Branch Merging for Effective Exploitation of Instruction-Level Parallelism*

Chien-Ming Chen  Yunn-Yen Chen  Chung-Ta King
Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan 30043, R.O.C.

Abstract
In this paper, we propose a novel algorithm for merging branches in VLIW and superscalar architectures employing multi-way branch mechanisms. The branch merging problem is to find sets of branches that can be merged and executed concurrently. The advantages of using branch merging are: (1) to reduce the number of stalls due to branches and (2) to enlarge the size of basic blocks. We show that the general branch merging problem is NP-complete, and then propose a heuristic algorithm to solve this problem.

1 Introduction
With the advance of computer architecture, high performance processors incorporating multiple pipelines and functional units are now commonplace. Computer systems employing such processors are capable of issuing and executing instructions concurrently. Therefore, the amount of parallelism available in the instructions is a very important factor to the system performance. It follows that architectures, such as very-long-instruction-word (VLIW) and superscalar, rely heavily on optimizing compilers, which help to exploit instruction-level parallelism inherited in a program. However, the presence of inter-instruction dependencies often restricts an optimizing compiler's ability to exploit such parallelism. To increase the system performance, compiler and hardware must cooperate tightly to minimize the stalls caused by instruction dependencies.

The instruction dependencies of a program can be classified into control dependency, resource conflict, and data dependency. Control dependencies are caused by branches. Concurrent executions usually cannot be extended across branches because the execution path cannot be determined until the branch operation is evaluated. Data dependency and resource conflict are two major factors limiting the degree of parallelism inside a basic block. For general applications, the sizes of basic blocks are usually small, due to high frequency of branches [6]. Thus, the instruction-level parallelism exploitable within a basic block is also small, often limited to a factor of two to three. Such a degree of parallelism is not enough to fully utilize the computing resources available in a VLIW or superscalar processor. It is necessary to get more parallelism for processor performance [2, 4, 5].

Architecture supporting multi-way branches is an effective approach [3] to decreasing the number of branches in a program. The penalty due to branches is thus reduced. Also, it enlarges basic blocks to increase the instruction-level parallelism. Consider the following example:

```
if (a*b+c > 0) then
  if (d+e-f > 4) then
    c = d+f;
  else
    f = e+f;
else
  f = g *h;
  g = g*i;
  j = 1/j;
  if (c+d/e > 2) then
    c = d*e;
  else if (e+d-a > 1) then
    e = f*g;
  else if (a/c-b > 0) then
    h = i-j;
  else
    i = j-k;
else
  k = i+1;
  l = m*n;
```

The control-data flow graph (CDFG) of the program is shown in Fig. 1. We divide the Operations in an IF-construct into two classes, test evaluation and data. Circles in Fig. 1 represent test evaluation operations and rectangles represent data operations. The branch probabilities associated with a test evaluation node (T_i) are listed besides the corresponding edges. Assume that the number of cycles taken for add, subtract, multiply, divide, load, and store operations is 1, 1, 2, 2, 1, and 1, respectively. To execute a test evaluation node, say T_1, we have to load the two variables, a and b, then perform the multiplication and load c concurrently. Finally, addition and test are performed. The execution time of T_1 is thus equal to 1 + 1 + 2 + 1 = 5 cycles. The execution times for T_2, T_3, ..., T_5 can be obtained similarly. As a result, the average number of cycles spent

---

*This work was supported by the National Science Council, R.O.C., under contract no. NSC 82-0408-E-007-066 and NSC 81-0408-E-007-581.
in computing the five test evaluation operations is
\[5 \times 0.1 + 6 \times 0.9 + 4 \times 0.9 \times 0.8 + \cdots = 16.56.\]
Now, suppose we have a superscalar processor with two multiply/divide units and one multi-way branch unit. We can then merge branches together using selective code duplication and branch combining. As shown in Fig. 2, if the nodes \(T_1\) and \(T_3\) are combined into a single node \(T_{1,3}\), the average number of cycles for evaluating the tests becomes \(8 \times 0.1 + 4 \times 0.72 + \cdots = 14.16.\) Furthermore, the size of the basic blocks also increases.

In this paper, we first define formally the branch merging problem, and show that the problem is \(NP\)-complete. We then present a branch merging algorithm. Our algorithm defines a cost function which computes the weights of combining each pair of test evaluation nodes and reflects the average latency (scheduling) of the branch operations. We consider test evaluation operations and data operations in the same basic block simultaneously. By applying the technique, we effectively reduce the number of cycles for branch tests and the number of branch instructions. Also various considerations can be modeled using our algorithm by adjusting the cost function.

2 Preliminary

In this section, we first describe the architecture model at which our compiler technique is targeting, followed by a formal definition of the branch merging problem.

The model architecture consists of multiple functional units, a register file, and a multi-way branch unit (see Fig. 3.) Thus, multiple data and branch operations can be issued in a machine cycle. The set of instructions which can be executed at the same cycle and do not violate hardware constraints is called a \textit{long instruction word}, as shown in Fig. 3. For VLIW architectures, the \textit{instruction gather unit} in the figure can be simplified, because instructions are gathered by compilers. In superscalar architectures, the gather unit simulates the instruction issuing unit and packs instructions into long instruction words. The multi-way branch unit performs branch tests and changes the program execution flow. We say that an architecture is a \(2^k\)-way branch architecture if it can perform at most \(k\) branch tests and branch to at most \(2^k\) positions in the program.

We now give the formal definition of a \(2^k\)-way branch merging problem. Given a CDFG, we first label its test evaluation nodes with positive integers, 1, 2, \(\ldots\), \(n\). Let the symbol \(e_{i_1,i_2,\ldots,i_r}\) denote the gain of merging the test evaluation nodes \(i_1, i_2, \ldots, i_r\). together. We will discuss one way to calculating the gains in the next section. Let \(S_k\) be the power set of \\{1, 2, \ldots, \(n\)\}, in which the cardinality of each element is no larger than \(k\).

\begin{definition}
Given a CDFG with test evaluation nodes \(N = \{1, 2, \ldots, n\}\) and merge gains \(E = \{e_x\mid x \in S_k - \emptyset\}\), a general \(2^k\)-way branch merging problem is defined as:
\[
\text{minimize } \sum_{x=1}^{j} e_{x},
\]
\end{definition}
Theorem 1: The general $2^k$-way branch merging problem is NP-complete.

Proof: see [1] □

In summary, using our branch merging method, we first combine branches to enlarge the basic blocks. The code in the enlarged basic blocks is then scheduled and can be refined using previously proposed compiler techniques (such as code motion and code percolation). The complexity of code scheduling is thus reduced and the performance of the generated code can be improved.

3 The $2^k$-Way Branch Merging Algorithm

In this section, we present our heuristic algorithm to solve the $2^k$-way branch merging problem.

Algorithm Paradigm

The proposed algorithm is based on an iterative and bottom-up paradigm, as shown in Fig. 4. The CDFG of the program segment and the architecture specification (representing hardware constraint) are inputs to our algorithm.

Suppose that the target architecture supports $2^k$-way branches only. The original CDFG is then transformed into $\text{CDFG}_{2\text{way}}$, which can be executed on architectures allowing 2-way branches. We reconstruct the cost matrix for $\text{CDFG}_{2\text{way}}$ and feed $\text{CDFG}_{2\text{way}}$ back to our algorithm. This time $\text{CDFG}_{4\text{way}}$ is generated. By iteratively applying the proposed algorithm, a near optimal solution, $\text{CDFG}_{p\text{-way}}$, can be obtained.

The Algorithm

In our algorithm, a cost matrix $C$ must be constructed first. For a CDFG with $n$ test evaluation nodes, $C$ is an $n \times n$ matrix. Entries $c_{ij}$ and $c_{ji}$, where $i < j$, represent the penalty of merging the test evaluation operations and associated data operations, respectively, corresponding to nodes $T_i$ and $T_j$. In the merge step, a test node $T_j$ may be moved forward to merge with other nodes. A test evaluation node corresponding to a 2-way branch has at most $2^k$ data operation nodes. Let $O_{j,m}$, $0 \leq m \leq 2^k$, denote a data operation node following the test evaluation node $T_j$ immediately. Let $(O_{p(j,m)})$ denote the data operation node preceding the test evaluation node $T_j$ immediately. Suppose that $T_j$ and its parent test node are merged. After merging, data nodes $(O_{p(j,m)})$ and $(O_{j,m})$ are grouped into a new data operation node $O_{p(j,m)}$.

The first step of our algorithm is to perform the list scheduling algorithm on all the nodes in the CDFG in order to estimate the number of cycles to execute each one. While performing the list scheduling, we take into account the hardware constraints. From the results of the list scheduling, each entry in $C$ can be defined as follows:

$$
\begin{align*}
c_{ij} &= \begin{cases} 
NC(T_i) \cdot p(T_i) - NC(T_j) \cdot p(T_j) & \text{if } i < j \\
0 & \text{if } i = j \\
\sum_{m} (NC(O_{p(j,m)}) \cdot p(O_{j,m})) - NC(O_{j,m}) \cdot p(O_{j,m}) & \text{if } i > j
\end{cases}
\end{align*}
$$

Note that the cost defined by the above equation is the penalty to merge two test evaluation nodes. The gain of merging can thus be expressed as $-c_{ij}$. Note that not every pair of branches can be merged. Branch merging is basically code motion and thus is subjected to the restrictions for code motion. For example, data dependencies must be observed at all time and some side effects originated from procedure calls or exceptions should also be taken care of. Branch merging which might cause semantic errors will have their corresponding $c_{ij}$ set to infinite.

The proposed algorithm is shown below:

Algorithm Merging:

/* $C$ is the cost matrix and $c_{ij}$ is an entry of $C$. */
Array $B$ stores information of node merging.
If $B[i] = j$, then $B[j] = i$ and nodes $T_i$ and $T_j$ of the CDFG are merged together. The matrix $D$ is used to avoid repeat selections. */

Step 1. For each $i$, $B[i] = i$, $D[i] = false$;
Step 2. For each $i < j$, $c_{ij} = c_{ij} + c_{ij}$, $c_{ij} = c_{ij}$;
Step 3. Select $c_{ij}$ such that $D_{ij} = false$,
$c_{ij} > 0$, where $i < j$;
If no such $c_{ij}$ exists goto 6;
$D_{ij}$ = true;
Step 4. If ($B[i] = i$ or ($B[j] = j$) goto 3;
Step 5. Swap $B[i]$ and $B[j]$; goto 3;
Step 6. Repeat
select $c_{ij}$ such that $B[i] = i$, $B[j] = j$,
and $c_{ij} + c_{ij} < c_{ij} + c_{ij}$;
$temp = B[i]$;
swap($B[i]$, $B[j]$); swap($B[temp]$, $B[j]$);
until no $c_{ij}$ exist;

In the algorithm, Step 1 sets all the nodes $T_i$ in the CDFG to be unmarried (i.e. not merged with others.) We compute the total cost of test evaluation and data operation nodes in Step 2. In the following steps, we only consider the entries within the upper triangle of matrix $C$. In Steps 3 and 4 we repetitively merge a pair of nodes which will result in maximum positive gain and are both unmarried until there is no such pair. We then try to maximize the profit by exchanging the parties between two merged pairs until no improvement can be found.

4 Experimental Results

In this section, we demonstrate the performance of our algorithm by using three examples. The three test cases are Example, a program segment extracted from an error correction system, and the diao_phase_decoder in the High-Level Synthesis Benchmark 91. The numbers of test evaluation nodes of the error correction system and diao_phase_decoder are 13 and 14, respectively. Table 1 lists the performance results obtained from a SUN SPARC station II. For reference purpose, results from the code without branch merging are also included. The numbers in the parentheses indicate the percentages of performance improvement over the code without branch merging.

<table>
<thead>
<tr>
<th>Part</th>
<th>Original</th>
<th>Iter. 1 (4-way)</th>
<th>Iter. 2 (16-way)</th>
<th>Iter. 1 (4-way)</th>
<th>Iter. 2 (16-way)</th>
<th>Iter. 1 (4-way)</th>
<th>Iter. 2 (16-way)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test op.</td>
<td>16.56</td>
<td>11.8</td>
<td>8.4</td>
<td>4.979</td>
<td>3.724</td>
<td>2.09</td>
<td>4.925</td>
</tr>
<tr>
<td>(%)</td>
<td>(0%)</td>
<td>(29%)</td>
<td>(49%)</td>
<td>(0%)</td>
<td>(25%)</td>
<td>(58%)</td>
<td>(0%)</td>
</tr>
<tr>
<td>(%)</td>
<td>(0%)</td>
<td>(16%)</td>
<td>(40%)</td>
<td>(0%)</td>
<td>(7%)</td>
<td>(16%)</td>
<td>(0%)</td>
</tr>
</tbody>
</table>

Table 1: Performance summary of the test cases

As shown in the table, we get at least 7% improvement in the total execution time. In some cases the improvement even reaches 40%. For all the cases, our algorithm obtain the optimal solution. The running time of the algorithm is less than 0.1 second for all the cases. Thus, our algorithm is very efficient.

5 Conclusion

In this paper, we discussed the branch merging problem on VLIW and superscalar architectures, which support multi-way branches. Architectures employing multi-way branch units are performance-effective and are becoming popular. By merging branches, we have shown that it is possible to enlarge the basic blocks, reduce the number of branches, and fully utilize the multi-way branch hardware. Branch merging thus enhances the performance dramatically.

We have shown that the branch merging problem is NP-complete. A heuristic algorithm is proposed to solve the problem. The heuristic algorithm is time-efficient and produces high-quality solution.

References