May 1990

UILU-ENG-90-2215
CSG-123

COORDINATED SCIENCE LABORATORY
College of Engineering

EFFICIENT INSTRUCTION SEQUENCING WITH INLINE TARGET INSERTION

Wen-mei W. Hwu
Pohua P. Chang

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY none	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE none	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2215 CSG-123	5. MONITORING ORGANIZATION REPORT NUMBER(S) none
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL *(If applicable)* N/A
7a. NAME OF MONITORING ORGANIZATION NSF, NCR, NASA, ONR	
6c. ADDRESS *(City, State, and ZIP Code)* 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS *(City, State, and ZIP Code)* NSF: 1800 G. Street, Washington, DC 20552 NCR: Personal Computer Div.-Clemson 1150 Anderson dr., Liberty, SC 29657
8a. NAME OF FUNDING / SPONSORING ORGANIZATION same as 7a.	8b. OFFICE SYMBOL *(If applicable)* N/A
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NSF: mIP-8809478 NCR: 1-6-41546 NASA: NASA NAG 1-613, ONR: N00014-88-K-0656	
8c. ADDRESS *(City, State, and ZIP Code)* same as 7b	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

Efficient Instruction Sequencing with Inline Target Insertion

12. PERSONAL AUTHOR(S)

Hwu, Wen-mei W. Chang, Pohua P.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ________ TO ________	1990 May	48

16. SUPPLEMENTARY NOTATION

none

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Inline Target Insertion, pipeline implementation, Branch Target Buffers, pipelining

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The trend of deep pipelining and multiple instruction issue has made instruction sequencing an extremely critical issue. Traditionally, compiler-assisted instruction sequencing methods have been considered not suitable for deep pipelining and multiple instruction issue. Hardware methods such as Branch Target Buffers have been proposed for deep pipelining and multiple instruction 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 work. First, branches inserted into branch slots are correctly executed. Therefore, the instruction sequencing efficiency is limited solely by the accuracy of compile-time branch prediction. This feature coupled with highly accurate compile-time branch prediction gives Inline Target Insertion excellent performance characteristics. Second, the execution returns correctly from interrupts or exceptions with only 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

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

7b. NASA Langle Research Center, Hampton, VA 23665
Office of Naval Research, 800 N. Quincy, Arlington, VA 22217

19.
delayed branching mechanisms. The compiler part of Inline Target Insertion has been shown to be straightforward with an implementation in the IMPACT-I C Compiler. A new code expansion control method has been proposed and included in the implementation. The code expansion and instruction sequencing efficiency are measured for real UNIX and CAD programs. The size of programs, the variety of programs, and the variety of inputs to each program are significantly larger than those used in the previous experiments. The stability of code restructuring based on profile information is proved empirically using a large number of diverse inputs to each benchmark program. The results show that Inline Target Insertion achieves high sequencing efficiency at a small cost of code expansion for deep pipelining and multiple instruction issue.

Efficient Instruction Sequencing with Inline Target Insertion

Wen-mei W. Hwu Pohua P. Chang

Center for Reliable and High-Performance Computing

University of Illinois

hwu@csg.uiuc.edu

April 11, 1990

Abstract

The trend of deep pipelining and multiple instruction issue has made instruction sequencing an extremely critical issue. Traditionally, compiler-assisted instruction sequencing methods have been considered not suitable for deep pipelining and multiple instruction issue. Hardware methods such as Branch Target Buffers have been proposed for deep pipelining and multiple instruction 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 work. First, branches inserted into branch slots are correctly executed. Therefore, the instruction sequencing efficiency is limited solely by the accuracy of compile-time branch prediction. This feature coupled with highly accurate compile-time branch prediction gives Inline Target Insertion excellent performance characteristics. Second, the execution returns correctly from interrupts or exceptions with only 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. The compiler part of Inline Target Insertion has been shown to be straightforward with an implementation in the IMPACT-I C Compiler. A new code expansion control method has been proposed and included in the implementation. The code expansion and instruction sequencing efficiency are measured for real UNIX and CAD programs. The size of programs, the variety of programs, and the variety of inputs to each program are significantly larger than those used in the previous experiments. The stability of code restructuring based on profile information is proved empirically using a large number of diverse inputs to each benchmark program. The results show that Inline Target Insertion achieves high sequencing efficiency at a small cost of code expansion for deep pipelining and multiple instruction issue.

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 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 [Kogg81].

Many hardware methods for handling branches in pipelined processors have been studied [Smith81] [Lee84] [DeRo88] [McFa86] [Hsu86] [Ditz87]. An important class of hardware methods, called Branch Target Buffers (or Branch Target Caches), employ 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 scratch the instructions fetched due to an incorrect prediction[Lee84]. Branch Target Buffers have been adopted by many commercial processors [Lee84][Hors90][Amd]. The performance of the hardware methods is determined by their capability 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[Smith81][Lee84][McFa86]. 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[Lee84][Hwu89a]. Their effectiveness is also sensitive to the frequency of context switching [Lee84].

Compiler-assisted methods have also been proposed to handle branches in pipelined processors. For example, *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[Radin 82], Stanford MIPS[Henn81], Berkeley RISC [Patt82], HP Spectrum [Birn86], SUN SPARC [Sun87], MIPS R2000 [Kane87], Motorola 88000[Mele89], AMD 29000[Amd], and Intel i860[Inte89]. 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[Gros82]. 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[McFa86]. 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[McFa86][Hill86][Chow87] [Mele89]. That is, the method is used when the compiler cannot 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 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) from the pipeline.

On the least expensive side, the hardware predicts all conditional branches to be either always taken (as in Stanford MIPS-X [Chow87]) or always not-taken (as in Motorola 88000 [Mele89]). Predicting all the instructions to be taken achieves about 65% accuracy whereas predicting not-taken does about 35%[Smith81][Lee84] [Emer84]. The compiler simply fills the branch slots according

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 predicted successors
Profiled delayed branches with squashing	Prediction bit and squashing	Execution profiling Fill slots with predicted successors

Table 1: A summary of delayed branching mechanisms.

to the hardware prediction. 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 require 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[McFa86]. 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. Through this bit, the compiler can convey the prediction decision to the hardware. McFarling and Hennessy also suggested methods for avoiding adding 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, multiple instruction issue, and high-bandwidth low-cost memory. 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 of instruction sequencing 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 number of bubbles created 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 [McFa86].

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, proof of correctness, clean interrupt/exception return, moderate code expansion, and high instruction sequencing efficiency. Our correctness proof of filling branch slots with branch instructions is also applicable to a previously proposed hardware scheme [Ples87].

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.

```
(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.

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.

(a)

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

(b)

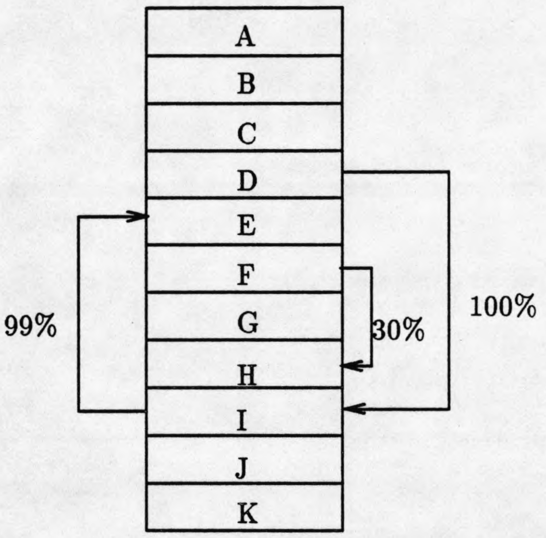

Figure 2: a) A machine language program generated from the C program shown in Figure 1. b) A simplified view of the machine language program.

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*. The *correct successors* of a dynamic instruction I is defined as the dynamic instructions to be executed after I as specified by the instruction set architecture. The k^{th} correct successor of I will be denoted as $S(I,k)$. 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[1] instruction is processed by the EX stage[2], 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.

Without branches, the pipeline fetches instructions sequentially from memory. 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 bubbles created by incorrectly fetching J and K reduce the throughput of the pipeline.

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

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

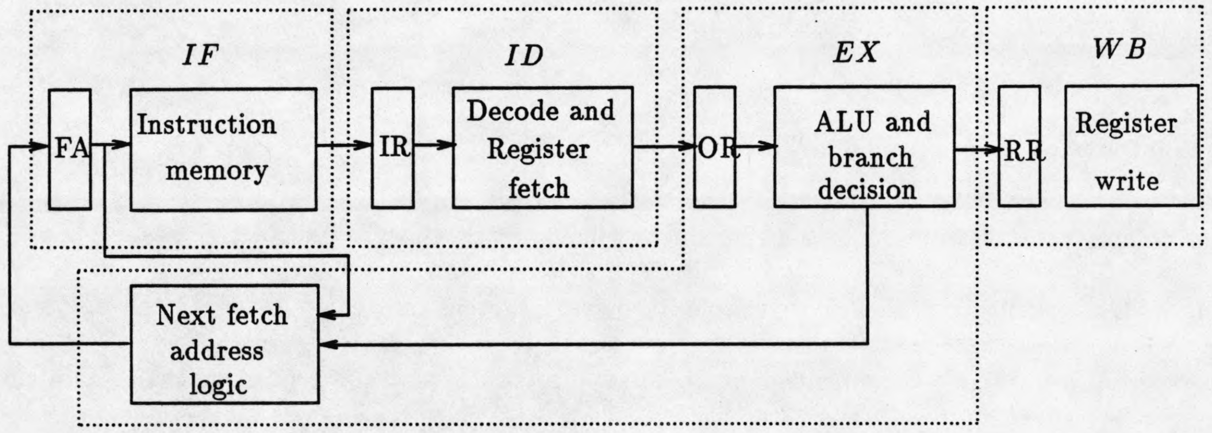

Figure 3: A block diagram and a simplified view of a pipelined processor. FA, IR, OR, RR are pipeline registers *Fetch Address, Instruction Register, Operand Register, and Result Register.*

	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.

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*[Tjad70] [Fost72][Kuck72][Nico84][Patt85][Hwu88] [Ples88][Smith89]. The timing diagram of such a pipeline is shown in Figure 6. In this example, two instructions are fetched per cycle. When a compare-and-branch (I_1) reaches the EX stage,

	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.

five $(I_2, I_3, I_4, I_5, I_6)$ instructions may be scratched from the pipeline.[3]

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.[4] 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[Hors90][5] 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.

2.4 High-Bandwidth Low-Cost Instruction Memory

Instruction caches have been adopted in many high performance processors. To support an execution rate of one instruction per cycle, most of the instruction caches provide single cycle access.

[3]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.

[4]An issue which distinguishes multiple instruction issue from 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.

[5]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.

Instruction caches work well when processors are implemented with mature technology which can accommodate large on-chip caches. They also work fairly well when the main-stream memory technology can provide external single-cycle access caches at a reasonable cost.

There are, however, at least two situations where high-bandwidth low-cost instruction memories (such as Video RAMs) [Nico88] may be preferred to instruction caches. One is in applications which require highly predictable instruction access time (due to real-time requirements), high instruction access bandwidth (due to high performance requirements), and low-cost memory system (to enable large volume production). Instruction caches with unpredictable cache misses can not provide predictable instruction access time. The cost for a Video RAM based instruction memory to support the same size and bandwidth is much lower than that for an instruction cache. For example, Advanced Micro Devices recommends using Video RAMs for constructing low-cost memory systems for the AMD29000-based products. The other situation is when an aggressive technology (e.g. GaAs) is used to build extremely high-performance processors without room for on-chip caches. In this case, it may be too expensive to build off-chip caches which provide single cycle access. High-bandwidth low-cost memories such as Video RAMs, on the other hand, may have the capability to provide one instruction every cycle at a much lower price.

High-bandwidth low-cost memories can be treated as pipelined memories. It takes several cycles to perform an initial access. Once the initial data is available, one can perform single cycle access to its sequential locations. In the context of instruction fetch, the first instruction access takes several (typically three) cycles but the subsequent sequential accesses complete in single cycle. Branches cause performance problem by disrupting the sequential fetch pattern. Fetching the target instruction of a taken branch involves the initial access delay in general. For example, there is a Branch Target Cache on AMD29000 to provide the first three successors of a taken branch to cover the initial latency for accessing the target instructions. Since the first three successors are supplied by the Branch Target Cache, the external instruction memory are accessed starting with the fourth successor of the branch.

In this paper, we model high-bandwidth low-cost memories with multiple pipeline stages for instruction fetch. While this model may not be exact in general, due to some boundary conditions, compile-time code restructuring together with hardware timing design can make Video RAMs behave exactly like a pipelined memory[Chan90]. As far as instruction sequencing is concerned, the use of high-bandwidth low-cost memory increases the depth of the instruction sequencing pipeline. Therefore, it increases the number of required branch slots. The question is whether we can achieve high performance with high-bandwidth low-cost instruction memories using clever compile-time code restructuring methods.

3 Inline Target Insertion

This section addresses three basic theoretical issues of Inline Target Insertion: formal models of implementation, proof of correctness, and return from interrupt/exception.

3.1 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 and passed on to the run-time hardware.

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

1. If I is a likely branch, then $T(I,1)$ is the target instruction of I. Otherwise $T(I,1)$ is the next sequential instruction of I.

2. $(I_1 = T(I,k)) \wedge (I_2 = T(I_1,1)) \rightarrow I_2 = T(I,k+1)$

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.

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

The code restructing algorithm is shown below. It is also illustrated by Figure 7. The goal 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.

Algorithm $ITI(N)$

1. Open N *insertion slots* after every likely branch [6].

2. Adjust the target label of the likely branches so that a likely branch I will branch to $T(I, N+1)$ rather than $T(I, 1)$ [7].

3. Copy the first N predicted successors of each likely branch into its slots[8]. If some of the inserted instructions are branches, make sure they branch to the same target after copying.[9]

Note that we referred 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. It is a compile-time concept. Only instructions in the insertion slots can be 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.

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 readers are encouraged to apply ITI(N=3) to the code for more insights into the algorithm.

[6]It is possible to extend the proofs to non-uniform number of slots in the same pipeline. The details are out side the scope of this paper.

[7]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.

[8]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.

[9]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.

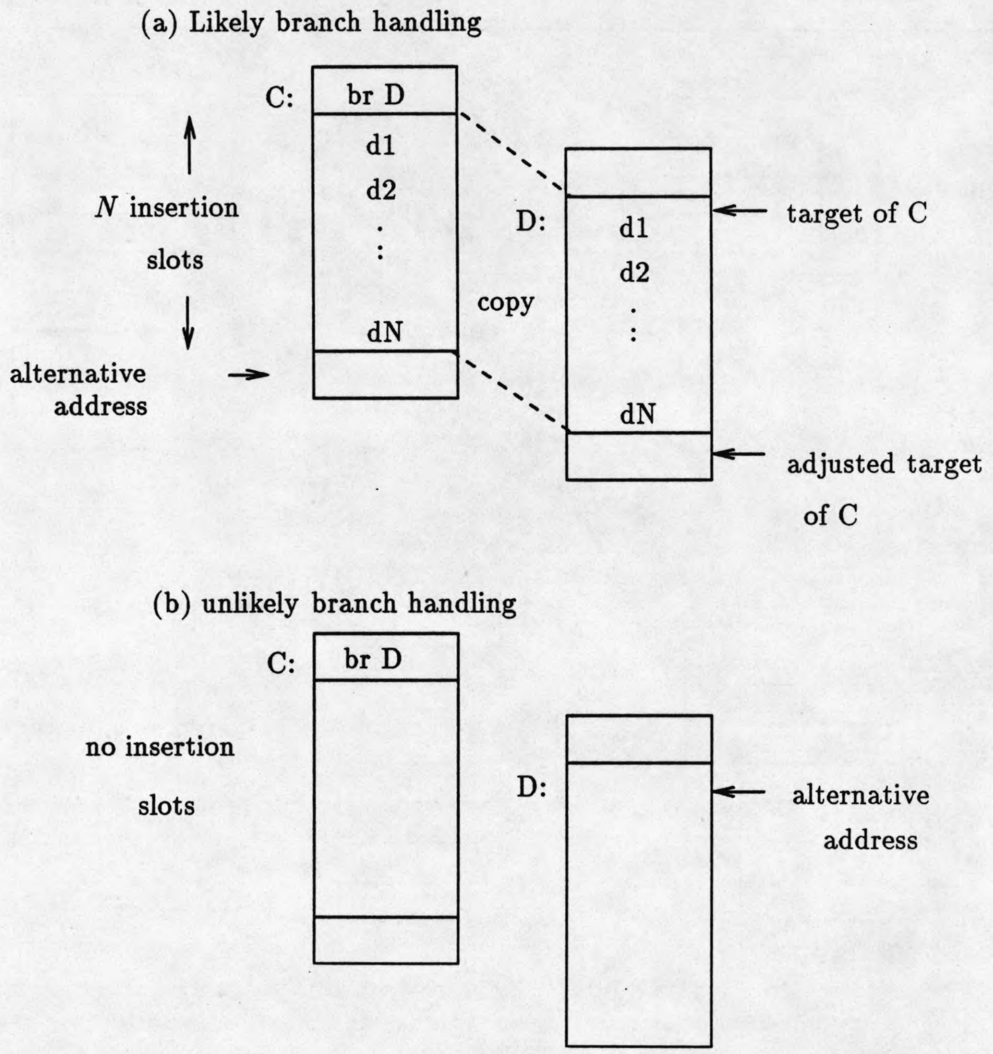

Figure 7: Handling branches in the ITI Algorithm.

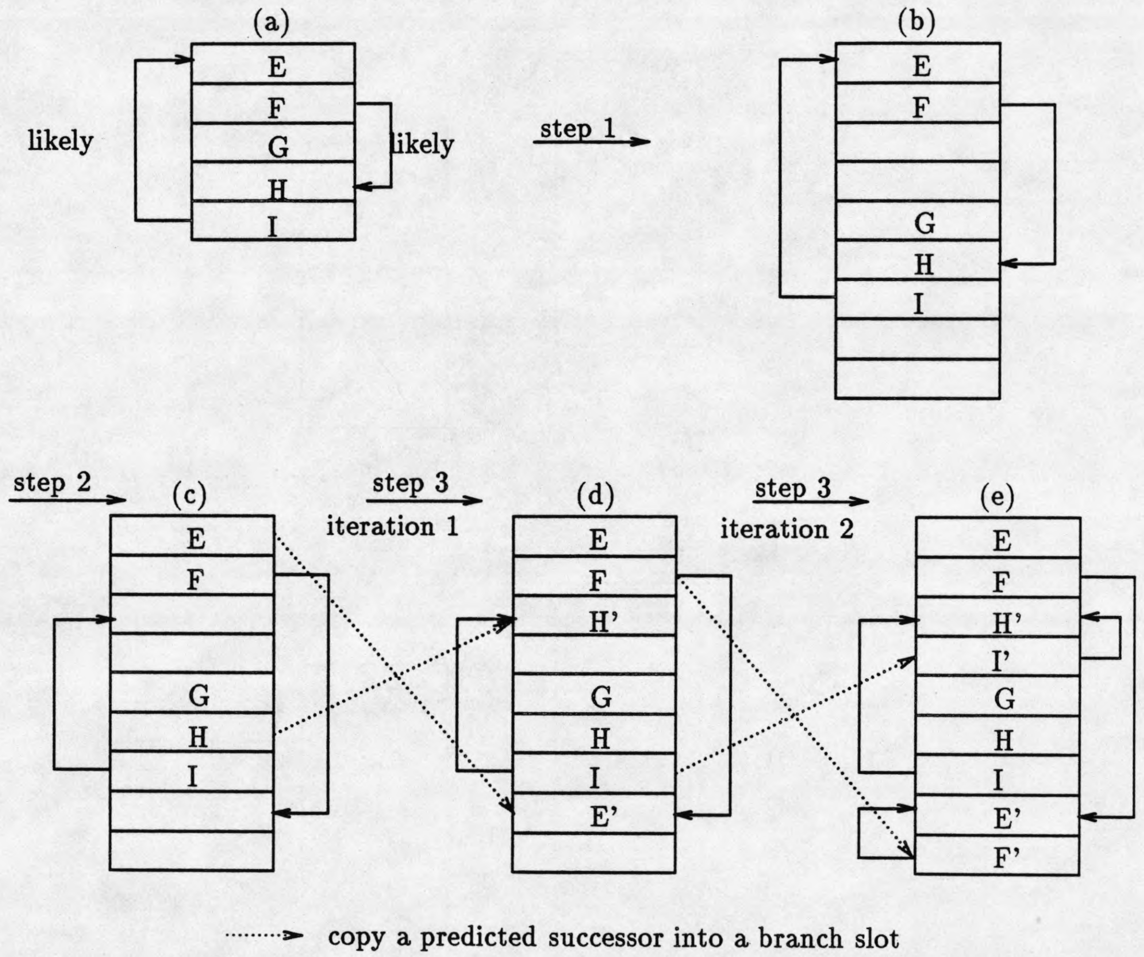

Figure 8: A running example of Inline Target Insertion.

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 I and I' are their individual addresses.

3.2 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 (e.g. instruction cache miss) in the sequencing pipeline, all the other instructions in the 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 $A_f(I(i,t))$.

The fetch address generation function of the sequencing pipeline is shown below. The sequencing pipeline fetches instructions sequentially by default. Each branch can 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.

```
Fetch Address Generation Function FAG(N)

    Pipeline stage 1:

    if $\{I(N+1,t) = EMPTY\}$ $A_f(I(1,t+1)) \leftarrow A_f(I(1,t)) + 1$
    else if $\{I(N+1,t) = REDIRECT\}$ $A_f(I(1,t+1)) \leftarrow$ adjusted target address of $I(N+1,t))$
    else $A_f(I(1,t+1)) \leftarrow A_f(I(1,t)) + 1$

    Other stages:

    $for\ k = 1...N$ $A_f(I(k+1,t+1)) \leftarrow A_f(I(k,t))$
```

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 the 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

(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
5	G			F

Figure 9: (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) A similar timing diagram for the sequence ... $E \rightarrow F \rightarrow G$ ...

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 to be fetched by the IF stage in cycle 5 is the next sequential instruction G.

A very important rule is that whenever the sequencing pipeline is empty, first instruction is always fetched from its original copy. The sequencing pipeline can be empty in three cases: program startup, incorrect branch prediction, and return from interrupt/exception. It is easy to guarantee that the program entrance address always be an original address. We will show in the next section that the appropriate original address for a program to resume after incorrect branch prediction and interrupt/exception handling is always conveniently available. These original addresses will be used by the sequencing to resume program execution.

$N+1$	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
$T(I,k)$	The k^{th} predicted successor of I
$S(I,k)$	The k^{th} correct successor of dynamic instruction I

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

3.3 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. We first show that branches are executed correctly with perfect branch prediction. We then finish the proof by showing that the execution of branches remains correct when they are incorrectly predicted.

Correctly Predicted Branches

The difficulties with proving the correctness of Inline Target Insertion are due to branches in insertion slots. 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 in no branch in insertion slots, the correctness follow from the description of the ITI Algorithm. All branch instructions would be originals and they would have their first N predicted successors in the next N sequential locations. Whereas a branch instruction is an insertion slot can not 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 $F \rightarrow H \rightarrow I...$?

Theorem 1 states that, without incorrectly predicted branches in the sequencing pipeline, the instructions in the sequencing pipeline are always the correct successors of the instruction at the end of the pipeline. Therefore, the sequence of instructions delivered by the sequencing pipeline is correct when all branches are predicted correctly.

Theorem 1 *If none of* $\{I(i,t), i = 1...N+1\}$ *is an incorrectly predicted branch, then* $I(i,t) = S(I(N+1,t), N-i+1), i = 1...N$.

Proof: The theorem can be proved by induction. Initially, the sequencing pipeline is empty. The first instruction I fetched into the pipeline must be an original. According to the code restructuring algorithm, the next N sequential instructions are the first N predicted successors to I. Since there is no instruction preceding I in the pipeline, the next N sequential instructions are fetched into the pipeline as I approach the end of the pipeline (see the Fetch Address Generation Function). Since there is no incorrect branch prediction, the first N predicted successors of I are also its first correct successors. This proves the initial step of the induction.

Assuming that the theorem holds up to cycle t, we show that it also holds for cycle $t+1$. *That is, knowing* $I(i,t) = S(I(N+1,t), N-i+1), i = 1...N$, *we need to show* $I(i,t+1) = S(I(N+1,t+1), N-i+1), i = 1...N$. *From the Fetch Address Generation Function,* $I(i+1,t+1) = I(i,t), i = 1...N$. *This implies* $I(i,t+1) = S(I(N+1,t+1), N-i+1), i = 2...N$. *It remains to be shown that* $I(1,t+1) = S(I(N+1,t+1), N)$.

If $I(N+1,t)$ *is not a taken branch, then it can not be a likely branch according to the assumption of correct branch prediction. Therefore,* $I(1,t)$ *can not be fetched from the last insertion slot of a branch.* $S(I(1,t),1) = S(I(2,t+1),1) = S(I(N+1,t+1),N)$ *must be* $I(1,t)$*'s next sequential* *instruction. According to the Fetch Address Generation function, $I(1, t+1)$ is simply the next sequential instruction of $I(1, t)$ if $I(N+1, t)$ is not a taken branch. Therefore $I(1, t+1) = S(I(N+1, t+1), N)$ is true if $I(N+1, t)$ is not a taken branch.*

If $I(N+1, t)$ is a taken branch, $I(1, t+1)$ would be the adjusted target of $I(N+1, t)$. This address of this adjusted target is N plus the original target address of $I(1, t+1)$. Note that an original target instruction is always an original instruction. The ITI algorithm ensures that the first N predicted successor of an original instruction are always found in the next N sequential locations. Therefore, the adjusted target of $I(N+1, t)$ is the N^{th} predicted successor of the original target of $I(N+1, t)$. Meanwhile, $I(N+1, t+1) = I(N, t)$ is a copy of the original target instruction of $I(N+1, t)$. Therefore, $I(1, t+1) = S(I(N, t), N) = S(I(N+1, t+1), N)$. QED.

Figure 10 illustrates Theorem 1 with the execution of instructions in Figure 8(e). Assume that correct instruction sequence should be equivalent to $E \rightarrow F \rightarrow H \rightarrow I \rightarrow E \rightarrow F$ in the original program in Figure 8(a). The pipeline starts by fetching E into the empty pipeline. Note that when F reaches the end of the pipeline in cycle 5, its correct successors H' and I' are already in the pipeline due to inline target insertion. F redirects the instruction fetch to E' which is the adjusted target of F. With correct branch prediction, the instructions at the IF and ID stages are always the correct successor of the one at the EX stage. Although instructions may be fetched from duplicate copies rather than their originals, the instructions delivered to the WB stage is equivalent to the correct sequence. The readers are encouraged to design an example involving the execution of F', a branch in a branch slot.

	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'

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

Incorrectly Predicted Branches

To execute an incorrectly predicted branch instruction correctly, the subsequent instructions in the sequencing pipeline must be scratched. The alternative target instruction address must be determined so that the instruction fetch can restart from that address. The results in this section show that the alternative target address for both likely and unlikely branches are conveniently available.

The case of unlikely branches is fairly straightforward. When the incorrectly predicted branch reaches the end of the sequencing pipeline, the alternative target address is easily derived from its fetch address (maintained by the hardware) and its target specification (e.g. target offset). Note that this address is always an original address (see the ITI Algorithm). Since the N predicted successors of an instruction always follow its original copy in memory, the pipeline correctly restarts fetching instruction from this address. Thus the alternative target address of an incorrectly predicted unlikely branch is conveniently available for restarting the instruction fetch.

The case of likely branches is not nearly as obvious. The general problem is illustrated in Figure 11. The alternative address of a likely branch I_1 is implicitly specified as N plus its original

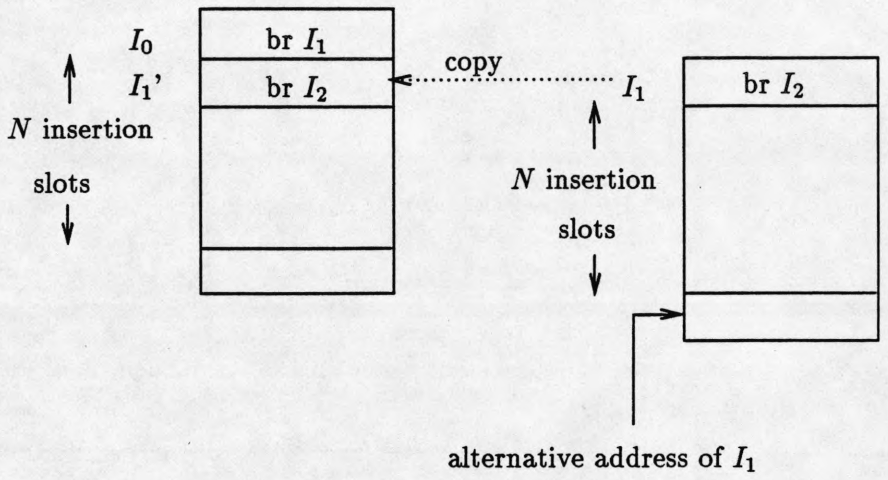

Figure 11: The problem of implicit alternative address for likely branches in insertion slots.

address. However, if the likely branch is copied into a branch slot, this implicit information is not copied with it. For example, if a dynamic instruction fetched from F' in Figure 8(e) is not taken, it must produce an instruction sequence equivalent to $F \rightarrow G \rightarrow H....$ To guarantee this, the address of G must be available when F' reaches the end of the sequencing pipeline. However, since F' does not carry any information about G being its alternative target instruction, it is not clear if the address of G will be available at that time. Fortunately, this is formally guaranteed by the Corollary to Theorem 2, whose proof is divided into Lemma 1 and Lemma 2.

Lemma 1 states that if a dynamic instruction $I(N+1,t)$ is fetched from its original copy, its original address is conveniently available in the form of $A_f(I(1,t)) - N$.

Lemma 1 *If the $I(N+1,t)$ is fetched from its original copy, then $A_f(I(1,t)) = N + A_o(I(N+1,t))$.*

Proof: Since $I(N+1,t) = I(1,t-N)$ and $I(N+1,t)$ is fetched from its original copy, $A_f(I(1,t-N)) = A_o(I(N+1,t))$. Because an original instruction can never reside in any branch slot, none of $\{I(N+1,t-N), I(N+1,t-N+1), ...I(N+1,t-1)\}$ can be likely branches. There could be unlikely branches among these last N instructions. However, unlikely branches do not *redirect instruction fetch unless they are incorrectly predicted. Any such incorrect prediction must be detected before $I(N+1,t)$ reaches the end of the pipeline. In this case, $I(N+1,t)$ would have been scratched from the pipeline before t. Therefore, none of the previous N instructions can be taken branches. The sequence pipeline fetches instructions sequentially between $t-n$ and t. This implies $A_f(I(1,t)) = N + A_f(I(1,t-N)) = N + A_o(I(N+1,t))$. QED.*

Lemma 2 states that if a dynamic instruction $I(N+1,t)$ is fetched from a duplicate copy, its original address is also conveniently available in the form of $A_f(I(1,t)) - N$. Note that each duplicate copy of a branch resides in a branch slot of an original likely branch. Thus we prove Lemma 2 by showing that for any arbitrary original likely branch B in the program, the Lemma holds for all the dynamic instructions fetched from its insertion slots. Since B is an arbitrary original branch instruction, this proves the Lemma for all dynamic instructions fetched from insertion slots.

Lemma 2 *If $I(N+1,t)$ is fetched from a duplicate copy, then $A_f(I(1,t)) = N + A_o(I(N+1,t))$.*

Proof: By induction. To prove the initial step of the induction, we prove that the Lemma is true for an dynamic instruction fetched from the first slot of an arbitrary original branch B. This slot contains a copy of $T(B,1)$ is the first target instruction of B. If $I(N+1,t) = T(B,1)$, then $I(N+1,t-1) = B$ and $I(N,t-1) = T(B,1)$. Since $T(B,1)$ is the first target instruction of B, the adjusted target address of B is $N + A_o(T(B,1))$. As a result, $A_f(I(1,t) =$ adjusted targe address of $B = N + A_o(T(B,1)) = N + A_o(I(N,t-1)) = N + A_o(I(N+1,t))$. This proves the initial step.

Assuming the Lemma holds for the $T(B,k)$, we show that the lemma also holds for $T(B,k+1)$. If $T(B,k)$ is a likely branch, the original of $T(B,k+1)$ must be its first target instruction. If $I(N+1,t) = T(B,k+1)$, then $I(N+1,t-1) = T(B,k)$ and $I(N,t-1) = T(B,k+1)$. Since $T(B,k+1)$ is the first target instruction of $T(B,k)$, adjusted target address of $T(B,k)$ is $N+A_o(T(B,k+1))$. As a *result, $A_f(I(1,t) =$ adjusted targe address of $T(B,k) = N + A_o(T(B,k+1)) = N + A_o(I(N,t-1)) = N + A_o(I(N+1,t))$. This proves the induction step for the case where $T(B,k)$ is a likely branch.*

If $T(B,k)$ is not a likely branch, the original of $T(B,k+1)$ follows immediately the original of $T(B,k)$. That is, $A_o(T(B,k+1))$ is equal to $1 + A_o(T(B,k))$. Also, $T(B,k)$ cannot be a taken branch. According to the induction assumption, if $I(N+1,t-1) = T(B,k)$, $A_f(I(1,t-1)) = N + A_o(T(B,k))$. Thus $A_f(I(1,t)) = 1 + A_f(I(1,t-1) = 1 + N + A_o(T(B,k)) = N + A_o(T(B,k+1)) = N + A_o(I(N+1,t)$. This proves the induction step for the case where $T(A,k)$ is not a likely branch. QED.

Theorem 2 *$A_f(I(1,t)) = N + A_o(I(N+1,t))$.*

Proof: Theorem 2 follows from the proofs of Lemma 1 and Lemma 2. QED.

Theorem 2 is perhaps the most critical result in proving the correctness of Inline Target Insertion. It assures that when an instruction reaches the end of the sequencing pipeline, N plus its original address is always available at no cost. There are two major applications for this result: recovery from incorrect branch prediction and return from interrupt/exception. The former is presented in this section and the latter will be the topic of the next section.

Corollary 1 states that the execution of an incorrectly predicted branch is very simple. When an incorrectly predicted likely branch reaches the end of the sequencing pipeline, it simply allows the pipeline to fetch the next sequential instruction (in addition to scratching all subsequent instructions in the sequencing pipeline).

Corollary 1 *The alternative address of a likely branch $I(N+1,t)$ is $1 + A_f(I(1,t))$.*

Proof: The $ITI(N)$ Algorithm opens N insertion slots after each likely branch. Therefore, the alternative target address ("fall through") of a likely branch I is always $1 + N + A_o(I)$. According the

	IF	ID	EX	WB
1	E			
2	F	E		
3	H'	F	E	
4	I'	H'	F	E
5	G			F
6	H	G		

Figure 12: Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow G \rightarrow H$ of instructions in Figure 8(e).

Theorem 2, when the branch is $I(N+1,t)$, then its alternative address is $1+N+A_o(I(N+1,t)) = 1+A_f(I(1,t))$. Note that is always the address of an original instruction (see Figure 7). The proof of Theorem 1 shows that the sequencing pipeline restarts correctly from the alternative address. QED.

Figure 12 shows the execution of instruction sequence $E \rightarrow F \rightarrow G \rightarrow H$ of Figure 8(e). When F reaches the EX stage, the hardware detects that it was incorrectly predicted. The two instructions in the sequencing pipeline (H' and I') will be scratched. The next sequential instruction of I' is G, which is exactly the alternative target instruction of F. This example is relatively simple because F was fetched from its original copy. The readers are encouraged to verify for themselves that the instruction sequence $I \rightarrow E' \rightarrow F' \rightarrow G \rightarrow H$ will be executed correctly. Note that F' is now fetched from a duplicate copy, which makes the situation slightly more complicated.

To summarize, we have shown the correctness of Inline Target Insertion in two steps. In the first step, we show that the branches are executed correctly if they are predicted correctly (Theorem 1). In the second step, we show that both likely and unlikely branches are executed correctly (second paragraph of this section and Corollary 1). It is also clear from the proofs that the hardware requirement for the execution is very small. The requirements are an array of fetch addresses of all instructions in the sequencing pipeline[10], an adder to derive the target address of a taken branch, and a mechanism to scratch instructions fetched due to an incorrectly predicted branch.

3.4 Interrupt/Exception Return

The problem of interrupt/exception return[Smith85][Hwu87] 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 can not 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

[10]It has also been shown that with a modification to the semantics of branch instructions, one can eliminate the array of fetch addresses as well.[Chan89a]

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. Theorem 3 shows that exception and interrupt return can be as simple as loading the empty[11] instruction sequencing pipeline with only one fetch address which is readily available upon detection of an interrupt/exception.

Theorem 3 *Interrupt/exception return to an instruction is correctly performed by loading the original address of the instruction to the fetch address of the first stage of an empty instruction sequencing pipeline.*

Proof: $A_o(I(N+1,t))$ is always available in the form of $A_f(1,t) - N$ (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 be available. Therefore, when an interrupt/exception occurs to an instruction, the processor can save the original address of that instruction as the return address. During exception return, the empty sequencing pipeline simply fetches instructions sequentially starting at the return address. Since the first instruction is an original instruction, all the first N predicted successors located in the next sequential locations. According to the proof of Theorem 1, the sequencing pipeline produces an instruction sequence equivalent to that without interrupt/exception. QED.

Figure 13 illustrates the equivalence between the sequence with and without exception to an instruction in a branch slot. Figure 13 shows the timing of a correct instruction sequence $E \rightarrow$

[11]The pipeline could still contain instructions from the interrupt/exception handler or from other processes. As far as the resuming process is concerned, the pipeline does not contain any instruction and/or sequencing information from the same process.

(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
5	E'	I'	H'	F
...				
1'	H			
2'	I	H		
3'	E'	I	H	

Figure 13: Timing diagram of a pipelined processor executing the sequence $E \rightarrow F \rightarrow H' \rightarrow I' \rightarrow E'$ of instructions in Figure 8(e).

$F \rightarrow H' \rightarrow I' \rightarrow E' \rightarrow F'$ from Figure 13 without exception. Figure 13 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[12]. 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.

An observation is 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 a superior alternative approach to reducing code expansion.

[12]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.

3.5 Extension to Out-of-order Execution

Inline Target Insertion can be extended to handle instruction sequencing for out-of-order execution machines [Toma67] [Weis84] [Acos86] [Hwu87] [Hwu88] [Smith89] . 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 followimg conditions may occur:

   (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 alternative 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 in this paper remain true 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[13] and CAD domains. The purpose is to show that Inline Target Insertion is an effective method for achieving high instruction sequencing efficiency for pipelined

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

processors. All the experiments are based on the an instruction set architecture which closely resembles MIPS R2000/3000[Kane87] with modifications to accommodate Inline Target Insertion. The IMPACT-I C Compiler, an optimizing C compiler developed for deeping pipelining and multiple instruction issue at the University of Illinois, is used to generate code for all the experiments [Chan88][Hwu89b][Chan89b][Chan89c].

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 average and sample deviation of all program and input combinations shown in Table 1. 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 have removed unnecessary unconditional branches via code restructuring [Chan88][Chan89b].

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 [Rude85]
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.

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[14] In Figure 8, target insertion for F and I increases the size of the loop from 5 to 11 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 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

[14]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 3.4).

benchmark	static *cond.*	static *uncond.*	dynamic *cond.*	dynamic *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[Hwu89a].

the static instructions in the programs. The numbers presented in Table 4 confirms that branches appear frequently in static programs. This supports the importance of being able to insert branches in the insertion slots (see Section 3.3). 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[Bako89].

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 to be 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 carefully 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 average 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.

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.

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 compile-time branch prediction can provide such a high prediction accuracy that the instruction sequencing efficiency remains high for large N values.

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 reaches the end of the sequencing pipeline where branches are executed. Therefore, the frequency of executing incorrectly predicted branches is not affected

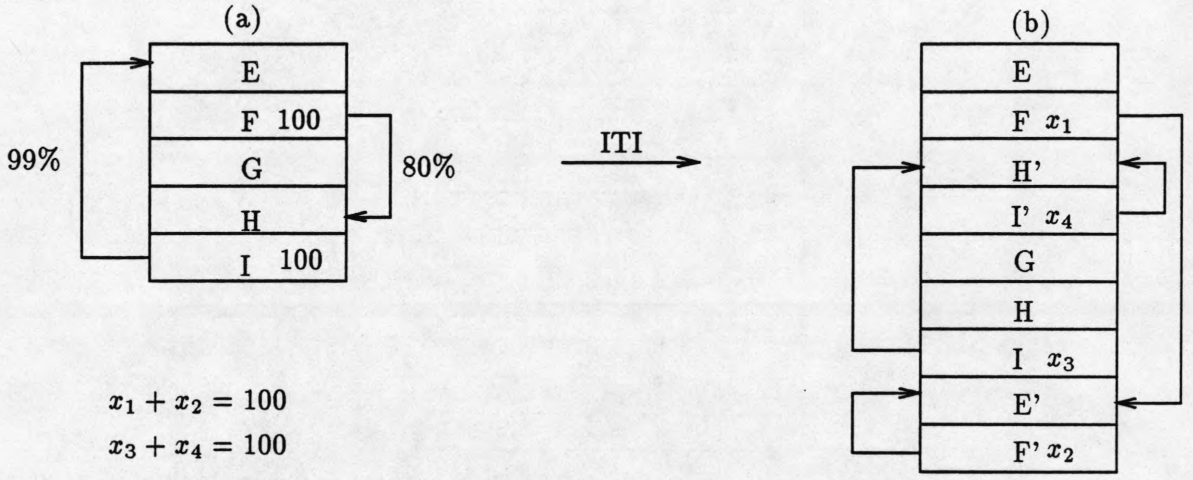

Figure 14: Evaluating the efficiency of instruction sequencing.

by Inline Target Insertion.

In Figure 14(a), the execution frequencies of E and F are both 100. E and F redirect the instruction fetch 99 and 80 times respectively. By marking E and F 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 bubbles 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 14(b) shows the code generated by INI(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 14(b) can be derived from the count of incorrect prediction in Figure 14(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 nonbranches. 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[15].

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 a the peak sequencing rate of a sequencing pipeline is one cycle per instruction ($K = 1$) and it requires 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 P values for a spectrum of threshholds averaged over all benchmarks. The $SD[P]$ column lists the sample deviations of P 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 instruc-

[15]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.

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.

tion 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 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. 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 MIPS-like 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 stability of code restructuring based on profile information is proved empirically using diverse inputs to each benchmark program.

The overall compile-time branch prediction accuracy is 92%. 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[Hwu89a] 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[Lee84]. 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. Both enable effective use of high bandwidth low cost instruction memories. The compiler complexity of Inline Target Insertion is simple enough not to be a major concern either. This has been proved 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, Nancy Warter, Sadun Anik, Thomas Conte, and all members of the IMPACT research group for their support, comments and suggestions. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, a donation from NCR, 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

[Acos86] 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.

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

[Bako89] Bakoglu et al, "IBM Second-Generation RISC Machine Organization," Proc. ICCD, pp.138-142, 1989.

[Birn86] J. S. Birnbaum and W. S. Worley, "Beyond RISC: High Precision Architecture", Spring COMPCON, 1986.

[Chan88] 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, San Diego, California, November, 1988.

[Chan89a] 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, Dublin, Ireland, August, 1989.

[Chan89b] P. P. Chang and W. W. Hwu, "Control Flow Optimization for Supercomputer Scalar Processing," Proceedings of the 1989 International Conference on Supercomputing, Crete, Greece, June 5-9, 1989.

[Chan89c] P. Chang, MS Thesis, "Aggressive Code Improving Techniques Based on Control Flow Analysis," Department of Electrical and Computer Engineering, Advisor W. W. Hwu, 1989.

[Chan90] P. P. Chang and W. W. Hwu, "Control Flow Optimization and Instruction Memory System Design Tradeoffs," Unpublished Report, draft available upon request, 1990.

[Chow87] P. Chow and M. Horowitz, "Architecture Tradeoffs in the Design of MIPS-X", Proceedings of the 14th Annual International Symposium on Computer Architecture, Pittsburgh, Pennsylvania, June, 1987.

[DeRo88] J. A. DeRosa and H. M. Levy, "An Evaluation of Branch Architectures", Proceedings of the 15th International Symposium on Computer Architecture, Honolulu, Hawaii, May, 1988.

[Ditz87] 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, Pittsburgh, Pennsylvania, June, 1987.

[Emer84] 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.

[Fost72] 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.

[Tjad70] 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.

[Gros82] T. R. Gross and J. L. Hennessy, "Optimizing Delayed Branches", Proceedings of the 15th Microprogramming Workshop, pp.114-120, October, 1982.

[Henn81] 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.

[Hill86] M. Hill and *etal*, "Design Decisions in SPUR", IEEE Computer, pp.8-22, November, 1986.

[Hors90] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple Instruction Issue in the NonStop Cyclone Processor," Proc. International Symposium on Computer Architecture, May 1990.

[Hsu86] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," Proceedings of the 13th International Symposium on Computer Architecture, pp. 386-395, Tokyo, Japan, June 1986.

[Hwu87] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines", IEEE Transactions on Computers, IEEE, December, 1987.

[Hwu88] 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.

[Hwu89a] 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, Jerusalem, Israel, May, 1989.

[Hwu89b] 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, Portland, Oregon, June 21-23, 1989.

[Inte89] Intel, "i860(TM) 64-bit Microprocessor," Order No. 240296-002, Santa Clara, California, April 1989.

[Joup89] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.272-282, April, 1989.

[Kane87] G. Kane, MIPS R2000 RISC Architecture, Prentice Hall, Englewood Cliffs, NJ, 1987.

[Kogg81] P. M. Kogge, *The Architecture of Pipelined Computers*, pp.237-243, McGraw-Hill, 1981.

[Kuck72] D. J. Kuck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup", IEEE Transactions on Computers, Vol. C-21, pp.1293-1310, December, 1972.

[Lee84] J.K.F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, January, 1984.

[McFa86] S. McFarling and J. L. Hennessy, "Reducing the Cost of Branches", The 13th International Symposium on Computer Architecture Conference Proceedings, pp.396-403, Tokyo, Japan, June, 1986.

[Mele89] Charles Melear, "The Design of the 88000 RISC Family", IEEE MICRO, pp.26-38, April, 1989.

[Nico88] J-D. Nicoud, "Video RAMs: Structure and Applications," IEEE MICRO, pp.8-27, February, 1988.

[Nico84] 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.

[Patt85] 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, Asilomar, CA, December, 1985.

[Patt82] D. A. Patterson and C. H. Sequin, "A VLSI RISC", IEEE Computer, pp.8-21, September, 1982.

[Ples87] 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.

[Ples88] A A. R. Pleszkun, G. S. Sohi, Multiple Instruction Issue and Single-chip Processors, Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture, San Diego, California, Nov. 1988.

[Radin 82] G. Radin, "The 801 Minicomputer", Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pp.39-47, March, 1982.

[Rude85] R. Rudell, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization", Proc. Cust. Int. Circ. Conf., May, 1985.

[Smith81] J. E. Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8th International Symposium on Computer Architecture, pp.135-148, June, 1981.

[Smith85] Smith, J. E. and Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", Proceedings of the 11th Annual Symposium on Computer Architectures, Boston, Massachusetts, June 17-19, 1985.

[Smith89] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instruction Issue", Third International Conference on Architectural Support for Programming Languages and Operating Systems, pp.290-302, April, 1989.

[Sun87] SUN Microsystems, "The SPARC(TM) Architecture Manual," SUN Microsystems, Part No. 800-1399-07, Revision 50, Mountain View, California, August 1987.

[Toma67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, vol.11, pp.25-33, January, 1967.

[Weis84] 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.