Efficient Instruction Sequencing with Inline Target Insertion[1]

Wen-mei W. Hwu, Member IEEE, [2] *and Pohua P. Chang*[3]

Abstract

The trend of deep pipelining and multiple instruction issue has made instruction sequencing an extremely critical issue. This paper defines Inline Target Insertion, a specific compiler and pipeline implementation method for Delayed Branches with Squashing. The method is shown to offer two important features not discovered in previous studies. First, branches inserted into branch slots are correctly executed. Second, the execution returns correctly from interrupts or exceptions with only one program counter. These two features make Inline Target Insertion a superior alternative (better performance and less software/hardware complexity) to the conventional delayed branching mechanisms.

1 Introduction

The instruction sequencing mechanism of a processor determines the instructions to be fetched from the memory system for execution. In the absence of branch instructions, the instruction sequencing mechanism keeps requesting the next sequential instructions in the linear memory space. In this

[1]This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), and the Office of Naval Research under Contract N00014-88-K-0656.

[2]W. W. Hwu is with the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 61801.

[3]P. P. Chang is with the Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR 97124. The work presented in this paper was conducted while he was with the Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Illinois, 61801.

sequential mode, it is easy to maintain a steady supply of instructions for execution. Branch instructions, however, disrupt the sequential mode of instruction sequencing. Without special hardware and/or software support, branches can significantly reduce the performance of pipelined processors by breaking the steady supply of instructions to the pipeline [26].

Many hardware methods for handling branches in pipelined processors have been studied [39][28][9][29][17][10]. An important class of hardware methods, called Branch Target Buffers (or Branch Target Caches), use buffering and extra logic to detect branches at an early stage of the pipeline, predict the branch direction, fetch instructions according to the prediction, and nullify the instructions fetched due to an incorrect prediction[28]. Branch Target Buffers have been adopted by many commercial processors [28][16]. The performance of such hardware methods is determined by their ability to detect the branches early and to predict the branch directions accurately. High branch prediction accuracy, about 85-90% hit ratio, has been reported for hardware methods[39][28][29]. Another advantage of using Branch Target Buffers is that they do not require recompilation or binary translation of existing code. However, the hardware methods suffer from the disadvantage of requiring a large amount of fast hardware to be effective[28][20]. Their effectiveness is also sensitive to the frequency of context switching [28].

Compiler-assisted methods have also been proposed to handle branches in pipelined processors. Table 1 lists three such methods. Delayed Branching has been a popular method to absorb branch delay in microsequencers of microprogrammed microengines. This technique has also been adopted by many recent processor architectures including IBM 801[37], Stanford MIPS[14], Berkeley RISC [33], HP Spectrum [3], SUN SPARC [43], MIPS R2000 [25], Motorola 88000[30], and AMD 29000[1]. In this approach, instruction slots immediately after a branch are reserved as the *delay slots* for that branch. The number of delay slots has to be large enough to cover the delay for evaluating the branch direction. During compile-time, the delay slots following a branch are filled with instructions that are independent of the branch direction, if the data and control dependencies allow such code movement[13]. Regardless of the branch direction, these instructions in the delay slots are always executed. McFarling and Hennessy reported that the first delay slot can be successfully filled by the compiler for approximately 70% of the branches, and the second delay slot can be filled only 25% of the time[29]. It is clear that delayed branching is not effective for processors requiring more than one slot.

Another compiler-assisted method, called Delayed Branches with Squashing, has been adopted by some recent processors to complement delayed branching[29][15][8][30][23]. That is, the method is used when the compiler cannot completely fill the delay slots for delayed branching. In this scheme, the number of slots after each branch still has to be large enough to cover the branch delay. However, instead of moving independent instructions into branch delay slots, the compiler can fill the slots with the predicted successors of the branch. If the actual branch direction differs from the prediction, the instructions in the branch slots are scratched (squashed or nullified) from the pipeline.

On the least expensive side, the hardware predicts all conditional branches to be either always taken (as in Stanford MIPS-X [8]) or always not-taken (as in Motorola 88000 [30]). Predicting all the instructions to be taken achieves about 65% accuracy whereas predicting not-taken does about 35%[39][28] [11]. Predicting all the branches to be either taken or not taken limits the performance of delayed branches with squashing. Furthermore, filling the branch slots for predicted-taken branches requires code copying in general. Predicting all branches to be taken can result in a large amount of code expansion.

McFarling and Hennessy proposed Profiled Delayed Branches with Squashing. In this scheme, an execution profiler is used to collect the dynamic execution behavior of programs such as the preferred direction of each branch[29]. The profile information is then used by a compile-time code restructurer to predict the branch direction and to fill the branch slots according to the prediction. In order to allow each branch to be predicted differently, an additional bit to indicate the predicted direction is required in the branch opcode in general[23]. Through this bit, the compiler can convey the prediction decision to the hardware. McFarling and Hennessy also suggested methods for avoiding adding a prediction bit to the branch opcode. Using pipelines with one and two branch slots, McFarling and Hennessy showed that the method can offer comparable performance with hardware methods at a much lower hardware cost. They suggested that the stability of using execution profile information in compile-time code restructuring should be further evaluated.

This paper examines the extension of McFarling and Hennessy's idea to processors employing deep pipelining and multiple instruction issue. These techniques increase the number of slots for each branch. As a result, four issues arise. First, there are only 3 to 5 instructions between branches in the static program (see Section 4.2) . In order to fill a large number of slots (on the order of ten), one must be able to insert branches into branch slots. Questions arise regarding the correct execution of branches in branch slots. Second, the state information about all branch instructions in the instruction pipeline becomes large. Brute force implementations of return from interrupts and exceptions can involve saving/restoring a large amount of state information of the instruction sequencing mechanism. Third, the code expansion due to code restructuring can be very large. It is important to control such code expansion without sacrificing performance. Fourth, the time penalty for refilling the instruction fetch pipeline due to each incorrectly predicted branch is large. It is very important to show extensive empirical results on the performance and stability of using profile information in compile-time code restructuring. The first three issues were not addressed by McFarling and Hennessy [29]. The second issue was not addressed by previous studies of hardware support for precise interrupt [18] [40].

In order to address these issues, we have specified a compiler and pipeline implementation method for Delayed Branches with Squashing. We refer to this method as Inline Target Insertion to reflect the fact that the compiler restructures the code by inserting predicted successors of branches into their sequential locations. Based on the specification, we show that the method exhibits desirable properties such as simple compiler and hardware implementation, clean interrupt/exception return, moderate code expansion, and high instruction sequencing efficiency. We also provide a proof that Inline Target Insertion is correct. Our correctness proof of filling branch slots with branch instructions is also applicable to a previously proposed hardware scheme [34].

The paper is organized into five sections. Section 2 presents background and motivation for Inline Target Insertion. Section 3 defines the compiler and pipeline implementation, proves the correctness of the proposed implementation, and suggests a clean method to return from interrupt and exception. Section 4 provides empirical results on code expansion control and instruction sequencing efficiency. Section 5 offers concluding remarks regarding the cost-effectiveness and applicability of Inline Target Insertion.

2 Background and Motivation

2.1 Branch Instructions

Branch instructions reflect the decisions made in the program algorithm. Figure 1(a) shows a C program segment which finds the largest element of an array. There are two major decisions in the algorithm. One decides if all the elements have been visited and the other decides if the current element is larger than all the other ones visited so far.

With the register allocation/assignment assumption in Figure 1(b), a machine language program can be generated as given in Figure 2. There are three branches in the machine language program. Instruction D ensures that the looping condition is checked before the first iteration. Instruction I checks if the loop should iterate any more. Instruction F determines if the current array element is larger than all the others visited so far.

The simplified view of the machine language program in Figure 2 highlights the effect of branches. Each arc corresponds to a branch where the head of an arc is the *target instruction*. The percentage on each arc indicates the probability for the corresponding branch to occur in execution. The percentages can be derived by program analysis and/or execution profiling. If the percentage on an arc is greater than 50%, it corresponds to a *likely branch*. Otherwise, it corresponds to an *unlikely branch*.

The instructions shown in Figure 2(a) are *static instructions*. These are the instructions generated by the compilers and machine language programmers. During program execution, each static instruction can be executed multiple times due to loops. Each time a static instruction is executed, it generates a *dynamic instruction*. A dynamic branch instruction which redirects the instruction fetch is called a *taken branch*.

2.2 Instruction Sequencing for Pipelined Processors

The problems with instruction sequencing for pipelined processors are due to the latency of decoding and/or executing branches. A simple hardware example suffices to illustrate the problem of instruction sequencing for pipelined processors. The processor shown in Figure 3 is divided into four stages: instruction fetch (IF), instruction decode (ID), instruction execution (EX), and result write-back (WB). The instruction sequencing logic is implemented in the EX stage. The *sequencing pipeline* consists of the IF, ID, and EX stages of the processor pipeline. When a compare-and-branch[4] instruction is processed by the EX stage[5], the instruction sequencing logic determines the next instruction to fetch from the memory system based on the comparison result.

The dynamic pipeline behavior is illustrated by the timing diagram in Figure 4. The vertical dimension gives the clock cycles and the horizontal dimension the pipeline stages. For each cycle, the timing diagram indicates the pipeline stage in which each instruction can be found.

The pipeline fetches instructions sequentially from memory until a branch is encountered. In Figure 4, the instructions to be executed are $E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow E \rightarrow F$. However, the direction of branch I is not known until cycle 7. By this time instructions J and K have already entered the pipeline. Therefore, in cycle 8 instruction E enters the pipeline while J and K are scratched. The nonproductive cycles introduced by incorrectly fetching J and K reduce the throughput of the pipeline.

2.3 Deep Pipelining and Multiple Instruction Issue

The rate of instruction execution is equal to the clock frequency times the number of instructions executed per clock cycle. One way to improve the instruction execution rate is to increase the clock frequency. The pipeline stages with the longest delay (critical paths) limit the clock frequency. Therefore, subdividing these stages can potentially increase the clock frequency and improve the overall performance. This adds stages in the pipeline and creates a deeper pipeline. For example, if the instruction cache access and the instruction execution limit the clock frequency, subdividing these stages may improve the clock frequency. A timing diagram of the resultant pipeline is shown in Figure 5. Now four instructions are scratched if a compare-and-branch redirects the instruction fetch. For example, $I_2 - I_5$ may be scratched if I_1 redirects the instruction fetch.

Another method to improve instruction execution rate is to increase the number of instructions executed per cycle. This is done by fetching, decoding, and executing multiple instructions per cycle. This is often referred to as *multiple instruction issue* [44] [12] [27] [31] [32] [19] [35] [36] [42] [24] [41] . The timing diagram of such a pipeline is shown in Figure 6. In this example, two

[4]Although the compare-and-branch instructions are assumed in the example, the methods in this paper apply to condition code branches as well.

[5]Although unconditional branch instructions can redirect the instruction fetch at the ID stage, we ignore the optimization in this example for simplicity.

instructions are fetched per cycle. When a compare-and-branch (I_1) reaches the EX stage, five (I_2, I_3, I_4, I_5, I_6) instructions may be scratched from the pipeline.[6]

As far as instruction sequencing is concerned, multiple instruction issue has the same effect as deep pipeling. They both result in increased number of instructions which may be scratched when a branch redirects the instruction fetch.[7] Combining deep pipelining and multiple instruction issue will increase the number of instructions to be scratched to a relatively large number. For example, the TANDEM Cyclone processor requires 14 branch slots due to deep pipeline and multiple instruction issue[16].[8] The discussions in this paper do not distinguish between deep pipelining and multiple instruction issue; they are based on the number of instructions to be scratched by branches.

3 Inline Target Insertion

Inline Target Insertion consists of a compile-time code restructuring algorithm and a run-time pipelined instruction fetch algorithm. The compile-time code restructuring algorithm transforms a sequential program P_s to a parallel program P_p. Inline Target Insertion is correct if the instruction sequence generated by executing P_p on a pipelined instruction fetch unit is identical to that generated by executing P_s on a sequential instruction fetch unit. In this section, we first formally define the sequential instruction fetch algorithm. Then, we formally define the code restructuring algorithm and the pipelined instruction fetch algorithm of Inline Target Insertion. From the formal models of implementation, we will derive a proof of correctness.

3.1 Sequential Instruction Fetch

In a sequential instruction fetch unit, $I_s(t)$ is defined as the dynamic instruction during cycle t. The address of $I_s(t)$ will be referred to as $A_s(I_s(t))$. The target instruction of a branch instruction $I_s(t)$ will be referred to as $target(I_s(t))$. The next sequential instruction of a branch instruction

[6]The number of instructions to be scratched from the pipeline depends on the instruction alignment. If I_2 rather than I_1 were a branch, four instructions (I_3, I_4, I_5, I_6) would be scratched.

[7]A difference between multiple instruction issue and deep pipelining is that multiple likely control transfer instructions could be issued in one cycle. Handling multiple likely control transfer instructions per cycle in a multiple instruction issue processor is not difficult in Inline Target Insertion. The details are not within the scope of this paper.

[8]The processor currently employs an extension to the instruction cache which approximates the effect of a Branch Target Buffer to cope with the branch problem.

$I_s(t)$ will be referred to as $fallthru(I_s(t))$. The sequential instruction fetch algorithm (SIF) is shown below.

```
Algorithm $SIF$ begin
  if ($I_s(t)$ is a taken branch) then
      $A_s(I_s(t+1)) \leftarrow A_s(target(I_s(t)))$;
  else
      $A_s(I_s(t+1)) \leftarrow A_s(I_s(t)) + 1$; [9]
end
```

The *correct successors* of a dynamic instruction $I_s(t)$ is defined as the dynamic instructions to be executed after $I_s(t)$ as specified by SIF. The k^{th} correct successors of $I_s(t)$ will be denoted as $CS(I_s(t), k)$. It should be noted that $CS(I_s(t), k) = I_s(t+k)$. For a sequential program, P_s, whose execution starts from instruction I_0, the instruction sequence is (I_0, $CS(I_0, 1)$, $CS(I_0, 2)$, .., $CS(I_0, n)$), where $CS(I_0, n)$ is the first terminating instruction.

3.2 Compiler Implementation

The compiler implementation of Inline Target Insertion involves compile-time branch prediction and code restructuring. Branch prediction marks each static branch as either likely or unlikely. The prediction is based on the estimated probability for the branch to redirect instruction fetch at the run time. The probability can be derived from program analysis and/or execution profiling. The prediction is encoded in the branch instructions.

The *predicted successors (PS)* of an instruction I are the instructions which tend to execute after I. The definition of predicted successors is complicated by the frequent occurrence of branches. Let $PS(I, k)$ refer to the k^{th} predicted successor of I. The predicted successors of an instruction can be defined recursively:

[9] In the discussions, all address arithmetics are in terms of instruction words. For example, $address \leftarrow address + 1$ advances the $address$ to the next instruction.

1. If I is a likely branch, then $PS(I,1)$ is $target(I)$. Otherwise $PS(I,1)$ is $fallthru(I)$.
2. $(I_1 = PS(I,k)) \wedge (I_2 = PS(I_1,1)) \rightarrow I_2 = PS(I,k+1)$.

For example, one can identify the first five predicted successors of F in Figure 2 as shown below. Since F is a likely branch, its first predicted successor is its target instruction H. The second predicted successor of F is I, which is a likely branch itself. Thus the third predicted successor of F is I's target instruction E.

$$\begin{array}{lcl}
 & & H = PS(F,1) \\
(H = PS(F,1)) \wedge (I = PS(H,1)) & \rightarrow & I = PS(F,2) \\
(I = PS(F,2)) \wedge (E = PS(I,1)) & \rightarrow & E = PS(F,3) \\
(E = PS(F,3)) \wedge (F = PS(E,1)) & \rightarrow & F = PS(F,4) \\
(F = PS(F,4)) \wedge (H = PS(F,1)) & \rightarrow & H = PS(F,5)
\end{array}$$

The code restructing algorithm for Inline Target Insertion is shown below. It is also illustrated by Figure 7.

—————————————————-

```
Algorithm $ITI(N)$ begin
    1. Open $N$ *insertion slots* after every likely branch [10].
    2. For each likely branch $I$, adjust its target label from the address of $PS(I,1)$ to
       (the address of $PS(I,1) + N$).
    3. For each likely branch $I$, copy its first $N$ predicted successors ($PS(I,1)$,$PS(I,2)$,
       ..,$PS(I,N)$) into its slots[11]. If some of the inserted instructions are branches, make
       sure they branch to the same target after copying.[12]
end
```

[10]It is possible to extend the proofs to a non-uniform number of slots in the same pipeline. The details are not in the scope of this paper.

[11]This step can be performed iteratively. In the first iteration, the first predicted successors of all likely branches are determined and inserted. Each subsequent iteration inserts one more predicted successor for all the likely branches. It takes N iterations to insert all the target instructions to their assigned slots.

[12]This is trivial if the code restructuring works on assembly code. In this case, the branch targets are specified as labels. The assembler automatically generates the correct branch offset for the inserted branches.

——————————————————-

The goal of ITI is to ensure that *all original instructions find their predicted successors in the next sequential locations.* This is achieved by inserting the predicted successors of likely branches into their next sequential locations.

We refer to the slots opened by the ITI Algorithm as *insertion slots* instead of more traditional terms such as *delay slots* or *squashing delay slots.* The insertion slots are only associated with likely branches. The instructions in the insertion slots are duplicate copies. All the others are original. This is different from what the terms *delay slots* and *squashing delay slots* usually mean. They often refer to sequential locations after both likely and unlikely branches, which can contain original as well as duplicate copies.

Figure 8 illustrates the application of $ITI(N = 2)$ to a part of the machine program in Figure 2. Step 1 opens two insertion slots for the likely branches F and I. Step 2 adjusts the branch label so that F branches to $H + 2$ and I branches to $E + 2$. Step 3 copies the predicted successors of F (H and I) and I (E and F) into the insertion slots of F (H' and I') and I(E' and F'). Note that the offset is adjusted so that I' and F' branches to the same target instructions as I and F. The reader is encouraged to apply $ITI(N = 3)$ to the code for more insights into the algorithm.

With Inline Target Insertion, each instruction may be duplicated into multiple locations. Therefore, the same instruction may be fetched from one of the several locations. The *original address*, $A_o(I)$, of a dynamic instruction is the address of the original copy of I. The *fetch address*, $A_f(I)$, of a dynamic instruction I is the address from which I was fetched. In Figure 8, the original address of both I and I' is the address of I. The fetch addresses of I and I' are their individual addresses.

It should be noted that ITI moves $fallthru(I)$ of a likely branch I to $A_o(I) + N + 1$ which is an original address.

3.3 Sequencing Pipeline Implementation

The sequencing pipeline is divided into $N + 1$ stages. The sequencing pipeline processes all instructions in their fetch order. If any instruction is delayed due to a condition in the sequencing pipeline (e.g. instruction cache miss), all the other instructions in the sequencing pipeline are delayed. This includes the instructions ahead of the one being delayed. The net effect is that the entire sequencing pipeline freezes. This ensures that the relative pipeline timing among instructions is accurately exposed to the compiler. It guarantees that when a branch redirects instruction fetch, all instructions in its insertion slots have entered the sequencing pipeline. Note that this restriction only applies to the instructions in the sequencing pipeline, the instructions in the execution pipelines (e.g., data memory access and floating point evaluation) can still proceed while the instruction sequencing pipeline freezes.

The definition of time in instruction sequencing separates the freeze cycles from execution cycles. Freeze cycles do not affect the relative timing among instructions in the sequencing pipeline. In this paper, cycle t refers to the t^{th} cycle of program execution excluding the freeze cycles. $I(k,t)$ is defined as the dynamic instruction at the k^{th} stage of the sequencing pipeline during cycle t. The implementation keeps an array of fetch addresses for all the instructions in the sequencing pipeline. The fetch address for the instruction at stage i in cycle t will be referred to as $A_f(I(i,t))$.

A hardware function $REFILL$[13] is provided to reload the instruction fetch pipeline from any original address. $REFILL$ is called when there is a program startup, an incorrect branch prediction, or a return from interrupt/exception. It is easy to guarantee that the program startup address is an original address. We will show in the next subsection that the appropriate original address for a program to resume after incorrect branch prediction and interrupt/exception handling is always available.

```
$REFILL(pc)$ begin
  $A_f(I(N+1,t+1)) \leftarrow pc$;
  for $k = 1..N$ do $A_f(I(N-k+1,t+1)) \leftarrow pc+k$;
end
```

The pipelined instruction fetch algorithm (PIF) that is implemented in hardware is shown below. The sequencing pipeline fetches instructions sequentially by default. Each branch can

[13] $REFILL$ is excluded from the accounting of time when proving correctness of Inline Target Insertion. $REFILL$ may be physically implemented as loading an initial address into $A_f(I(1,t))$ and subsequently computing $A_f(I(1,t+k)) = A_f(I(1,t+k-1))+1$, for $k = 1..N$. $REFILL$ is included in the accounting of time when evaluating the performance of Inline Target Insertion (Section 4).

redirect the instruction fetch and/or scratch the subsequent instructions when it reaches the end of the sequencing pipeline. If a branch redirects the instruction fetch, the next fetch address is the adjusted target address determined in Algorithm ITI. If the decision of a branch is incorrectly predicted, it scratches all the subsequent instructions from the sequencing pipeline.

———————————————————-

```
Algorithm $PIF(N)$ begin
  if ($I(N+1,t)$ is not a branch) then
      $A_f(I(1,t+1)) \leftarrow A_f(I(1,t))+1$;
      for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$;
  else if ($I(N+1,t)$ is likely and is taken) then
      $A_f(I(1,t+1)) \leftarrow A_o(target(I(N+1,t)))+N$;
      for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$;
  else if ($I(N+1,t)$ is unlikely and is not taken) then
      $A_f(I(1,t+1)) \leftarrow A_f(I(1,t))+1$;
      for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$;
  else if ($I(N+1,t)$ is unlikely but is taken) then
      $REFILL(A_o(target(I(N+1,t))))$;
  else if ($I(N+1,t)$ is likely but is not taken) then
      $REFILL(A_f(I(1,t))+1)$;
end
```

———————————————————-

Figure 9(a) shows a timing diagram for executing the instruction sequence ($E \rightarrow F \rightarrow H \rightarrow I \rightarrow E$) of the machine program in Figure 8(a). With Inline Target Insertion (Figure 8(e)), the instruction sequence becomes ($E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E'$). In this case, the branch decision for F is predicted correctly at compile time. When F reaches the EX stage in cycle 4, no instruction is scratched from the pipeline. Since F redirects the instruction fetch, the instruction to be fetched by the IF stage in cycle 5 is E' (the adjusted target of F) rather than the next sequential instruction G.

Figure 9(b) shows a similar timing diagram for executing the instruction sequence ($E \rightarrow F \rightarrow G$). With Inline Target Insertion, the instruction fetch sequence becomes ($E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow G$). In this case, the branch decision for F is predicted incorrectly at the compile time. When F reaches the EX stage in cycle 4, instructions H' and I' are scratched from the pipeline. Since F does not redirect the instruction fetch, the instruction fetch pipeline is refilled from the next sequential instruction G.

3.4 Correctness of Implementation

Branches are the central issue of Inline Target Insertion. Without branches, the sequencing pipeline would simply fetch instructions sequentially. The instructions emerging from the sequencing pipeline would be the correct sequence. Therefore, the correctness proofs of the compiler and pipeline implementation will focus on the correct execution of branches. For pipelines with many slots, it is highly probable to have branches inserted into insertion slots (see Section 4.2). In the case where there are no branches in insertion slots, the correctness follows from the description of the ITI Algorithm. All branch instructions would be original and they would have their first N predicted successors in the next N sequential locations. Whereas a branch instruction in an insertion slot cannot have all its N predicted successors in the next N sequential locations. For example, in Figure 8(e), questions arise regarding the correct execution of F'. When F' redirects the instruction fetch, how do we know that the resulting instruction sequence is always equivalent to the correct sequence $F \rightarrow H \rightarrow I...$?

Definition 1 *Inline Target Insertion is correct if the instruction sequence that is generated by* (PIF, P_p) *is* (I_0, $CS(I_0, 1)$, $CS(I_0, 2)$, .., $CS(I_0, n)$), *where* $CS(I_0, n)$ *is the first stop instruction.*

We shall prove that the instruction sequence that is issued by (PIF, P_p) is identical to that by (SIF, P_s). Unfortunately, it is difficult to compare the output of PIF and SIF on a step by step basis. We will first identify sufficient conditions for (PIF, P_p) to generate the same instruction sequence as (SIF, P_s), and then show that these conditions are guaranteed by Inline Target Insertion.

To help the reader to read the following lemmas and theorems, we list important terms in Table 2. We define two equality relations on the state variables of the instruction fetch pipeline.

R(t): $I(i,t) = PS(I(N+1,t), N-i+1), i = 1..N$.

S(t): $A_f(I(1,t)) = A_o(I(N+1,t)) + N$.

Theorem 1 states that these two equality relations are sufficient to ensure the correctness of Inline Target Insertion.

Theorem 1 *If $R(t)$ and $S(t)$ are true for all t, then $I(N+1,t) = CS(I_0,t)$.*

Proof: The theorem can be proved by induction on t.

$P(t) : I(N+1,t) = CS(I_0,t)$.

Induction basis: *From the definition of $REFILL$, $I(N+1,0) = I_0$. $P(0)$ is true for $t = 0$.*

Induction step: *Assuming $P(t)$ is true, show $P(t+1)$ is also true.*

Case 1: *$I(N+1,t)$ is not an incorrectly predicted branch.*

According to PIF, $I(N+1,t+1) = I(N,t)$. $R(t)$ implies that $I(N,t) = PS(I(N+1,t),1)$. For a correctly predicted instruction $I(N+1,t)$, $PS(I(N+1,t),1)$ is equal to $CS(I(N+1,t),1)$. Hence, $I(N+1,t+1) = I(N,t) = PS(I(N+1,t),1) = CS(I(N+1,t),1) = CS(I_0,t+1)$.

Case 2: *$I(N+1,t)$ is unlikely but is taken.*

PIF performs $REFILL(A_o(target(I(N+1,t))))$ at t. According to the definition of $REFILL$, $I(N+1,t+1)$ becomes $target(I(N+1,t))$ which is $CS(I(N+1,t),1)$. Hence, $I(N+1,t+1) = CS(I(N+1,t),1) = CS(I_0,t+1)$.

Case 3: *$I(N+1,t)$ is likely but is not taken.*

PIF performs $REFILL(A_f(I(1,t))+1)$ at t. According to the definition of $REFILL$ and $S(t)$, $A_f(I(N+1,t+1)) = A_f(I(1,t))+1 = A_o(I(N+1,t))+N+1$. Because $I(N+1,t)$ is a likely branch, ITI allocates N insertion slots after $A_o(I(N+1,t))$, and $fallthru(I(N+1,t))$ is at $A_o(I(N+1,t))+N+1$.[14] *Because $I(N+1,t)$ is not taken, $CS(I(N+1,t),1)$ is $fallthru(I(N+1,t))$. Hence, $I(N+1,t+1) = fallthru(I(N+1,t)) = CS(I(N+1,t),1) = CS(I_0,t+1)$.*

□

[14]It should be noted that, if $I(N+1,t)$ is a likely branch, the original copy of $fallthru(I(N+1,t))$ is always at $A_o(I(N+1,t))+N+1$ according to ITI. Therefore, $A_o(I(N+1,t))+N+1$ is a legal argument for $REFILL$.

Theorem 1 shows that $R(t)$ and $S(t)$ are sufficient to ensure correct execution. Therefore, we formulate the next theorem as the ultimate correctness proof of Inline Target Insertion.

Theorem 2 *ITI and PIF ensure that $R(t)$ and $S(t)$ are true for all t.*

Theorem 2 has a standard induction proof. We start by proving that $R(0)$ and $S(0)$ are true. Then we show that, if $R(t)$ and $S(t)$ are true, $R(t+1)$ and $S(t+1)$ are also true. Because PIF and ITI are complex algorithms, we need to consider several cases in each step of the proof. Instead of presenting the proof as a whole, we will first present several lemmas, from which the proof of Theorem 2 naturally follows.

Lemma 1 *If $REFILL(A_o(I_{entry}))$ is performed at time t so that I_{entry} is $I(N+1, t+1)$ then $R(t+1)$ and $S(t+1)$ are true.*

Proof:

ITI ensures that the original instructions find their N predicted successors in their next N sequential addresses. $R(t+1)$ naturally follows the definition of $REFILL$.

$A_f(I(1,t+1)) = A_f(I(N+1,t+1)) + N$ is implied by the definition of $REFILL$. Because $A_f(I(N+1,t+1)) = A_o(I(N+1,t+1))$, $A_f(I(1,t+1)) = A_o(I(N+1,t+1)) + N$. Therefore, $S(t+1)$ is also true.

$\square$

Lemma 1 shows that refilling the instruction fetch pipeline from an original address ensures that $R(t+1)$ and $S(t+1)$ are true. The instruction sequence pipeline is initialized by $REFILL(A_o(I_0))$, where I_0 is the entry point of a program. It follows from Lemma 1 that $R(0)$ and $S(0)$ are true.

We proceed to prove that, if $R(t)$ and $S(t)$ are true, $S(t+1)$ is also true. We first prove for the case when $I(N+1, t+1)$ is fetched from its original address, and then prove for the case when $I(N+1, t+1)$ is fetched from one of its duplicate addresses.

Lemma 2 *If $R(t)$ and $S(t)$ are true and $A_f(I(N+1,t+1)) = A_o(I(N+1,t+1))$, then $S(t+1)$ is also true.*

Proof:

Since $I(N+1,t+1)$ is fetched from its original address, $I(N+1,t)$ cannot be a likely branch. We need to consider only the following two cases.

Case 1: *$I(N+1,t)$ is not a branch or is an unlikely branch which is not taken.*

PIF performs $A_f(I(1,t+1)) = A_f(I(1,t))+1$ for this case.

Adding 1 to both sides of $S(t)$ results in $A_f(I(1,t))+1 = A_o(I(N+1,t))+N+1$.

Because ITI allocates insertion slots only for likely branches and $I(N+1,t)$ is not a likely branch, the original addresses of $I(N+1,t)$ and $I(N+1,t+1)$ must be adjacent to each other. In other words, $A_o(I(N+1,t))+1 = A_o(I(N+1,t+1))$. Hence, $A_f(I(1,t+1)) = A_f(I(1,t))+1 = A_o(I(N+1,t))+N+1 = A_o(I(N+1,t+1))+N$. Therefore, $S(t+1)$ is true.

Case 2: *$I(N+1,t)$ is an unlikely branch but is taken.*

PIF performs $REFILL(A_o(target(I(N+1,t))))$ at time t. Correctness of $S(t+1)$ follows from Lemma 1. Note that $A_o(target(I(N+1,t)))$ is an original (and therefore legal) address for $REFILL$.

□

The case where $I(N+1,t+1)$ is fetched from an insertion slot is fairly difficult to prove. We will first prove an intermediate lemma.

Lemma 3 *If $A_f(I(N+1,t+1)) \neq A_o(I(N+1,t+1))$, then there must be a k that satisfies all the following four conditions.*

(1) $0 \leq k \leq N-1$.

(2) $I(N+1,t-k)$ is a likely branch.

(3) There can be no likely branches between $I(N+1,t-k+1)$ and $I(N+1,t)$ inclusively.

(4) There is no incorrectly predicted branch between $I(N+1,t-k)$ and $I(N+1,t)$ inclusively.

Proof:

Since $I(N+1,t+1)$ is not fetched from its original address, it must be fetched from an insertion slot. Therefore, there must be at least one likely branch among the N instructions fetched before $I(N+1,t+1)$. The one that is fetched closest to $I(N+1,t+1)$ satisfies (1), (2), and (3).

We can prove (4) by contradiction. Assume that there was an incorrectly predicted branch between $I(N+1,t-k)$ and $I(N+1,t)$ inclusively. Then, a $REFILL$ was performed after $(t-k-1)$ at an original address. Because there was no likely branch between $I(N+1,t-k+1)$ and $I(N+1,t)$ inclusively, $I(N+1,t+1)$ must be fetched from its original address. This is a contradiction to the *precondition of this Lemma:* $A_f(I(N+1,t+1)) \neq A_o(I(N+1,t+1))$. *Therefore, our assumption that there was an incorrectly predicted branch between* $I(N+1,t-k)$ *and* $I(N+1,t)$ *(inclusively) cannot be true.*

$\square$

Lemma 4 *If* $A_f(I(N+1,t+1)) \neq A_o(I(N+1,t+1))$ *and* $R(t)$ *and* $S(t)$ *are true, then* $S(t+1)$ *is also true.*

Proof:

We will use the k *found in Lemma 3.*

Case 1: $k = 0$.[15]

$I(N+1,t)$ is a likely branch. In this case, PIF performs $A_f(I(1,t+1)) = A_o(target(I(N+1,t))) + N$. $R(t)$ *implies that* $I(N,t) = PS(I(N+1,t),1)$. *Because PIF performs* $A_f(I(N+1,t+1)) = A_f(I(N,t))$ *for this case,* $I(N+1,t+1) = PS(I(N+1,t),1) = target(I(N+1,t))$ *and* $A_o(I(N+1,t+1)) = A_o(target(I(N+1,t)))$. *Therefore,* $A_f(I(1,t+1)) = A_o(target(I(N+1,t))) + N = A_o(I(N+1,t+1)) + N$.

Case 2: $1 \leq k \leq N-1$.

(1) Because $I(N+1,t-k)$ *was a likely branch, PIF performed* $A_f(I(1,t-k+1)) = A_o(target(I(N+1,t-k))) + N$.

(2) Because $I(N+1,t-k)$ *was a likely branch,* $I(N,t-k) = target(I(N+1,t-k))$. *Therefore,* $A_o(I(N+1,t-k+1)) = A_o(I(N,t-k)) = A_o(target(I(N+1,t-k))$.

(3) Because there was no likely branch between $I(N+1,t-k+1)$ *and* $I(N+1,t)$ *inclusively,* $A_f(I(1,t+1)) = A_f(I(1,t-k+1)) + k$.

(4) From (1), (2) and (3), $A_f(I(1,t+1)) = A_o(I(N+1,t-k+1)) + N + k$.

(5) Because there was no likely branch between $I(N+1,t-k+1)$ *and* $I(N+1,t)$ *inclusively,* $A_o(I(N+1,t-k+1)) + k = A_o(I(N+1,t+1))$.

(6) From (4) and (5), $A_f(I(1,t+1)) = A_o(I(N+1,t+1)) + N$.

$\square$

[15] Case 1 could be included in Case 2 of the proof. We separate the two cases to make the proof more clear.

Lemma 2 and Lemma 4 collectively ensure that, if $S(i)$ and $R(i)$ are true for $0 \leq i \leq t$, $S(t+1)$ is also true. We proceed to show that $R(t+1)$ is also true.

Lemma 5 *If $R(t)$, $S(t)$, and $S(t+1)$ are true, then $R(t+1)$ is also true.*

Proof:

Case 1: *$I(N+1,t)$ is an incorrectly predicted branch.*

For this case, PIF performs a $REFILL$. Lemma 1 ensures that $I(i,t+1) = PS(I(N+1,t+1), N-i+1), i = 1..N$ after a $REFILL$.

It remains to be shown that the argument to $REFILL$ is an original address. If $I(N+1,t)$ is an unlikely branch, the argument to $REFILL$ is $A_o(target(I(N+1,t)))$ which is an original address.

If $I(N+1,t)$ is a likely branch, the argument to $REFILL$ is $A_f(I(1,t))+1$. According to Lemma 2 and Lemma 4, $A_f(I(1,t))+1 = A_o(I(N+1,t))+N+1$. Because $I(N+1,t)$ is a likely branch, ITI ensures that $A_o(I(N+1,t))+N+1 = (A_o(fallthru(I(N+1,t))))$.

Case 2: *$I(N+1,t)$ is not an incorrectly predicted branch.*

(1) From Lemma 2 and Lemma 4, $A_f(I(1,t+1)) = A_o(I(N+1,t+1))+N$.

(2) According to ITI, an original instruction can find its predicted successors in the next sequential instructions. Therefore, $I(1,t+1)$ must be $PS(I(N+1,t+1),N)$ to be placed in $A_o(I(N+1,t+1))+N$.

(3) Because $I(N+1,t)$ is not an incorrectly predicted branch, PIF performs "for $k = 1..N$ do $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$". Therefore, $R(t)$ implies that $I(i,t+1) = PS(I(N+1,t+1),N-i+1)$ for $i = 2..N$.

(4) From (2) and (3), $R(t+1)$ is true.

□

Proof of Theorem 2 By induction on t. It follows from Lemma 1 that $R(0)$ and $S(0)$ are true. From Lemma 2, Lemma 4, and Lemma 5, if $R(t)$ and $S(t)$ are true, $R(t+1)$ and $S(t+1)$ are also true.

□

3.5 Interrupt/Exception Return

The problem of interrupt/exception return arises when interrupts and exceptions occur to instructions in insertion slots. For example, assume that the execution of code in Figure 8(e) involves an instruction sequence, $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E' \rightarrow F'$. Branch F is correctly predicted to be taken. The question is, if H' caused a page fault, how much instruction sequencing information must be saved so that the process can resume properly after the page fault is handled? If one saved only the address of H', the information about F being taken is lost. Since H' is a not a branch, the hardware would assume that I' was to be executed after H'. Since I' is a likely branch and is taken, the hardware would incorrectly assume that G and H resided in the insertion slots of I'. The instruction execution sequence would become $H' \rightarrow I' \rightarrow G \rightarrow H \rightarrow ...$, which is incorrect.

The problem is that resuming execution from H' violated the restriction that an empty sequencing pipeline always starts fetching from an original instruction. The hardware does not have the information that H' was in the first branch slot of F and that F was taken before the page fault occurred. Because interrupts and exceptions can occur to instructions in all insertion slots of a branch and there can be many likely branches in the slots, the problem cannot be solved by simply remembering the branch decision for one previous branch.

A popular solution to this problem is to save all the previous N fetch addresses plus the fetch address of the re-entry instruction. During exception return, all the $N+1$ fetch addresses will be used to reload their corresponding instructions to restore the instruction sequencing state to before the exception. The disadvantage of this solution is that it increases the number of states in the pipeline control logic and can therefore slow down the circuit. The problem becomes more severe for pipelines with a large number of slots.

In Inline Target Insertion, interrupt/exception return to an instruction I is correctly performed by $REFILL(A_o(I))$. $A_o(I(N+1,t))$ is always available in the form of $A_f(I(1,t))-N$ (Theorem 2). One can record the original addresses when delivering an instruction to the execution units. This guarantees that the original address of all instructions active in the execution units are available. Therefore, when an interrupt/exception occurs to an instruction, the processor can save the original address of that instruction as the return address. Lemma 1 ensures that $R(t+1)$ and $S(t+1)$ are true after $REFILL$ from an original address.

Figure 10 shows the effect of an exception on the sequencing pipeline. Figure 10(a) shows the timing of a correct instruction sequence $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E' \rightarrow F'$ from Figure 8(e) without exception. Figure 10(b) shows the timing with an exception to H'. When H' reaches the end of the sequencing pipeline (EX stage) at t, its $A_o(H')$ is availble in the form of $A_f(I(1,t) = E') - 2$. This address will be maintained by the hardware until H' finishes execution[16]. When an exception is detected, $A_o(H')$ is saved as the return address. During exception return, the sequencing pipeline resumes instruction fetch from H, the original copy of H'. Note that the instruction sequence produced is $H \rightarrow I \rightarrow E'$, which is equivalent to the one without exception.

Note that the original copies must be preserved to guarantee clean implementation of interrupt/exception return. In Figure 8(e), if normal control transfers always enter the section at E', there is an opportunity to remove E and F after Inline Target Insertion to reduce code size. However, this would prevent clean interrupt/exception return if one occurs to E' or F'. Section 4.2 presents an alternative approach to reducing code expansion.

3.6 Extension to Out-of-order Execution

Inline Target Insertion can be extended to handle instruction sequencing for out-of-order execution machines [46] [47] [45] [18] [19] [41] . The major instruction sequencing problem for out-of-order execution machines is the indeterminate timing of deriving branching conditions and target addresses. It is not feasible in general to design an efficient sequencing pipeline where branches always have their conditions and target addresses at the end of the sequencing pipeline. To allow efficient out-of-order execution, the sequencing pipeline must allow the subsequent instructions to proceed whenever possible.

To make Inline Target Insertion and its correctness proofs applicable to out-of-order execution machines, the following changes should be made to the pipeline implementation.

1. The sequencing pipeline is designed to be long enough to identify the target addresses for program-counter-relative branches and for those whose target addresses can be derived without interlocking.

2. When a branch reaches the end of the sequencing pipeline, the following conditions may occur:

[16]The real original address does not have to be calculated until an exception is detected. One can simply save $A_f(I(1,t))$ and only calculate $A_o(I(N+1,t))$ when an exception actually occurs. This avoids requiring an extra subtractor in the sequencing pipeline.

(a) The branch is a likely one and its target address is not available yet. In this case, the sequencing pipeline freezes until the interlock is resolved.

(b) The branch is an unlikely one and its target address is not yet available. In this case, the sequencing pipeline proceeds with the subsequent instructions. Extra hardware must be added to secure the target address when it becomes available to recover from incorrect branch prediction. The execution pipeline must also be able to cancel the effects of the subsequent instructions emerging from the sequencing pipeline for the same reason.

(c) The branch condition is not yet available. In this case, the sequencing pipeline proceeds with the subsequent instructions. Extra hardware must be added to secure the repair address to recover from incorrect branch prediction. The execution pipeline must be able to cancel the effects of the subsequent instructions emerging from the sequencing pipeline for the same reason.

If a branch is program counter relative, both the predicted and alternative addresses are available at the end of the sequencing pipeline. The only difference from the original sequencing pipline model is that the condition might be derived later. Since the hardware secures the alternative address, the sequencing state can be properly recovered from incorrectly predicted branches. If the branch target address is derived from run-time data, the target address of a likely branch may be unavailable at the end of the sequencing pipeline. Freezing the sequencing pipeline in the above specification ensures that all theorems hold for this case. As for unlikely branches, the target address is the alternative address. The sequencing pipeline can proceed as long as the alternative address is secured when it becomes available. Therefore, all the proofs above hold for out-of-order execution machines.

4 Experimentation

The code expansion cost and instruction sequencing efficiency of Inline Target Insertion can only be evaluated empirically. This section reports experimental results based on a set of production quality software from UNIX[17] and CAD domains. The purpose is to show that Inline Target Insertion is

[17]UNIX is a trademark of AT&T.

an effective method for achieving high instruction sequencing efficiency for pipelined processors. All the experiments are based on the an instruction set architecture which closely resembles MIPS R2000/3000[25] with modifications to accommodate Inline Target Insertion. The IMPACT-I C Compiler, an optimizing C compiler developed for deep pipelining and multiple instruction issue at the University of Illinois, is used to generate code for all the experiments [4][21][6][7].

4.1 The Benchmark

Table 3 presents the benchmarks chosen for this experiment. The *C lines* column describes the size of the benchmark programs in number of lines of C code (not counting comments). The *runs* column shows the number of inputs used to generate the profile databases and the performance measurement. The *input description* column briefly describes the nature of the inputs for the benchmarks. The inputs are realistic and representative of typical uses of the benchmarks. For example, the grammars for a C compiler and for a LISP interpreter are two of ten realistic inputs for *bison* and *yacc*. Twenty files of several production quality C programs, ranging from 100 to 3000 lines, are inputs to the *cccp* program. All the twenty original benchmark inputs form the input to *espresso*. The experimental results will be reported based on the mean and sample deviation of all program and input combinations shown in Table 3. The use of many different real inputs to each program is intended to verify the stability of Inline Target Insertion using profile information. The IMPACT-I compiler automatically applies trace selection and placement, and has removed unnecessary unconditional branches via code restructuring [4][6].

4.2 Code Expansion

The problem of code expansion has to do with the frequent occurrence of branches in programs. Inserting target instructions for a branch adds N instructions to the static program.[18] In Figure 8, target insertion for F and I increases the size of the loop from 5 to 9 instructions. In general, if Q is the probability for static instructions to be likely branches (Q = 18% among all the benchmarks), Inline Target Insertion can potentially increase the code size by $N * Q$ (180% for $Q = 18\%$ and

[18]One may argue that the originals of the inserted instructions may be deleted to save space if the flow of control allows. We have shown, however, preserving the originals is crucial to the clean return from exceptions in insertion slots (see Section 3.5).

$N = 10$). Because large code expansion can significantly reduce the efficiency of hierarchical memory systems, the problem of code expansion must be addressed for pipelines with a large number of slots.

Table 4 shows the static control transfer characteristics of the benchmarks. The *static cond.* (*static uncond.*) column gives the percentage of conditional (unconditional) branches among all the static instructions in the programs. The numbers presented in Table 4 confirm that branches appear frequently in static programs. This shows for the need for being able to insert branches in the insertion slots (see Section 3.4). The high percentage of branches suggests that code expansion must be carefully controlled for these benchmarks.

A simple solution is to reduce the number of likely branches in static programs using a threshold method. A conditional branch that executes fewer number of times than a threshold value is automatically converted into an unlikely branch. An unconditional branch instruction that executes a fewer number of times than a threshold value can also be converted into an unlikely branch whose branch condition is always satisfied. The method reduces the number of likely branches at the cost of some performance degradation. A similar idea has been implemented in the IBM Second Generation RISC Architecture[2].

For example, if there are two likely branches A and B in the program. A is executed 100 times and it redirects the instruction fetch 95 times. B is executed 5 times and it redirects the instruction fetch 4 times. Marking A and B as likely branches achieves correct branch prediction 99 (95+4) times out of a total of 105 (100+5). The code size increases by $2 * N$. Since B is not executed nearly as frequently as A, one can mark B as an unlikely branch. In this case, the accuracy of branch prediction is reduced to be 96 (95+1) times out of 105. The code size only increases by N. Therefore, a large saving in code expansion could be achieved at the cost of a small loss in performance.

The idea is that all static likely branches cause the same amount of code expansion but their execution frequency may vary widely. Therefore, by reversing the prediction for the infrequently executed likely branches reduces code expansion at the cost of slight loss of prediction accuracy. This is confirmed by results shown in Table 5. The *threshold* column specifies the minimum dynamic execution count per run, below which, likely branches are converted to unlikely branches. The $E[Q]$ column lists the mean percentage of likely branches among all instructions and the $SD[Q]$

column indicates the sample deviations. The code expansion for a pipeline with N slots is $N * E[Q]$. For example, for ($N = 2$) with a threshold value of 100, one can expect a 2.2% increase in the static code size. Without code expansion control (threshold=0), the static code size increase would be 36.2% for the same sequencing pipeline. For another example, for a 11-stage sequencing pipeline ($N = 10$) with a threshold value of 100, one can expect about 11% increase in the static code size. Without code expansion control (threshold=0), the static code size increase would be 181% for the same sequencing pipeline. Note that the results are based on control intensive programs. The code expansion cost should be much lower for programs with simple control structures such as scientific applications.

4.3 Instruction Sequencing Efficiency

The problem of instruction sequencing efficiency is concerned with the total number of dynamic instructions scratched from the pipeline due to all dynamic branches. Since all insertion slots are inserted with predicted successors, the cost of instruction sequencing is a function of only N and the branch prediction accuracy. The key issue is whether the accuracy of compile-time branch prediction is high enough to ensure that the instruction sequencing efficiency remains high for large values of N.

Evaluating the instruction sequencing efficiency with Inline Target Insertion is straighforward. One can profile the program to find the frequency for the dynamic instances of each branch to go in one of the possible directions. Once a branch is predicted to go in one direction, the frequency for the branch to go in other directions contributes to the frequency of incorrect prediction. Note that only the correct dynamic instructions reach the end of the sequencing pipeline where branches are executed. Therefore, the frequency of executing incorrectly predicted branches is not affected by Inline Target Insertion.

In Figure 11(a), the execution frequencies of F and I are both 100. E and F redirect the instruction fetch 80 and 99 times respectively. By marking F and I as likely branches, we predict them correctly for 179 times out of 200. That is, 21 dynamic branches will be incorrectly predicted. Since each incorrectly predicted dynamic branch creates N nonproductive cycles in the sequencing pipeline, we know that the instruction frequencing cost is 21*N. Note that this number is not changed by Inline Target Insertion. Figure 11(b) shows the code generated by ITI(2). Although we do not know exactly how many times F and F' were executed respectively, we know that their total execution count is 100. We also know that the total number of incorrect predictions for F and F' is 20. Therefore, the instruction sequencing cost of Figure 11(b) can be derived from the count of incorrect prediction in Figure 11(a) multiplied by N.

Let P denote the probability that any dynamic instruction is incorrectly predicted. Note that this probability is calculated for all dynamic instructions, including both branches and non-branches. The average instruction sequencing cost can be estimated by the following equation:

$$relative\ sequencing\ cost\ per\ instruction = 1 + P * N \tag{1}$$

If the peak sequencing rate is $1/K$ cycles per instruction, the actual rate would be $(1 + P * N)/K$ cycles per instruction[19].

Table 4 highlights the dynamic branch behavior of the benchmarks. The *dynamic cond.* (*dynamic uncond.*) column gives the percentage of conditional (unconditional) branches among all the dynamic instructions in the measurement. The dynamic percentages of branches confirm that branch handling is critical to the performance of processors with large number of branch slots. For example, 20% of the dynamic instructions of *bison* are branches. The P value for this program is the branch prediction miss ratio times 20%. Assume that the sequencing pipeline has a peak sequencing rate of one cycle per instruction ($K = 1$) and it has three slots ($N = 3$). The required prediction accuracy to achieve a sequencing rate of 1.1 cycles per instruction can be calculated as follows:

$$1.1 >= 1 + (1 - accuracy) * 0.2 * 3 \tag{2}$$

The prediction accuracy must be at least 83.3%.

Table 6 provides the mean and sample deviation of P for a spectrum of threshholds averaged over all benchmarks. Increasing the threshhold effectively converts more branches into unlikely branches. With $N = 2$, the relative sequencing cost per instruction is 1.036 per instruction for threshhold equals zero (no optimization). For a sequencing pipeline whose peak sequencing rate is one instruction per cycle, this means a sustained rate of 1.036 cycles per instruction. For a sequencing pipeline which sequences k instructions per cycle, this translates into $1.036/k$ (.518

[19]This formula provides a measure of the efficiency of instruction sequencing. It does not take external events such as instruction misses into account. Since such external events freeze the sequencing pipeline, one can simply add the extra freeze cycles into the formula to derive the actual instruction fetch rate.

for $k = 2$) cycles per instruction. When the threshhold is set to 100, the relative sequencing cost per instruction is 1.04. With $N = 10$, the relative sequencing cost per instruction is 1.18 for threshhold equals zero (no optimization). When the threshhold is set to 100, the sequencing cost per instruction instruction becomes 1.20. Comparing Table 5 and Table 6, it is obvious that converting infrequently executed branches into unlikely branches reduces the code expansion at little cost of instruction sequencing efficiency.

5 Conclusion

We have defined Inline Target Insertion, a cost-effective instruction sequencing method extended from the work of McFarling and Hennessy[29]. The compiler and pipeline implementation offers two important features. First, branches can be freely inserted into branch slots. The instruction sequencing efficiency is limited solely by the accuracy of compile-time branch prediction. Second, the execution can return from an interruption/exception to a program with one single program counter. There is no need to reload other sequencing pipeline state information. These two features make Inline Target Insertion a superior alternative (better performance and less software/hardware complexity) to the conventional delayed branching mechanisms.

Inline Target Insertion has been implemented in the IMPACT-I C Compiler to verify the compiler implementation complexity. The software implementation is simple and straightforward. The IMPACT-I C Compiler is used in experiments reported in this paper. A code expansion control method is also proposed and included in the IMPACT-I C Compiler implementation. The code expansion and instruction sequencing efficiency of Inline Target Insertion have been measured for UNIX and CAD programs. The experiments involve the execution of more than a billion instructions. The size of programs, variety of programs, and variety of inputs to each program are significantly larger than those used in the previous experiments.

The overall compile-time branch prediction accuracy is about 92% for the benchmarks in this study. For a pipeline which requires 10 branch slots and fetches two instructions per cycle, this translates into an effective instruction fetch rate of 0.6 cycles per instruction(see Section 4.3). In order to achieve the performance level reported in this paper, the instruction format must give the compiler complete freedom to predict the direction of each static branch. While this can be easily achieved in a new instruction set architecture, it could also be incorporated into an existing architecture as an upward compatible feature.

It is straightforward to compare the performance of Inline Target Insertion and that of Branch Target Buffers. For the same pipeline, the performance of both are determined by the branch prediction accuracy. Hwu, Conte and Chang[20] performed a direct comparison between Inline Target Insertion and Branch Target Buffers based on a similar set of benchmarks. The conclusion was that, without context switches, Branch Target Buffers achieved an instruction sequencing efficiency slightly lower than Inline Target Insertion. Context switches could significantly enlarge the difference[28]. All in all, Branch Target Buffers have the advantages of binary compatibility with existing architectures and no code expansion. Inline Target Insertion has the advantage of not requiring extra hardware buffers, better performance, and performance insensitive to context switching.

The results in this paper do not suggest that Inline Target Insertion is always superior to Branch Target Buffering. But rather, the contribution is to show that Inline Target Insertion is a cost-effective alternative to Branch Target Buffer. The performance is not a major concern. Both achieve very good performance for deep pipelining and multiple instruction issue. The compiler complexity of Inline Target Insertion is simple enough not to be a major concern either. This has been proven in the IMPACT-I C Compiler implementation. If the cost of fast hardware buffers and context switching are not major concerns but binary code compatibility and code size are, then Branch Target Buffer should be used. Otherwise, Inline Target Insertion should be employed for its better performance characteristics and lower hardware cost.

Acknowledgements

The authors would like to thank Michael Loui, Guri Sohi, Nancy Warter, Sadun Anik, Thomas Conte, and all members of the IMPACT research group for their support, comments and suggestions. We also like to thank the anonymous referees for their comments which were extremely helpful in improving the quality of this paper. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, Dr. Lee Hoevel at NCR, the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, the National Aeronautics and

Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), and the Office of Naval Research under Contract N00014-88-K-0656.

References

[1] Advanced Micro Devices, "Am29000 Streamlined Instruction Processor, Advance Information," Publication No. 09075, Rev. A, Sunnyvale, California.

[2] Bakoglu et al, "IBM Second-Generation RISC Machine Organization," Proceedings of the International Conference on Computer Design, pp.138-142, 1989.

[3] J. S. Birnbaum and W. S. Worley, "Beyond RISC: High Precision Architecture", Spring COMPCON, 1986.

[4] P. P. Chang and W. W. Hwu, "Trace Selection for Compiling Large C Application Programs to Microcode", Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitectures, pp.21-29, November, 1988.

[5] P. P. Chang and W. W. Hwu, "Forward Semantic: A Compiler-Assisted Instruction Fetch Method For Heavily Pipelined Processors", Proceedings of the 22nd Annual International Workshop on Microprogramming and Microarchitecture, pp.188-198, August, 1989.

[6] P. P. Chang and W. W. Hwu, "Control Flow Optimization for Supercomputer Scalar Processing", Proceedings of the 1989 International Conference on Supercomputing, June, 1989.

[7] P. Chang, "Aggressive Code Improving Techniques Based on Control Flow Analysis", M.S. Thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, Advisor W. W. Hwu, 1989.

[8] P. Chow and M. Horowitz, "Architecture Tradeoffs in the Design of MIPS-X", Proceedings of the 14th Annual International Symposium on Computer Architecture, June, 1987.

[9] J. A. DeRosa and H. M. Levy, "An Evaluation of Branch Architectures", Proceedings of the 15th International Symposium on Computer Architecture, May, 1988.

[10] D. R. Ditzel and H. R. McLellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero", Proceedings of the 14th Annual International Symposium on Computer Architecture, pp.2-9, June, 1987.

[11] J. Emer and D. Clark, "A Characterization of Processor Performance in the VAX-11/780", Proceedings of the 11th Annual Symposium on Computer Architecture, June, 1984.

[12] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution", IEEE Transactions on Computers, Vol. C-21, pp.1411-1415, December, 1972.

[13] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches", Proceedings of the 15th Microprogramming Workshop, pp.114-120, October, 1982.

[14] J. L. Hennessy, N. Jouppi, F.Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture", Proceedings of the CMU Conference on VLSI Systems and Computations, October 1981.

[15] M. Hill and *etal*, "Design Decisions in SPUR", IEEE Computer, pp.8-22, November, 1986.

[16] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple Instruction Issue in the NonStop Cyclone Processor", Proceedings of the International Symposium on Computer Architecture, May 1990.

[17] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing", Proceedings of the 13th International Symposium on Computer Architecture, pp. 386-395, June 1986.

[18] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines", IEEE Transactions on Computers, vol.C-36, no.12, pp.1496-1514, December, 1987.

[19] W. W. Hwu, "Exploiting Concurrency to Achieve High Performance in a Single-chip Microarchitecture", Ph.D. Dissertation, Computer Science Division Report, no. UCB/CSD 88/398, University of California, Berkeley, January, 1988.

[20] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing Software and Hardware Schemes For Reducing the Cost of Branches", Proceedings of the 16th Annual International Symposium on Computer Architecture, pp.224-231, May, 1989.

[21] W. W. Hwu and P. P. Chang, "Inline Function Expansion for Compiling Realistic C Programs", ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, June, 1989.

[22] W. W. Hwu and P. P. Chang, "Efficient Instruction Sequencing with Inline Target Insertion", Technical Report CSG-103, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, 1990.

[23] Intel, "i860(TM) 64-bit Microprocessor", Order No. 240296-002, Santa Clara, California, April 1989.

[24] N. P. Jouppi, and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.272-282, April, 1989.

[25] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[26] P. M. Kogge, *The Architecture of Pipelined Computers*, pp.237-243, McGraw-Hill, 1981.

[27] D. J. Kuck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-likePrograms and Their Resulting Speedup", IEEE Transactions on Computers, Vol. C-21, pp.1293-1310, December, 1972.

[28] J.K.F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, pp.6-22, January, 1984.

[29] S. McFarling and J. L. Hennessy, "Reducing the Cost of Branches", The 13th International Symposium on Computer Architecture Conference Proceedings, pp.396-404, June, 1986.

[30] C. Melear, "The Design of the 88000 RISC Family", IEEE MICRO, pp.26-38, April, 1989.

[31] A. Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures", IEEE Transactions on Computer, vol.C-33, no.11, pp.968-976. November, 1984.

[32] Y. N. Patt, W. W. Hwu, and M. C. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction", Proceedings of the 18th International Microprogramming Workshop, pp.103-108, December, 1985.

[33] D. A. Patterson and C. H. Sequin, "A VLSI RISC", IEEE Computer, pp.8-21, September, 1982.

[34] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. B. Schechter, "WISQ: A Restartable Architecture Using Queues", Proceedings of the 14th International Symposium on Computer Architecture Conference, pp.290-299, June, 1987.

[35] A. R. Pleszkun, and G. S. Sohi, "Multiple Instruction Issue and Single-chip Processors", Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture, November, 1988.

[36] A. R. Pleszkun, and G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors", Proceedings of the 15th Annual International Symposium on Computer Architecture, pp.37-44, May, 1988.

[37] G. Radin, "The 801 Minicomputer", Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pp.39-47, March, 1982.

[38] R. Rudell, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization", Proc. Cust. Int. Circ. Conf., May, 1985.

[39] J. E. Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8th International Symposium on Computer Architecture, pp.135-148, June, 1981.

[40] J. E. Smith and Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", Proceedings of the 11th Annual Symposium on Computer Architectures, June, 1985.

[41] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instruction Issue", Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.290-302, April, 1989.

[42] G. S. Sohi, and S. Vajapeyam, "Tradeoffs in Instruction Format Design for Horizontal Architectures", Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.15-25, April, 1989.

[43] SUN Microsystems, *The SPARC(TM) Architecture Manual*, SUN Microsystems, Part No. 800-1399-07, Revision 50, Mountain View, California, August 1987.

[44] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computers, vol.c-19, no.10, pp. 889-895, October, 1970.

[45] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors", IEEE Transactions on Computers, vol. C-35, no.9, pp.815-828, September, 1986.

[46] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, vol.11, pp.25-33, January, 1967.

[47] S. Weiss and J. E. Smith, "Instruction Issue Logic in Pipelined Supercomputers", IEEE Transactions on Computers, vol.C-33, pp.1013-1022, IEEE, November, 1984.

Scheme	Hardware features	Compiler features
Delayed branches	None	Fill slots with independent code
Delayed branches with squashing	Uniform prediction and squashing	Fill slots with independent code or instructions from the predicted path
Profiled delayed branches with squashing	Prediction bit and squashing	Execution profiling Fill slots with instructions from the predicted path

Table 1: A summary of delayed branching mechanisms.

$N+1$	The number of stages in the instruction sequencing pipeline
$I(k,t)$	The dynamic instruction occupying the k^{th} pipeline stage at cycle t
$A_f(I)$	The fetch address of dynamic instruction I
$A_o(I)$	The original address of dynamic instruction I
$PS(I,k)$	The k^{th} predicted successor of I
$CS(I,k)$	The k^{th} correct successor of dynamic instruction I
$R(t)$	$I(i,t) = PS(I(N+1,t), N-i+1),\ i = 1..N$
$S(t)$	$A_f(I(1,t)) = A_o(I(N+1,t)) + N$

Table 2: A summary of important definitions used in the proofs.

name	*C lines*	*runs*	*input description*
bison	6913	10	grammar for a C compiler, etc
cccp	4660	20	C programs (100-3000 lines)
cmp	371	16	similar/dissimilar text files
compress	1941	20	same as cccp
eqn	4167	20	papers with .EQ options
espresso	11545	20	original benchmarks [38]
grep	1302	20	exercised various options
lex	3251	4	lexers for C, Lisp, awk, and pic
make	7043	20	makefiles for cccp, compress, etc
tar	3186	14	save/extract files
tbl	4497	20	papers with .TS options
tee	1063	18	text files (100-3000 lines)
wc	345	20	same as cccp
yacc	3333	10	grammar for a C compiler, etc

Table 3: Benchmarks.

	static	static	dynamic	dynamic
benchmark	*cond.*	*uncond.*	*cond.*	*uncond.*
bison	0.12	0.17	0.19	0.01
cccp	0.10	0.11	0.17	0.04
cmp	0.09	0.15	0.16	0.04
compress	0.09	0.14	0.11	0.01
eqn	0.08	0.12	0.21	0.02
espresso	0.09	0.12	0.13	0.02
grep	0.15	0.19	0.30	0.05
lex	0.15	0.16	0.30	0.01
make	0.12	0.14	0.18	0.01
tar	0.10	0.17	0.12	0.00
tbl	0.18	0.20	0.21	0.05
tee	0.09	0.15	0.29	0.07
wc	0.07	0.10	0.22	0.02
yacc	0.14	0.15	0.23	0.01

Table 4: Static and dynamic characteristics. *The high percentage of static unconditional branches is due to the code layout optimization in IMPACT-I CC to reduce the number of likely branches. Note that very few static unconditional branch are executed frequently. This optimization improves the efficiency of both Inline Target Insertion and Branch Target Buffers [20].*

threshold	*E[Q]*	*SD[Q]*
0	18.1%	3.7%
1	4.8%	2.1%
10	2.1%	1.6%
20	1.8%	1.5%
40	1.5%	1.3%
60	1.3%	1.2%
80	1.2%	1.1%
100	1.1%	1.0%
200	0.9%	0.8%
400	0.6%	0.6%
600	0.5%	0.5%

Table 5: Percentage of likely branches among all static instructions. *Unconditional branches are treated as likely branches in this table.*

threshold	*E[P]*	*SD[P]*
0	0.018	0.010
1	0.018	0.010
10	0.019	0.010
20	0.019	0.010
40	0.020	0.010
60	0.020	0.010
80	0.020	0.010
100	0.020	0.010
200	0.023	0.010
400	0.023	0.010
600	0.025	0.011

Table 6: Probability of prediction miss among all dynamic instructions.

(a):
```
MaxElement = 0;
for (i = 0; i < IMax; i++) {
    if (Array[i] > MaxElement) MaxElement = Array[i];
} ...
```

(b):
```
r1 ← i
r2 ← temporary for Array[i]
r3 ← IMax
r4 ← MaxElement
```

Figure 1: (a) An example C program for finding the largest element in Array. (b) The register assignment.

(a)

```
A: r4 ← 0
B: r1 ← 0
C: r3 ← IMax
D: if (r1 ≥ r3) goto J [unlikely]
E: r2 ← Array(r1)
F: if (r2 ≤ r4) goto H [likely]
G: r4 ← r2
H: r1 ← r1 + 1
I: if (r1 < r3) goto E [likely]
J: MaxElement ← r4
K: ...
```

(b)

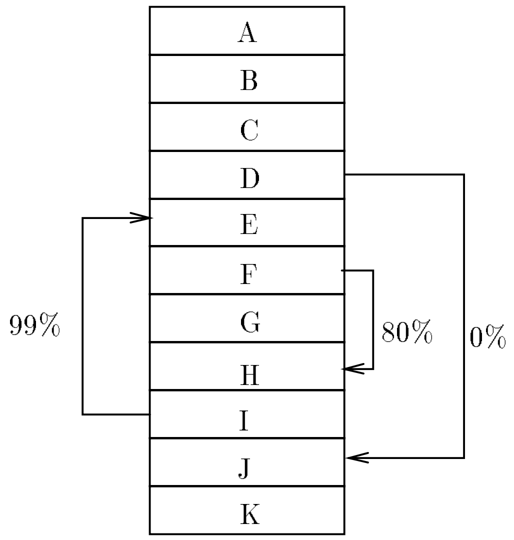

Figure 2: (a) A machine language program generated from the C program shown in Figure 1. (b) A simplified view of the machine language program.

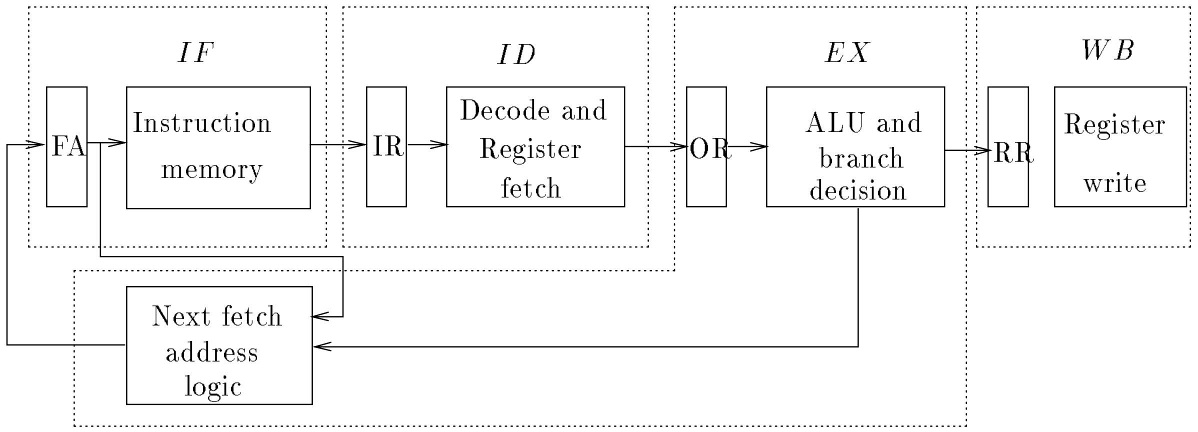

Figure 3: A block diagram and a simplified view of a pipelined processor. FA, IR, OR, RR are pipeline registers *Fetch Address, Instruction Register, Operand Register, and Result Register.*

	IF	ID	EX	WB
1	E			
2	F	E		
3	G	F	E	
4	H	G	F	E
5	I	H	G	F
6	J	I	H	G
7	K	J	I	H
8	E			I
9	F	E		

Figure 4: A timing diagram of the pipelined processor in Figure 3 executing the sequence of instructions $E \rightarrow F \rightarrow G \rightarrow H \rightarrow I \rightarrow E \rightarrow F$ of Figure 2. Instructions J and K are scratched from the pipeline because I is taken.

	IF_1	IF_2	ID	EX_1	EX_2	WB
1	I_1					
2	I_2	I_1				
3	I_3	I_2	I_1			
4	I_4	I_3	I_2	I_1		
5	I_5	I_4	I_3	I_2	I_1	
6	I_6					I_1

Figure 5: A timing diagram of a pipelined processor which results from further dividing the IF and EX stages of the processor in Figure 3.

	IF	ID	EX	WB
1	I_2,I_1			
2	I_4,I_3	I_2,I_1		
3	I_6,I_5	I_4,I_3	I_2,I_1	
4	I_8,I_7			I_1

Figure 6: A timing diagram of the pipelined processor which processes two instructions in parallel.

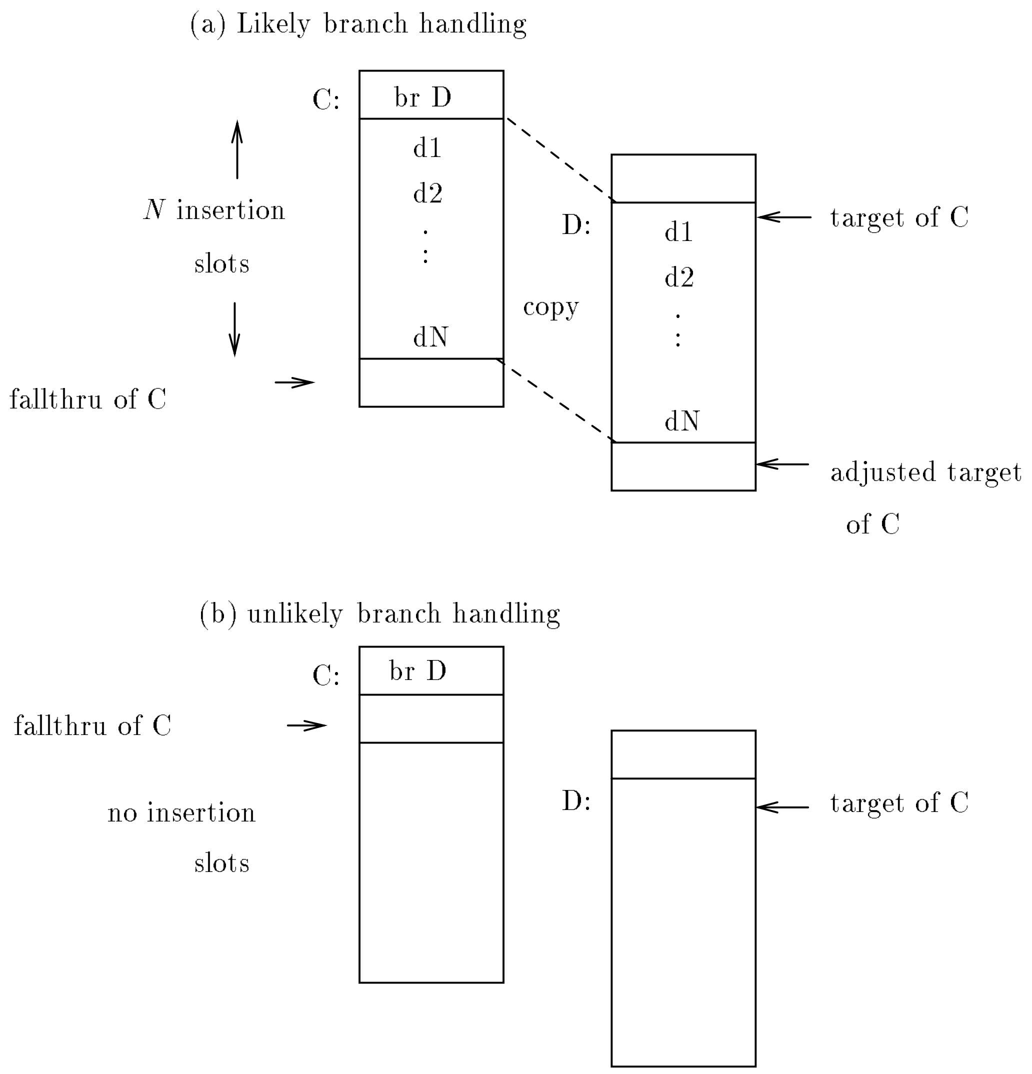

Figure 7: Handling branches in the ITI Algorithm.

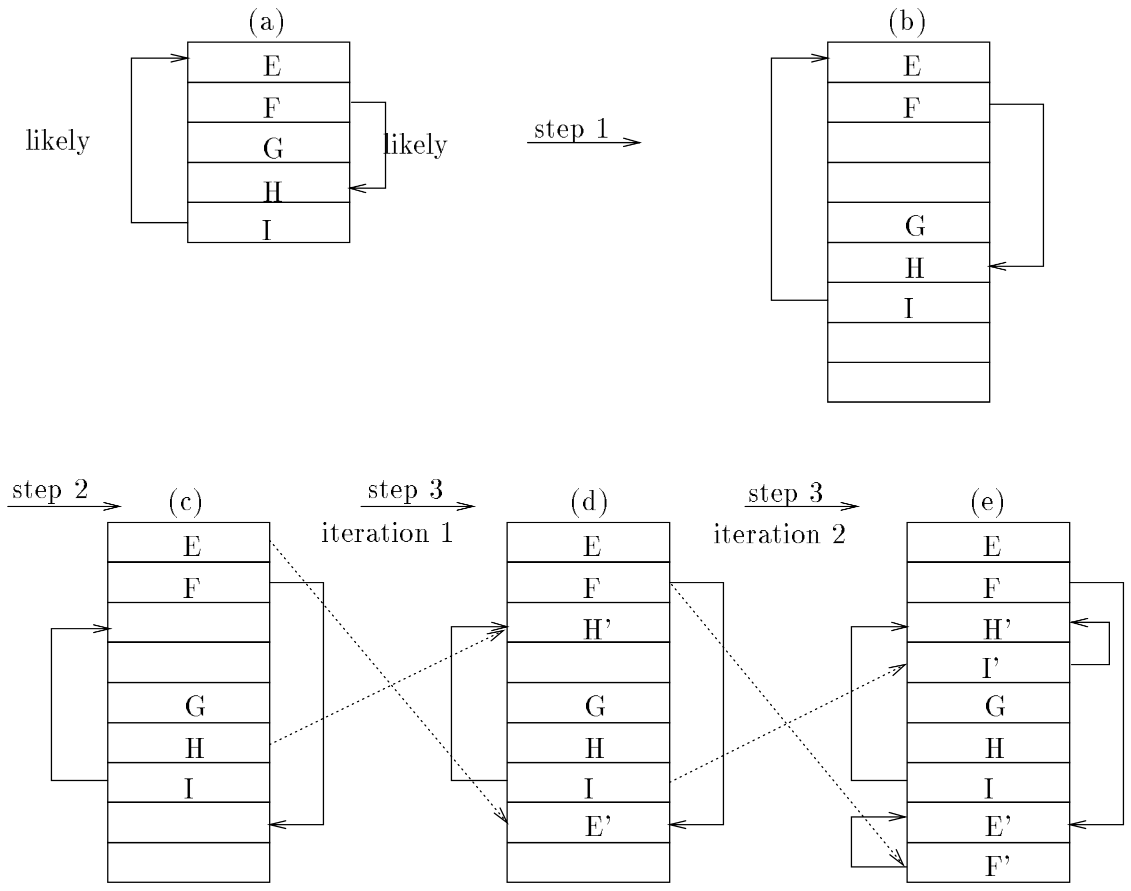

Figure 8: A running example of Inline Target Insertion.

(a)	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F

(b)	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
$REFILL(A_o(G))$				
5	I	H	G	F

Figure 9: (a) Timing diagram of a pipelined processor executing the sequence, $E \rightarrow F \rightarrow H'$ ... of instructions in Figure 8(e). (b) A similar timing diagram for the sequence, $E \rightarrow F \rightarrow G$ ...

(a)	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F
6	F'	E'	I'	H'

(b)	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	E'	I'	H'	F
$REFILL(A_o(H))$				
6	E'	I	H	
7	F'	E'	I	H

Figure 10: (a) Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E'$ of instructions in Figure 8(e). (b) Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow H' \rightarrow I \rightarrow E$ of instructions in Figure 8(e) because of an interrupt at I'.

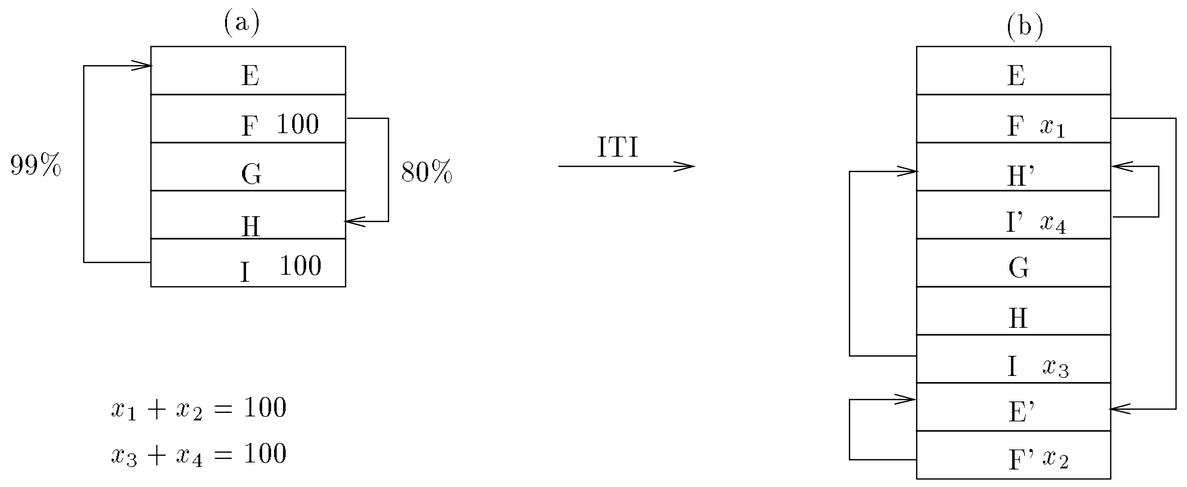

Figure 11: Evaluating the efficiency of instruction sequencing.