficiently pipeline loops in control-intensive programs.

Register rotation renames registers by adding the register number to the value of a rotating register base (`rb`) register modulo the size of the rotating register file. The `rb` register is decremented when a software-pipelined loop branch is executed at the end of each kernel iteration. Decrementing the `rb` register makes the value in register X appear to move to register X+1. 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. Below is an example of register rotation. The `swp_branch` pseudo-instruction represents a software-pipelined loop branch:

```

L1:  ld4    r35 = [r4],4    // post increment by 4
      st4   [r5] = r37,4   // post increment by 4
      swp_branch L1 ;;

```

The value that the load writes to r35 is read by the store two iterations (and two rotations) later as r37. In the meantime, two more instances of the load are executed. Because of register rotation, those instances write their results to different registers and do not destroy the value needed by the store.

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 pipeline. To do this, 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 predicate for the first stage, p17 is defined to be the predicate for the second stage, etc. A possible pipeline schedule for a source iteration of an example counted loop is shown below (before allocation of general and floating-point registers – capital V in the register name indicates that it is a virtual register) with the

stage predicate assignments. Each stage is one cycle long ( $II = 1$ ) and a load latency of 2 is assumed:

```
stage 1: (p16) ld4      Vr4 = [Vr5],4
stage 2: (p17)                // empty stage
stage 3: (p18)  add      Vr7 = Vr4,r9
stage 4: (p19)  st4      [Vr6] = Vr7,4
```

A register rotation takes place at the end of each stage (when the software-pipelined loop branch is executed in the kernel loop). Thus a one written to p16 enables the first stage and then is rotated to p17 at the end of the first stage to enable the second stage for the same source iteration. Each one written to p16 sequentially enables all the stages for a new source iteration. This behavior is used to enable or disable the execution of the stages of the pipelined loop during the prolog, kernel, and epilog phases.

Generally speaking, each time a software-pipelined loop branch is executed, the following actions take place:

1. A decision is made on whether or not to continue kernel loop execution.
2. The registers are rotated (rbr registers are decremented).
3. p16 is set to a value to control execution of the stages of the software pipeline.
4. Special Loop Count (LC) and Epilog Count (EC) registers are selectively decremented.

There are two types of software-pipelined loop branches: counted (br.ctop) and while (br.wtop). For counted loops, during the prolog and kernel phase, a decision to continue kernel loop execution means that a new source iteration is started. p16 is set to one to enable the stages of the new source iteration. LC is decremented to update the count of remaining source iterations. EC is not modified.

Once LC reaches zero, 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. p16 is now set to zero because there are no more new source iterations and the instructions that correspond to non-existent source iterations must be disabled. EC contains the count of the remaining execution stages for the last source iteration and is decremented during the epilog. When the EC register reaches one, the pipeline has been fully drained, and the kernel loop is exited.

A pipelined version of the example counted loop, using Itanium software pipelining features, is shown below assuming two memory units and 200 iterations:

```
mov    lc = 199          // LC = loop count - 1
mov    ec = 4            // EC = epilog stages + 1
mov    pr.rot = 1 << 16 ;; // PR16 = 1, rest = 0

L1: (p16) ld4      r32 = [r5],4
      (p18) add    r35 = r34,r9
      (p19) st4    [r6] = r36,4
      br.ctop    L1 ;;
```

There are several differences in the operation of the while loop branch compared to the counted loop branch. The br.wtop does not access LC. Like ordinary conditional branches in the Itanium™ architecture, a qualifying predicate determines the behavior of this branch instead. If the qualifying predicate is one, the br.wtop is taken and another kernel iteration is

executed. If the qualifying predicate is zero, then the br.wtop is taken if EC is greater than one (i.e. the pipeline is not yet fully drained). Also, p16 is always set to zero. The reasons for these differences are related to the nature of while loops vs. counted loops.

In a purely counted loop, the br.ctop instruction does not depend on any other instructions in the loop, allowing a decision to be quickly made on whether or not to start a new source iteration. In while loops, the decision to execute another source iteration depends on a more general computation ultimately terminating in a compare that sets the qualifying predicate for the br.wtop branch. The computation may be complex and contain loads or other long latency instructions. Thus the br.wtop branch may not be ready for execution until late in the source iteration, resulting in limited overlap of the iterations.

To increase overlap, subsequent iterations can be started speculatively [15] using Itanium's support for control speculation. The software pipeline stages prior to the one containing the compare that sets the br.wtop's qualifying predicate are called the *speculative stages* of the pipeline because it is not possible for the compare and br.wtop branch to control the execution of these stages. Therefore these stages are not assigned stage predicates. The result of the compare is also used as a stage predicate to control the first non-speculative stage of the pipeline, the stage containing the compare itself. The compare rather than the br.wtop generates the predicate that enables the non-speculative portion of the next source iteration.

A possible software pipeline schedule for one iteration of a simple while loop with  $II$  equal to one is shown below (including stage predicate assignments):

```
stage 1: L1:  ld4  Vr4 = [Vr5],4
stage 2:                // empty stage
stage 3: (p18) add  Vr7=Vr4+Vr9 // first non-speculative stage
      (p18) cmp  Vp1,p0 = (Vr4!=0)
stage 4: (p19) st4  [Vr6] = Vr7,4
      (Vp1)br  L1
```

Note that instructions that already have qualifying predicates (such as the br L1 above) are not assigned stage predicates. The pipelined version of the while loop with  $II$  equal to one is shown below. A check for the speculative load is included:

```
mov    ec = 3          // # epilog stages + # speculative
                        pipeline stages
mov    pr.rot = 1 << 16 ;; // PR16 = 1, rest = 0
L1:    ld4.s   r32 = [r5],4 // speculative load
(p18)  chk.s   r34, Recovery // check r34 (result of
                        ld.s) for exception token
(p18)  add    r35 = r34 + r9
(p18)  cmp    p17,p0 = (r34!=0)
(p19)  st4    [r6] = r36,4
(p17)  br.wtop L1 ;;
```

There is no stage predicate assigned to the load because it is speculative. The compare sets p17. This is the branch predicate for the current source iteration and, after rotation, the stage predicate for the first non-speculative stage (stage three) of the next source iteration. During the prolog, the compare cannot produce its first valid result until the third iteration of the kernel loop. The initialization of the predicates provides a pipeline of predicate values that keeps the compare disabled until the first source iteration reaches stage three. At that point the compare

is enabled and starts generating stage predicates to control the non-speculative stages of the pipeline for the next iteration. Notice that the compare is conditional. If it were unconditional, it would always write a zero to p17 and the pipeline would not get started correctly.

During the last source iteration, the compare result is zero. Therefore, during the next kernel iteration the stage predicate for the compare is zero. The conditional compare stops writing results and the stream of zeros written by the br.wtop branch is rotated in to drain the pipeline.

#### 4. TIRUMALAI METHOD FOR EARLY EXIT TRANSFORMATION

This section describes the method of Tirumalai et al [15]. We describe their method in detail so that it can be compared with our methods on the same example program. We use the loop in Figure 2 as a running example. This control flow graph shows a loop with two blocks A and B. Blocks A and B are *exit blocks* of the loop. The successor block of an exit block that is outside the loop is called the *post-exit* block. In this example, block D is the post-exit block of block A and block C is the post-exit block of block B. For simplicity, we only consider a loop that is in a bottom-test form. The *bottom block* (block B) of the loop must be an exit block. An exit block that is not the bottom block is an *early exit*. We consider innermost loops with only one backedge. If the loop has more than one backedge, the loop can be transformed into a nested loop and the innermost loop will have only one backedge.

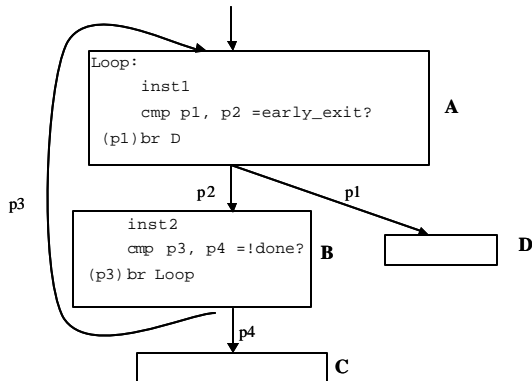


Figure 2. An example loop

Tirumalai et al [15] describes a method that converts a loop with multiple exits into a loop with a single exit. In their method, each exit of the loop is associated with a unique number. When an exit condition is satisfied, its associated number is assigned to a variable, and if the exit is not at the bottom of the loop, control is immediately transferred to the loop exit at the bottom of the loop. After the loop completes and exits from the bottom block, control transfers to the appropriate post-exit block depending on the value in the variable. Figure 3 shows the loop after the control flow transformations of their method.

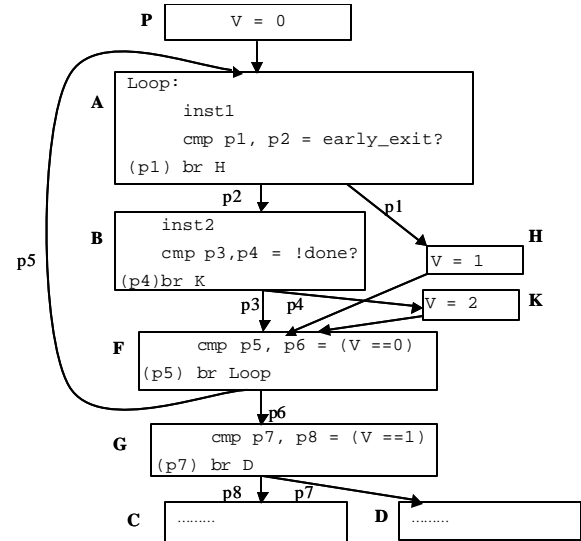


Figure 3. Tirumalai transformation of early exits.

The transformation consists of the following steps:

1. Initialize a new virtual register V to zero in the loop preheader P.
2. Create a new bottom block F after the original bottom block B.
3. Create a new post-exit block G as the new fall-through block for block F.
4. For the exit from block A, create a new block H. The target of block A is changed from D to the new block H. A new instruction V = 1 is inserted into block H. Block H branches to new bottom block F. Similarly for the exit from block B that branches to the post-exit block C, we create a new block K. The target of block B is changed from C to the new block K. A new instruction V = 2 is inserted into block K. Block K branches to the new bottom block F. In general, for a loop with N exits, N blocks containing assignments to V are added.
5. Block F checks whether or not V is zero. If V is zero, it branches back to loop entry. Otherwise, control falls through to post-exit block G.
6. Block G branches to the appropriate target of the original loop exit depending on the value of V. If (V == 1), then the early exit condition is true and control flows to block D. Otherwise, control falls through to block C. In general, if there are N exits, N-1 branches will be inserted in block G.

Notice that this method adds the following three instructions to the loop: V = 1, V = 2, and cmp p5, p6 = (V == 0). In addition, V is initialized to zero in the loop preheader and the loop post-tail contains the compare of V with one. In general, for a loop with N exits, their method adds (N+1) instructions to the loop, one instruction to the loop preheader and (N-1) compare instructions following the loop exit. These additional instructions often increase the resource MII of the software-pipelined loop and may potentially increase the size of the pipelined kernels.

After the transformation, the loop has a single exit. The loop is then if-converted. Figure 4 shows the result of if-converting the transformed loop.

```

V = 0
Loop:
    inst1
    cmp p1, p2 = early_exit?
    (p1) V = 1
    (p2) inst2
    (p2) cmp.unc p3, p4 = done?
    (p3) V = 2
    cmp p5, p6 = (V == 0)
    (p5) br Loop
Block G:
    cmp p7, p8 = (V == 1)
    (p7) br D

```

Figure 4. If-converted code for the example in Figure 2.

Notice that the recurrence MII of the loop transformed by this method is at least 4 cycles, assuming that all the instructions in the recurrence cycle have unit latencies. The dependence cycle is shown in Figure 5. The loop-independent edges are caused by flow dependencies, while the loop-carried edge between “cmp p5, p6=” and “cmp p1, p2=” is caused by control dependence. In general, for a loop with N early exits, this method will have a recurrence MII of at least (N+3) cycles.

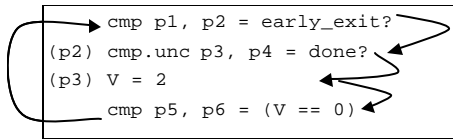


Figure 5. Dependence cycle of the if-converted code in Figure 4.

## 5. SIMPLE IMPROVEMENTS TO TIRUMALAI METHOD

The Tirumalai algorithm can be improved slightly by removing the assignment “V=2” as shown in Figure 6. This reduces resource usage in the loop and also reduces the recurrence MII by one. There are now two dependence cycles with 3 cycles each for the loop.

```

V = 0
Loop:
    p5 = 0
    inst1
    cmp p1, p2 = early_exit?
    (p1) V = 1
    (p2) inst2
    (p2) cmp.unc p3, p4 = !done?
    (p3) cmp p5, p6 = (V == 0)
    (p5) br Loop
Block G:
    cmp p7, p8 = (V == 1)
    (p7) br D

```

Figure 6. Improved code for the example.

We can further improve the above code using the Itanium™ architecture’s *parallel compare* instructions. Using AND-type parallel compares, the two compare instructions “(p2) cmp.unc

p3, p4 = !done?” and “(p3) cmp p5, p6 = (V == 0)” in Figure 6 can be transformed so that they can be executed in a single cycle. This converts the example loop into the code as shown in Figure 7. Although the transformed loop still has a recurrence MII of 3 cycles, it has only one dependence cycle with 3 cycles. Figure 8 shows the longest dependence cycle. In general, for a loop with N early exits, these improvements will lead to a recurrence MII of at least (N+2) cycles. This is an improvement of 1 cycle over the Tirumalai algorithm.

```

V = 0
Loop:
    p5 = 1
    inst1
    cmp p1, p2 = early_exit?
    (p1) V = 1
    (p2) inst2
    (p2) cmp.and p5 = !done?
    cmp.and p5 = (V == 0)
    (p5) br Loop
Block G:
    cmp p7, p8 = (V == 1)
    (p7) br D

```

Figure 7. Improved early exit transformation with parallel compare.

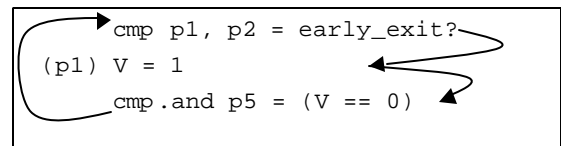


Figure 8. Dependence cycle of the if-converted code in Figure 7.

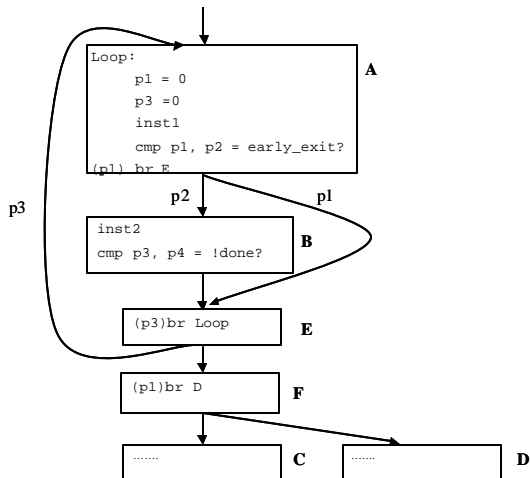
## 6. EARLY EXIT TRANSFORMATION USING PREDICATE REGISTERS

This section presents a new method for transforming loops with multiple exits into a single exit loop for efficient software pipelining. In Tirumalai’s method, a variable is dedicated to record which early exit in the loop has been taken. A closer examination of the original method reveals that the qualifying predicate register for each early exit branch also carries the taken-exit information. With this insight, we have derived a more efficient method for transforming a multiple-exit loop into a single-exit loop by utilizing the predicate registers and unconditional compare instructions in the Itanium™ architecture. The new method assigns one predicate register to each early exit in the loop. The predicate register is set to one whenever its corresponding early exit is taken. Right after the software-pipelined loop, control flow is redirected to the correct target block depending on which of these predicate registers is one. Since the compare instructions for checking the exit conditions already exist in the loop body, new instructions are only introduced for initializing these predicate registers, which can be eliminated by predicate initialization optimization for most cases. Furthermore, as a result of if-conversion, all the early exit branches are removed, resulting in a decrease in the number of instructions in the loop.

## 6.1 Transformations Using Predicate Registers

We will use the example loop of Figure 2 to illustrate the method using predicate registers. The transformation consists of two phases. First, the loop is transformed into a single-exit loop. Then the transformed loop body is collapsed into a single basic block using if conversion. Figure 9 shows the loop in Figure 2 after the first phase of the transformation. The first phase of the transformation consists of the following steps.

1. Split the original bottom block B into two blocks -- B and E. The new bottom block E now contains the loop branch while the other instructions remain in Block B.
2. Redirect the targets of all early exits to the new bottom block E.
3. Create a new block F between the new bottom block E and original post-exit block C, one for each early exit. The new block F contains a new branch to the original target of the early exit, that is, block D. The new branch instruction is predicated by the predicate register assigned to that early exit. For example, in Figure 9 block F contains a branch to block D if p1 is one.
4. Initialize the predicate register (p3 in our example) that controls the loop back branch to zero at the beginning of the loop entry block so that control will exit the loop if an early exit is taken.
5. Initialize all predicates for early exit branches to zero in the beginning of the loop entry block.



**Figure 9. The loop in Figure 2 after transformation using predicate registers.**

Note that not all of the predicate initializations introduced in steps 4 and 5 are really necessary. We will discuss how to optimize away these predicate initializations in Section 6.2.

We then apply if conversion to the resulting control flow graph after the first phase of the transformation, collapsing the loop body into a single basic block. Figure 10 shows the loop from Figure 9 after a straightforward if-conversion.

```

Loop:
    p1 = 0
    p3 = 0
    inst1
    cmp p1, p2 = early_exit?
    (p2) inst2
    (p2) cmp p3, p4 = !done?
    (p3) br Loop
Block_F:
    (p1) br D
    
```

**Figure 10. The if-converted code of Figure 9.**

## 6.2 Optimization of Predicate Initializations

As mentioned before, not all of the predicate initializations are really necessary. Here we describe several approaches to optimize away these predicate initializations. The first optimization is to apply traditional dead code elimination on the if-converted code. For example, in Figure 10, initialization of p1 is killed by the instruction “cmp p1,p2=” before reaching any of its use. Hence “p1=0” is dead code and can be safely removed.

The second optimization makes use of the unconditional form of the compare in the Itanium™ architecture to combine predicate initialization and compare instruction into one single instruction, eliminating the standalone predicate initialization instruction. In the if-converted code of Figure 10, we could replace the compare “(p2) cmp p3,p4=” with an unconditional compare “(p2) cmp.unc p3,p4=”, eliminating the initialization of p3 to 0. When the path ABE in Figure 9 is taken, p2 is one and thus p3 will get its value from the result of the compare instruction in block B. When the path AE is taken, p2 is zero and the unconditional compare will reset p3 to zero.

```

Loop:
    inst1
    cmp p1, p2 = early_exit?
    (p2) inst2
    (p2) cmp.unc p3, p4 = !done?
    (p3) br Loop
Block_F:
    (p1) br D
    
```

**Figure 11. The loop of Figure 10 after predicate initialization optimization**

Figure 11 shows the loop body from Figure 10 after applying optimization on predicate initializations. Note that the merging of a predicate initialization “p=0” and a compare instruction “cmp p,q=” into an unconditional compare instruction is applicable if and only if the following condition holds. That is, all uses of the destination predicate p (or q) after the compare instruction “cmp p,q=” in the if-converted code must be either defined by the predicate initialization instruction “p=0” (or “q=0”) or defined by the compare instruction “cmp p,q=” in the control flow graph before if conversion. This condition prevents the unconditional compare transformation from killing any other reaching definition to the use of predicate p (or q).

Figure 12 illustrates the necessity of this condition. Figure 12(a) is the control flow graph before if conversion and Figure 12(b) shows the if-converted code with “cmp p3,p1=” in block C turned into an unconditional compare “(p2) cmp.unc p3,p1=” for illustration purposes. Assume that the compare “cmp p1,p2=” results in p1=1 and p2=0, that is, execution is expected to follow the path ABD and the loop back branch is to be taken. When

execution reaches “(p2) cmp.unc p3,p1=”, the unconditional compare would reset both p1 and p3 to zero because its qualifying predicate p2 is zero. This would result in an incorrect value of p1=0 when execution reaches “(p1) cmp.unc p4,p5=” and thus the loop would exit prematurely.

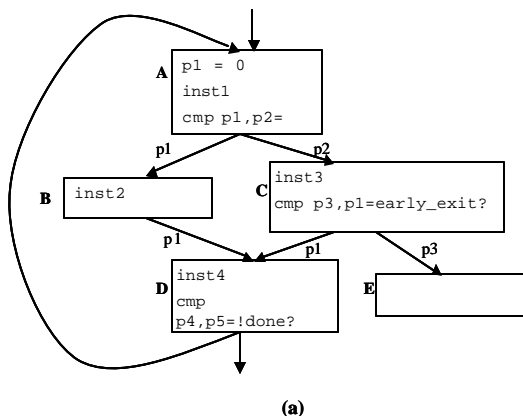
The third optimization applies to early exits caused by **break** statements, such as the code segment shown below.

```

while (.....) {
    if (cond)
        break;
}

```

In such cases, it is not necessary to “remember” which early exit was taken, because the targets of the early exit branches are the same as the fall-through block of the loop’s bottom block. In such cases, the predicates for the early exit branches are not live out of the loop and they need not be computed. Therefore, the predicate initializations for the early exit predicates can be removed and the compares that define such predicates need not be converted into unconditional compares. These cases account for many of the early exits in structured programs, and are well suited for our method.



```

Loop:
    inst1
    cmp p1, p2 =
    (p1) inst2
    (p2) inst3
    (p2) cmp.unc p3, p1 = early_exit?
    (p1) inst4
    (p1) cmp.unc p4, p5 = !done?
    (p4) br Loop
Post_exit:
    (p3) br E

```

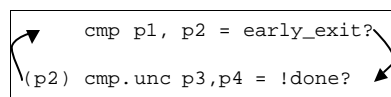
**Figure 12. An example where the predicate initialization cannot be optimized.**

### 6.3 Performance Improvement for Transformation Using Predicate Registers

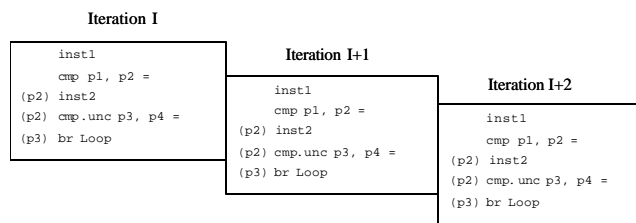
We evaluate the performance improvement for our new transformation using two metrics commonly used to measure the performance of software pipelined loops, namely the Resource MII and Recurrence MII.

Transformation using predicate registers does not add any new instructions to the loop body except predicate initializations, which could be optimized away in most of the cases. Furthermore, early loop exit branches are moved out of the loop body to the loop post-exit, reducing the number of instructions in the loop body. Hence the transformed loop always has a smaller number of instructions in the loop body in comparison with Tirumalai’s algorithm, resulting in the same or a smaller Resource MII.

To illustrate potential improvement in recurrence MII, consider the code in Figure 11. Assume that we are generating kernel-only code for the software-pipelined loop and the latency of compares is one cycle. The two compare instructions in this loop create a dependence cycle. There is a loop-independent data dependence edge from “cmp p1,p2=early\_exit?” instruction to “(p2) cmp.unc p3,p4=!done?” instruction because of the flow dependence on p2. Furthermore a loop-carried control dependence edge from “(p2) cmp.unc p3,p4=!done?” instruction to “cmp p1,p2=early\_exit?” instruction of the next iteration is required if we are generating kernel-only code without explicit epilogs [16]. See Figure 13. The Recurrence MII is at least 2 cycles, which is better than the recurrence MII lower bound in Tirumalai’s algorithm, which is at least 4 cycles. In general, for loop with N early exits, our method results in a minimum recurrence MII of (N+1) cycles as compared to the (N+3) cycles required by Tirumalai’s algorithm.



**Figure 13. Dependence cycle of the code in Figure 11.**



**Figure 14. An example illustrating the necessity of the loop-carried control dependence edge.**

The necessity of the loop-carried control dependence edge for kernel only code can be illustrated using the example in Figure 14. Here we intentionally break the loop-carried control dependence edge between two consecutive iterations by speculatively executing “cmp p1,p2=” from the (I+1)<sup>th</sup> iteration before we know that this iteration will be executed. Figure 14 shows the overlapped execution of three iterations of the transformed loop. Assume that, in the I<sup>th</sup> iteration, predicate p3 is set to zero by “(p2) cmp.unc p3,p4=” because an early exit has been signaled during this iteration which sets the qualifying

predicate p2 to zero. Hence, after iteration I, the software pipelined loop kernel is expected to enter the epilog phase and drain the pipeline by maintaining a zero value in p3. However, the compare instruction “cmp p1,p2=” from the (I+1)<sup>th</sup> iteration is executed *speculatively* before the unconditional compare “(p2) cmp.unc p3,p4=” in iteration I. The execution of “cmp p1,p2=” from iteration (I+1) may speculatively set p2 to one which could then in turn set p3 to one during the epilog phase. This results in incorrect program behavior because the execution of the software pipeline epilog is not maintained until the pipeline is drained.

If the loop-carried control dependence edge is properly honored, the compare instruction “cmp p1,p2=” in iteration (I+1) would have been qualified by a stage predicate that is set to one only if the execution of iteration (I+1) is really required. Thus predicate p1 and p2 in iteration (I+1) will be properly set to zero during the epilog phase, draining the pipeline properly.

Note that the loop-carried control dependence edge is only necessary for generating kernel-only code. Alternatively, we could generate a single explicit epilog as the schema for the software-pipelined loop, eliminating the need for the loop-carried control dependence edge. In this case, we could break the recurrence cycle in both Tirumalai’s and our methods but our method still has a better resource MII. Furthermore, code size may increase if we generate an explicit epilog.

```

ConvertMultipleExitsToSingleExitLoop() {
1. Split the bottom block into two blocks such that the new bottom
   block NB contains only the loop branch;
2. Initialize the loop branch predicate to zero in the loop entry block;
3. FOR every early exit branch B guarded by predicate P and with a
   target block T in the loop DO
   {
4. Change the target of branch B to the
   new bottom block NB;
5. IF (B is not caused by a “break”) {
6. Initialize P to zero at the
   beginning of the loop entry block;
7. Add a new block before the original
   post-exit block containing the
   branch instruction “(P) br T”
   } }

/* Now do if conversion and optimization. */
8. Apply If-conversion to collapse the loop body into a single basic
   block;
9. Apply dead code elimination to remove dead predicate
   initialization;
10. FOR each compare defining predicates p and q DO {
11. IF (p is used later and an unconditional
   definition of p at this location kills
   a previous definition of p other than
   “p=0”)
12. CONTINUE;
13. IF (q is used later and an unconditional
   definition of q at this location kills
   a previous definition of q other than
   “q=0”)
14. CONTINUE;
15. Convert the compare into an unconditional compare;
16. Remove the previous predicate initialization “p=0” and
   “q=0”, if they exist;
17. } }

```

**Figure 15. Outline of transformation algorithm using predicate registers.**

## 6.4 Algorithm

Figure 15 outlines the algorithm for transforming a loop with multiple exits into a single exit loop utilizing predicate registers in the Itanium™ architecture.

## 7. EXPERIMENTAL RESULTS

We have implemented both our method (described in Section 6) and the method of Tirumalai *et al* in the Intel optimizing compiler for Itanium. In this section, we compare the results of using these two techniques on the SPEC CINT2000 suite of benchmark programs. **Table 1** shows the percentage improvements obtained by our method over the technique of Tirumalai *et al*. These results were obtained by running the benchmarks with O2 level optimization. Profile feedback was not used.

The second column shows the total number of loops in each benchmark that is pipelined. This includes loops with early exits as well as loops that only have one exit. The third column in the table shows the number of loops with early exits that are pipelined. The fourth column shows the average percentage improvements in the II of the pipelined loops that have early exits. Performance gains in II vary from 18% for **181.mcf** to 44% for **252.eon**. The fifth column presents the gains in execution times on an Itanium 800 MHz machine. On an average, our method provides a 31% improvement in II for the early exit loops and an overall 1% gain in measured performance on an Itanium 800 MHz machine for all of SPEC CINT2000. These results confirm our claim in Section 6 that our method should lead to better loop performance as compared to the technique of Tirumalai *et al*.

SPEC Cint2000 Benchmarks	Number of pipelined loops	Number of pipelined loops with early exits	Percentage improvement in II for early exit loops	Percentage improvement in execution time on Itanium
164.gzip	50	7	25%	0.6%
175.vpr	90	20	43%	0.0%
176.gcc	707	402	28%	0.9%
181.mcf	22	5	18%	0.0%
186.craftv	84	20	37%	2.2%
197.parser	72	22	28%	0.2%
252.eon	95	15	44%	-0.8%
253.perlbmk	201	108	33%	1.4%
254.gap	578	244	36%	1.4%
255.vortex	39	21	41%	2.1%
256.bzip2	43	5	21%	2.7%
300.vortex	339	70	20%	0.5%
Total/Average	2320	939	31%	1.0%

**Table 1 . Experimental Results**

The percentage improvements in II for the early exit loops are computed *statically* i.e. they do not take into account the number of times these loops are executed. All loops are given the same weight. The values of II for both the methods are those achieved when compiling for the Itanium™ processor. An interesting point that we noticed is that our method provides a greater percentage improvement in II for smaller values of II. This can be explained as follows. First, a loop with larger II is probably constrained by other recurrence cycles or by a resource MII that is a greater than the recurrence cycle caused by the compares of the early exit branches. Therefore, the improvement due to our method does not have an effect on the scheduled II. This was observed in the case of the **300.twolf** benchmark. Second, the reduction in recurrence MII due to our method has a larger impact on loops with smaller II. A reduction of II from 4 cycles to 2 cycles produces a performance gain of 50% while reducing the II from 12 to 10 cycles leads to a gain of only 17%. The

**255.vortex** benchmark has the lowest average II of all the benchmarks and has one of the greatest performance gains of 41% due to our method.

## 8. CONCLUSIONS

We have presented two new methods for transforming a loop with multiple exits so that it can be efficiently software-pipelined on an EPIC architecture like the Itanium™ architecture. They make control flow transformations followed by if-conversion and some optimizations to reduce the number of instructions in the loop. They take advantage of Itanium features such as predicate registers, parallel compares, unconditional compares, etc. They are better than existing techniques for pipelining loops with multiple exits because: (1) the resource MII and recurrence MII of pipelined loops with these new methods are smaller, and (2) they do not generate explicit epilogs, which means reduced code size. The experimental results that we have obtained with one of our methods shows significant improvements in the achieved II values of pipelined loops in SPEC CINT2000 benchmarks over the technique of Tirumalai et al. These methods enable software-pipelining of control-intensive non-numeric programs such as SPEC CINT2000 and database engines and improve their performance on EPIC architectures such as Itanium.

## 9. ACKNOWLEDGEMENTS

Comments from anonymous reviewers helped improve the presentation of the paper. We deeply appreciate the support and encouragement of this work from the management and our colleagues at the Intel Compiler Labs and the Intel Microprocessor Research Labs.

## 10. REFERENCES

- [1] A. Aiken and A. Nicolau. Optimal Loop Parallelization. In Proceedings of PLDI'88 (June 1988), 308-317.
- [2] A. Charlesworth. An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family. IEEE Computer (Sept. 1981).
- [3] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In Proceedings of the 20<sup>th</sup> Annual Workshop on Microprogramming and Microarchitecture (Oct. 1981), 183-198.
- [4] B. R. Rau and D. W. L. Yen and W. Yen and R. A. Towle. The Cydra 5 Departmental Supercomputer". IEEE Computer. Vol 22, No. 1. January 1989.
- [5] D.M. Lavery and W.W. Hwu. Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs. In Proceedings of the MICRO-29 (Dec. 1996), 126-137.
- [6] G. R. Beck, David W.L. Yen, and Thomas L. Anderson. The Cydra 5 Minisupercomputer: Architecture and Implementation. The Journal of Supercomputing. Volume 7, No. 1/2, 1993. Pages 143-180.
- [7] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt. Overlapped Loop Support in the Cydra 5. In Proceedings of ASPLOS'89 (Apr. 1989), 26-38.
- [8] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In Proceedings of POPL'83 (Jan. 1983), 177-189.
- [9] J. Rutenber, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In Proceedings of PLDI'96 (May 1996), 1-11.
- [10] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing. Volume 7, No. 1/2, 1993. Pages 181-228.
- [11] K. Ebcioglu. A Compilation Technique for Software Pipelining of Loops with Conditional Jumps. In Proceedings of the 20<sup>th</sup> Annual Workshop on Microprogramming and Microarchitecture (Dec. 1987), 69-79.
- [12] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In Proceedings of PLDI'88 (June 1988), 318-328.
- [13] M.G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In Proceedings of MICRO-29 (Dec. 1996), 262-273.
- [14] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced Modulo Scheduling for Loops with Conditional Branches. In Proceedings of MICRO-25 (Dec. 1992), 170-179.
- [15] P. Tirumalai, M. Lee, and M. Schlansker. Parallelization of Loops with Exits on Pipelined Architectures. In Proceedings of Supercomputing '90 (Dec. 1990), 200-212.
- [16] Ramakrishna Rau, Michael S Schlansker, and P. P. Tirumalai. Code Generation Schema for Modulo Scheduled Loops. MICRO-25 (1992).
- [17] Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In Proceedings of MICRO-27 (1994).
- [18] S. A. Mahlke, R. E. Hank, J.E. McCormick, D. I. August, and W. W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. In Proceedings of ISCA'95 (June 1995), 138-150.
- [19] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel Scheduling for Superscalar and VLIW Processors. In Proceedings of ASPLOS'92 (Oct. 1992), 238-247.
- [20] S. Shim and S-K Moon. Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows. In Proceedings of MICRO-31 (Nov. 1998), 93-102.
- [21] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. ACM Computing Surveys, 27(3) (Sept. 1995).