Branch Prediction of Conditional Nested Loops through an Address Queue

Zhigang Jin  
Department of Computer Science  
Midwestern State University  
Wichita Falls, TX 76308

Nelson L. Passos  
Department of Computer Science & Engineering  
University of Notre Dame  
Notre Dame, IN 46556

Virgil Andronache

Abstract -Multi-dimensional applications, such as image processing and medical imaging, require high computer performance. The critical sections of such applications consist of nested loops with the possibility of embedded conditional branch instructions. Branch prediction techniques usually require extra hardware, redundancy or do not guarantee the prediction accuracy. This paper shows a new architecture design, able to handle the conditional branches found in nested loops with minimum extra hardware and one hundred percent prediction accuracy. Detailed examples demonstrate the effectiveness of the method.

Keywords: loop transformation, branch prediction, architecture, retiming, instruction-level parallelism

1. Introduction

Multi-dimensional applications, such as fluid mechanics, and medical imaging, require high computer performance. The critical sections of such applications consist of nested loops with the possibility of embedded conditional branch instructions. Most of the previous results on optimizing loops are based solely on one-dimensional uniform problems, which do not consider the effects of conditional branches [5, 8]. This study focuses on the parallelism inherent to one-dimensional and multi-dimensional applications involving branch decisions ignored by the one-dimensional methods.

Some recent research has been conducted in the scheduling of multi-dimensional applications, such as the affine-by-statement technique [6], and the index shift method [10]. However, these methods are targeted to uniform loops. Modern superscalar microprocessors take advantage of instruction level parallelism (ILP) and move toward multiple instruction issues per cycle. These techniques require the use of effective branch prediction to exploit the full performance of these processors. Accurate hardware branch prediction is the key to many techniques for enhancing and exploiting instruction level parallelism. Pipeline stalls due to conditional branches represent one of the most significant impediments to realizing the performance potential of ILP. An effective branch predictor minimizes the number of pipeline stalls by predicting the direction of the branch and locating the next instruction before the true outcome of the branch is calculated.

In a previous study, the concept of multi-dimensional retiming was described [13]. That technique allows a nested loop body, represented by a general form of data flow graph (DFG), to be restructured while preserving data dependencies. The retiming technique produces overlapped iterations similar to those found in software pipeline solutions [8]. However, the overlapped iterations are not able to handle nested branch statements. The existence of branches or conditional statements in the code needs to be handled by different techniques [15].

Current branch prediction schemes can achieve average prediction accuracy in the range of 80-95%, depending on the type of prediction and the program being executed [4, 12]. Many branch predictors have been proposed to help alleviate the problem caused by conditional branches [1, 3]. Static branch prediction schemes use information gathered before program execution, such as execution profiles, to predict branch direction [2, 4, 14]. Dynamic branch prediction schemes use information gathered at run-time to predict branch direction. These prediction schemes use hardware to record branch history during the execution of a program and predict future branch directions by studying their previous behavior. Branch prediction accuracy of 80% to 90% has been reported for some simple branch history schemes [2, 4].

Current commercial systems use branch prediction and software pipelining techniques to avoid control hazard penalties. The branch prediction technique utilizes if-conversion methods to eliminate control dependencies and predicate registers to keep track of the validity of computed results when simultaneously executing the two possible branch paths. This technique allows an easy and safe implementation of software pipelining [11]. However, this solution requires the execution of instructions that will be discarded later, increasing the power consumption of the system.

As an example, a two-dimensional problem consisting of an edge detection algorithm based in the computation of a Laplace edge enhancement is shown on Figure 1 [9]. In a 5-stage pipelined processor (fetch, decode, execute, execute, write-back), one can easily conclude that the minimum execution time per iteration would be equivalent to 19 cycles, assuming a correct prediction of the branch, as seen on Figure 2(a). The execution time increases at least 4 cycles if a
misprediction occurs, shown in Figure 2(b). The new architecture proposed in this paper allows the possibility of achieving an execution time of 12 cycles.

![Figure 1. MDFG representing an edge detector filter](image)

2. Basic Principles

A data flow graph (DFG) is used to model the problems to be scheduled. A DFG $G$ consists of a tuple $(V,E,d,t)$, where $V$ is the set of computation nodes, $E$ represents the set of dependence edges, $d$ is a function representing the delays between two nodes, and $t$ is a function representing the computation time of each node, assumed to be one time unit in this paper. Iterations are identified by a multi-dimensional (MD) index equivalent to integer coordinates of a Cartesian space, known as iteration space. Inter-iteration dependencies are represented by vector-weighted edges.

An iteration is associated with a static schedule that is repeatedly executed for the loop. For any iteration $j$, an edge $e$ from $u$ to $v$ with delay vector $d(e)$ means that the computation of node $v$ at iteration $j$ depends on the execution of node $u$ at iteration $j - d(e)$. An edge with no delay, i.e., $d(e)=(0,0, ..., 0)$ represents a data dependence within the same iteration. The period during which all computation nodes in an iteration are executed, according to existing data dependencies is the iteration cycle of the static schedule for the corresponding DFG.

Retiming a Multi-Dimensional Data Flow Graph

A multi-dimensional retiming $r$ is a function from $V$ to $Z^*$ that redistributes the nodes in a dependence graph created by the replication of a MDFG $G$ [13]. A new MDFG $G_i$ is created, such that each iteration still has one execution of each node in $G$. The retiming vector $r(u)$ of a node $u \in G$ represents the offset between the original iteration containing $u$, and the one after retiming. The delay vectors change accordingly to preserve dependencies, i.e., $r(u)$ represents delay components pushed into the edges $u \rightarrow v$, and subtracted from the edges $w \rightarrow u$, $u \rightarrow v$, $w \in G$. After retiming, the execution of node $u$ at iteration $i$ is moved to the iteration $i - r(u)$. An example of retimed graph is shown in Figure 3.

![Figure 3. Retimed graph](image)

Branch Prediction

The behavior of typical branches is far from random. Most branches are strongly biased in a particular direction and are either usually taken or usually not taken [2]. History based prediction takes advantage of this bimodal distribution of branch behavior and attempts to distinguish “usually taken” from “usually not taken” branches. The history-based method is implemented using an array in which each branch in the executing program is mapped to an entry in the array. After the true outcome of the branch is determined the history bits for the branch are updated. The prediction accuracy of this technique depends on the number of history registers used, the number of bits in the registers, and the number of bits used in a second level [7].

![Figure 2. Execution (a) correctly predicted (b) mispredicted](image)

3. The New Architecture

The original concepts of retiming and software pipelining are applicable to uniform loops with no conditional statements [13]. The introduction of
conditionals in the loop prevents the regular use of these techniques, requiring the use of branch predication and simultaneous execution of the two possible paths depending on the outcomes of the branch. In order to avoid this extra cost, a new architecture must be able to handle the branches in such a way that mispredictions will not occur and speculative execution will not be required.

In order to understand the behavior of a retimed (software pipelined) loop, let us examine a simple example of a loop as represented by the DFG shown in Figure 4(a). In this example, considering that none of the instructions is a branch instruction, all the directed edges in the DFG represent data dependencies. For simplicity, we assume this DFG represents a special case of a one-dimensional problem where no loop-carried dependencies exist. Since all the edges have zero delays, after retiming, we can put at least one delay on each edge. Note that in this example the execution of instruction A does not depend on previous results of instructions E or F. Under such an assumption, the retiming technique allows the designer to push as many delays to each edge as necessary. The resulting DFG after retiming is shown in Figure 4(b). The following sequence of execution of those instructions would be an acceptable sequence for avoiding data dependencies:

$$A_0|A_1,B_0|A_2,B_1,C_0,D_0|A_3,B_2,C_1,D_1,E_0,F_0|A_4,B_3,C_2,D_2,E_1,F_1$$

In this sequence, the subscript indicates the original iteration for that instruction, with the underlined sequence being repeated according to the loop boundaries and the subscripts incremented by one at every repetition. The first three groups of sequences (A0), (A1, B0), (A2, B1, C0, D0), known as the prologue, are used to initialize the process.

Now, consider that node B is an if statement in the loop represented in Figures 4(a) and 4(b). In such a case, the decision made on B would command the path to be followed. So the edges B→C and B→D no longer represent data dependencies but control dependencies. Let us call the path CE the “taken” path and the path DF the “not-taken” path nt according to branch B. Now, let us assume that B jumps to t and nt alternately. Assuming a branch prediction strategy such that t will always be the selected path, and ignoring the possible consequences of mixing up the instructions of different paths, the sequence of instructions would be something like:

$$A_0 | A_1, B_0 | A_2, B_1, C_0 | A_3, B_2, C_1, D_1, E_0 | A_4, B_3, C_2, E_1$$

As can be seen, C1 was wrongly executed and after the decision implied by B1 was made, D1 was activated. However, in the next iteration, E1 could be wrongly executed instead of F1 because E follows C in the path t. Therefore, there are two problems with this sequence: the loss of performance due to the instruction that was wrongly executed and the logic error caused by the sequence of statements in the path t. These problems make software pipelining and retiming useless in loops that contain conditional branches unless a different technique is used in controlling the execution sequence.

![Figure 4. (a) A loop example represented by a DFG (b) The retimed loop](image)

In order to formalize this situation, consider a branch statement c controlling the execution of two alternate paths p and q within a loop, and X, the occurrence of event X at iteration i, two new pipeline hazards which occur in software pipelined code without branch predication are defined below.

**Definition 1.** A control anticipation hazard occurs when an operation in p or q is scheduled before c.

**Definition 2.** A sequence hazard occurs when two operations, X, and Y belonging to the same path p and different iteration instances (i and k) are executed in successive clock cycles due to the program counter advancement process and p, and q, were the actual paths that should be executed according to the branch statement.

The control anticipation hazards and the sequence hazards are two primary problems to be solved in order to apply software pipelining and retiming techniques on branch predictions. In order to avoid control anticipation hazards, the decisions taken at every instance of instruction S need to be stored along multiple iterations in a queue-like structure which should be used as input to the program counter register. The second group of hazards, sequence hazards, can also be handled by queuing the address of the next instruction to make sure that paths that have not been initiated will not have any instruction being executed. Since the most commonly used computer architectures today either do not support such execution sequences of multiple iterations or require the use of branch predication, a modified architecture is necessary. The following concepts propose this new architecture, which allows the program counter to be updated with the correct instruction address. Figure 5(a) shows an architectural diagram representing the commonly used method today for branch prediction based on branch history buffers. Branch predication is not considered in this study due to its requirement for duplicated resources. Figure 5(b) shows a slightly modified architecture, which is the one proposed in this
paper. In this design a new Address Queue has been added to the diagram showing a new input to the program counter (PC) updates.

In Figure 5(b), the current common architecture remains almost untouched except for adding the new Address Queue, which works very closely with the PC. When there is no branch in the loop execution sequence, the Address Queue is deactivated and the PC keeps updating in the regular fashion, i.e. incremented by one. However, when a branch is encountered within a loop, the Address Queue is enabled. The PC will update its content with instruction address loaded from the Address Queue, which contains correct sequence of instruction addresses by using software pipelining or retiming technique.

To understand how the queue is updated, refer to the previous example presented in Figure 4 and assume that node B represents a conditional statement. Figure 6(a) shows the expected (optimal) execution sequence before retiming and Figure 6(b) shows the execution sequence after retiming and placement of one delay on each edge.

As can be seen from Figure 6, after retiming, the execution sequence along the arrows has been changed in such a way that the execution of the next instruction does not have to wait for the completion of the previous instruction. Instead, the next instruction can be loaded into the pipeline right after the previous one has been fetched because there is no data or control dependency between any pair of consecutive instructions in the new execution sequence. Notice that when each instruction is described with a subscript, the subscript only represents the loop index. For example, the instruction $A_{k+4}$ and $A_{k+5}$ are actually the same instruction, which has only one instruction address in the memory.

Traditionally, the Program Counter is updated at the first stage of the pipeline for each instruction. In our design, however, the Program Counter is updated with the head element of the Address Queue when the system is executing the loop instructions within the loop and the Address Queue is updated at the last stage of the pipeline for each instruction, concurrently with the writing back of the result into memory.

<table>
<thead>
<tr>
<th>Loop Index</th>
<th>Actual Path</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>t</td>
<td>$A_k$</td>
<td>$B_k$</td>
<td>$C_k$</td>
<td>$E_k$</td>
</tr>
<tr>
<td>k+1</td>
<td>nt</td>
<td>$A_{k+1}$</td>
<td>$B_{k+1}$</td>
<td>$D_{k+1}$</td>
<td>$F_{k+1}$</td>
</tr>
<tr>
<td>k+2</td>
<td>nt</td>
<td>$A_{k+2}$</td>
<td>$B_{k+2}$</td>
<td>$D_{k+2}$</td>
<td>$F_{k+2}$</td>
</tr>
<tr>
<td>k+3</td>
<td>t</td>
<td>$A_{k+3}$</td>
<td>$B_{k+3}$</td>
<td>$C_{k+3}$</td>
<td>$E_{k+3}$</td>
</tr>
<tr>
<td>k+4</td>
<td>t</td>
<td>$A_{k+4}$</td>
<td>$B_{k+4}$</td>
<td>$C_{k+4}$</td>
<td>$E_{k+4}$</td>
</tr>
<tr>
<td>k+5</td>
<td>nt</td>
<td>$A_{k+5}$</td>
<td>$B_{k+5}$</td>
<td>$D_{k+5}$</td>
<td>$F_{k+5}$</td>
</tr>
<tr>
<td>k+6</td>
<td>t</td>
<td>$A_{k+6}$</td>
<td>$B_{k+6}$</td>
<td>$C_{k+6}$</td>
<td>$E_{k+6}$</td>
</tr>
</tbody>
</table>

Figure 6. (a) Execution Sequence Before Retiming (b) Execution Sequence After Retiming

To make sure that the system performs correctly using the Address Queue and the retiming and software pipelining techniques, some constraints must be applied to the new design.

Consider the DFG shown in Figure 7, which represents a loop embedded with a sole conditional branch. Assume that the DFG starts with a sequence of statements $S = \{ a^0, a^1, \ldots, a^{k-1} \}$ where $k$ is the number of vertices in $S$, followed by a conditional statement $c$ with two outgoing paths $P = \{ x_i^0, x_i^1, \ldots, x_i^{m_i} \}$ and $Q = \{ y^0, y^1, \ldots, y^{n_i} \}$ where $m$ is the number of vertices in $P$ and $n$ is the number of vertices in $Q$. There are two paths $ScP$ and
Queue needed to execute the loop properly is: N-stage pipeline, the minimum capacity of the Address Queue is defined as the number of vertices along the path. So in Figure 7 the length of ScP is (k+m+1) and the length of ScQ is (k+n+1). If m≤n then either path ScP or path ScQ can be treated as either the longest path or the shortest path. Notice there are possibly more than one path that may lead to the conditional statement c. However, since all instructions preceding the conditional statement will be executed, for simplicity, sequence S is assumed to represent the general path that leads to the conditional statement c and the number of vertices in S k represents the total number of vertices preceding the conditional statement c.

Next, a lemma is introduced and applied to a DFG mentioned above so that each edge in the DFG has at least one delay after retiming. By doing so all the nodes in the DFG can be executed in the pipeline in such a way that at each machine cycle a new instruction can be brought into the pipeline to be executed. So there will be no pipeline stalls between fetches of each instruction into the pipeline.

**LEMMA 3.** Given a conditional loop as represented by a DFG which starts with a statement sequence S = \{a^0, \ldots, a^i, \ldots, a^{k-1}\} followed by a conditional statement c with two outgoing paths P = \{x^0, \ldots, x^i, \ldots, x^{m-1}\} and Q = \{y^0, \ldots, y^j, \ldots, y^{n-1}\}. If the longest path has LL vertices, then the retiming function necessary to introduce one delay on each edge is:

\[
\begin{align*}
    r(a^i) &= LL - i - 1, \quad \text{for } 0 \leq i \leq k-1, \\
    r(c) &= LL - k - 1, \\
    r(x^j) &= LL - k - j - 2, \quad \text{for } 0 \leq j \leq m - 1 \\
    r(y^j) &= LL - k - j - 2, \quad \text{for } 0 \leq j \leq n - 1.
\end{align*}
\]

This lemma can be proved by computing the number of delays required in the longest path. The Address Queue is an on-chip modification, thus its size is a major factor in regarding to the cost of the chip. On the other hand, the larger the size of the Address Queue, the longer the sequential paths in a loop can be that the system can execute. So the size of the Address Queue is one of the design criteria in building a system architecture with an Address Queue. This is especially important in designing ASIC systems. The following lemma defines the relationship between the length of the path and the minimum size of the Address Queue.

**LEMMA 4.** Given a loop embedded with a conditional branch c represented by a DFG with a starting path S with k nodes, and two alternate paths with m and n nodes in a N-stage pipeline, the minimum capacity of the Address Queue needed to execute the loop properly is:

\[
    Q_{\text{min}} = (\min[m, n] + k + 1) \ast d - N + 2,
\]

where d is the retiming delay per edge and each edge in the DFG has equal retiming delays.

This lemma can be proved by calculating the difference between the number of delays in the shortest path and the number of stages needed to have the result of an instruction available as the instruction go through the pipeline. By retiming, the data dependence between any consecutive pair of instructions will be delayed by a factor of the retiming delay. In other words, assume that the retiming delay is d and the result of instruction A_k in current iteration j will not be needed as an input value of instruction B_k until iteration j+d. However, there is constraint that may affect this result when using the Address Queue. As an instruction A is loaded into the pipeline for execution, its result is not available until it is in the last stage (write back) in the pipeline. In order to update the Address Queue with instruction A_k's following instruction B_k, the result of instruction A_k should be available before instruction B_k is loaded into the Address Queue and pushed into the pipeline for execution. The following lemma gives a general constraint on this issue.

**LEMMA 5.** Given a loop embedded with a conditional branch c represented by a DFG having starting paths with total of k nodes, and two alternate paths with m and n nodes in a N-stage pipeline, the necessary condition on which the Address Queue is not empty at any machine cycle is:

\[
    (k + 1) + \min[m, n] \geq (N - 1) / d,
\]

where d is the retiming delay per edge and assuming all the edge in the DFG have equal retiming delays.

Lemma 5 can be proved by Lemma 4 and calculating the minimum number of instructions in the Address Queue. Notice that the retiming delay r in Lemma 5 is based on one-dimensional retiming technique. The positive retiming delays after retiming guarantee the legal retiming. For a MDFG case, the problem in finding a legal retiming is more complex than that of one-dimensional cases. However, for the multidimensional problem, positive delay vectors are too restrictive since a legal retiming of a MDFG may be still realizable even if it has negative delays. To find a legal multidimensional retiming for a MDFG, both incremental multidimensional retiming and chained multidimensional retiming techniques can be used to solve the problem [13]. When dealing with the multidimensional problem, a unique retiming delay may not be found for all the edges in the MDFG, thus the condition in Lemma 5 may not hold for MD problems.

4. **Algorithm**

Based on the new architecture with the Address Queue and application of software pipelining and retiming technique, an algorithm to implement the
execution of the conditional loop using the Address Queue as follows.

**Algorithm ReCLAQ (G = (V,E,d,t) )**
( Retimed Conditional Loop with Address Queue )
\[ \forall u \in V, LV(u) \leftarrow 0 \]
\[ L_{\text{max}} \leftarrow 0 \]
Queue \( V \leftarrow \emptyset \)
\[ \forall v \in E, E \leftarrow E - \{ e, \text{ s.t. } d(e) \neq (0,...,0) \} \]
Queue \( V \leftarrow \text{Queue}V \cup \{ u \in V, \text{ s.t. } \text{indegree} (u) = 0 \} \]
while \( \text{Queue} V \neq \emptyset \)
get(\( u \), Queue\( V \))
schedule(\( u \))
\[ v \text{ such that } u \rightarrow v \]
\[ \text{indegree} (v) \leftarrow \text{indegree} (v) - 1 \]
\[ LV(v) \leftarrow \max \{ LV(v), LV(u) + 1 \} \]
\[ L_{\text{max}} \leftarrow \max \{ LV(v), L_{\text{max}} \} \]
\[ MC(v) \leftarrow \max \{ MC(v), MC(u) \} \]
if \( \text{indegree} (v) = 0 \)
Queue \( V \leftarrow \text{Queue} V \cup \{ v \} \)
endif
endwhile
\[ \forall u \in V, r(u) = \left( L_{\text{max}} - LV(u) \right) \]
Get \( LL = \text{getLongPath}() \), \( SL = \text{getShortPath}() \); Given \( N \) = number of pipeline stages,
check if checkPath (\( N \), \( SL \)) = TRUE.
check if checkAddrQ (\( N \), \( SL \)) = TRUE.
End ReCLAQ

5. Conclusion

Most of the earlier scheduling methods do not explore loop pipelining across different dimensions. More recent techniques that accomplish that task are restricted to uniform loops without conditional statements. The use of if-conversion and branch prediction can be used to adapt the code to those techniques. This paper presents a new architecture using the Address Queue combined with software pipelining and retiming techniques to schedule the execution sequence of conditional loops for a single processor system with pipeline. The software pipelining and retiming techniques convert the execution sequence of instructions into a consecutive fashion without pipeline stalls and the Address Queue provides a mechanism that leads to a 100% accuracy in branch prediction.

6. Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. MIP 9704276.

References


