Performance Evaluation of a Decoded Instruction Cache for Variable Instruction-Length Computers

Gideon Intrater
National Semiconductor (I.C.) Ltd.
P.O.Box 3007, Herzlia B. 46104, ISRAEL
E-mail gideon@nsc.nsc.com

Ilan Spillinger
Department of Electrical Engineering
TECHNION - Israel Institute of Technology
Haifa 32000, ISRAEL

ABSTRACT
A Decoded INstruction Cache (DINC) serves as a buffer between the instruction decoder and the other instruction-pipeline stages. In this paper we explain how techniques that reduce the branch penalty based on such a cache, can improve CPU performance. We analyze the impact of some of the design parameters of DINC's on variable instruction-length computers, e.g., CISC machines.

Our study indicates that tuning the mapping function of the instructions into the cache, can improve the performance substantially. This tuning must be based on the instruction length distribution for a specific architecture. In addition, the associativity degree has a greater effect on the DINC's performance, than on the performance of regular caches. We also discuss the difference between the performance of DINC's and other caches, when longer cache lines are used. The results presented were obtained by both analytical study and trace-driven simulations of several integer UNIX™ applications.

1. INTRODUCTION
Pipelined single-processor computers reach high performance by overlapping the execution of different instructions. A fundamental disadvantage of pipelining is the performance loss incurred, due to branches that require stalling or flushing of the pipeline. A combination of hardware, software and architectural techniques are used to reduce the branch penalty [1], [2]. Each technique contributes to the processor performance.

Branch folding [3] is a promising method to reduce the branch penalty. Branches are folded into other instructions, making it unnecessary to execute them as separate instructions. As in microprograms, an instruction issued for execution specifies both the operation code, and a conditional branch. At the end of the operation, the processor performs the (folded) branch if the condition is true. If the condition is false, the next sequential instruction is executed. Since branches are executed in parallel with other instructions, the branch penalty can be eliminated. In all the other branching techniques, branch execution requires at least one cycle, to issue the branch instruction in the pipeline.

To implement branch folding, two instructions must be examined in parallel. Folding can be implemented during compilation (as in the CCI Regulus [4]), but this increases the code size, and does effect existing code. A possible hardware solution is to merge the two instructions in the decoding stage of the pipeline. This is a demanding task for a single pipeline stage, especially on variable instruction-length computers, where the second instruction cannot be decoded before the length of the first one is known. The hardware required to implement such parallel decoding is complex and bulky.

A simpler method is to use a Decoded INstruction Cache (DINC) [3]. A DINC is a buffer between the instruction decoder, and the rest of the pipeline stages (Figure 1). Each entry in the DINC contains a decoded instruction, a branch condition, and a branch target address. Providing a branch target address for every instruction in the cache, has the same effect as turning every instruction into a potential branch instruction. This way, no specific branch instructions are required within the processor (except for indirect branches). Branches exist only in the machine language program.

Figure 1. Instruction execution path for a processor with a DINC

The folding algorithm can be performed before storing the instructions into the DINC. This eliminates the requirement to examine two instructions in the same cycle. Usually, folded instructions are fetched from the DINC. On a DINC miss, instructions must be folded while being fetched. As misses are infrequent, a single instruction decoder can be used with very little penalty.

Since most of the world's software runs on variable instruction-length machines (e.g., Intel-86, VAX and IBM-370), and users want to upgrade their system's performance without changing the existing code, A DINC seems to be a promising way to achieve this goal. In this work we study several design parameters and their effect on DINC performance in a CISC environment.
The outline of this paper is as follows. Section 2 describes the potential CPU performance. Problems in achieving a high cache hit rate in a DINC used in a CISC environment, are shown in Section 3. Our analytical model for DINC behavior with different mapping functions, is introduced in Section 4. Section 5 presents simulation results for various associativity schemes. In Section 6 we suggest a new method to improve DINC performance using large line sizes. Conclusions are summarized in Section 7.

2. BRANCH PENALTY

The execution time of a program can be expressed by: \( \text{Path Length} \times \text{CPI} \times \text{Cycle Time} \). \( \text{Path Length} \) is the dynamic size of the program (i.e., number of executed instructions), \( \text{CPI} \) is the average number of Cycles Per Instruction, and \( \text{Cycle Time} \) is the basic clock time unit of the processor. Techniques to minimize branch penalty increase the CPU performance by reducing the CPI and sometimes the Cycle Time. In this section we estimate the performance potential of different branching techniques. For our analyses we use a typical CISC instruction pipeline with 6 stages, like the one found in Zilog Z80000 [5] and Motorola 68040 [6] microprocessors.

![Instruction Pipeline Diagram](image)

**Figure 2.** A basic CISC instruction pipeline

The pipeline stages are:

- **IF**: Instruction Fetch - Fetches the next instruction from memory or the instruction cache.
- **ID**: Instruction Decode - Decodes the instruction, reads registers for address calculation (for instructions with an operand in memory) and calculates the branch target (for a branch instruction).
- **AG**: Address Generation - For a memory operand, calculates its address and sends it to the Data Cache.
- **MA**: Memory Access - Reads the operand from the Register File or from the Data Cache.
- **EX**: Execute - Executes the instruction. A No-Operation for LOAD and STORE instructions.
- **ST**: Store result - Writes result to the Register File, Data Cache, or to memory.

Execution of up to 6 different instructions can be overlapped, to reach a peak of one instruction completion per cycle (1 CPI). For sequential execution, the pipeline speed is limited by dependencies between instructions in the pipeline, and by storage delays (for a comprehensive discussion see [7]). For non-sequential execution (i.e., when branch instructions are introduced), the execution of the branch depends on the calculation of the branch target, and on the condition code. In our pipeline model, the condition codes are calculated in the EX stage, and the branch target is calculated in the ID stage.

To minimize the branch penalty, a branch-prediction mechanism is usually incorporated [8]. The processor predicts which direction the conditional branch will take, and starts to fetch instructions from that address. If the prediction was wrong, the processor needs to flush all the instructions that were prefetched from the wrong stream, and start to fetch from the correct address. The simplest branch prediction techniques are Always-Taken and Never-Taken. Under the Never-Taken strategy the IF continues fetching from the sequential stream. When the Always-Taken strategy is used, the IF fetches instructions beginning at the branch target. In Table 1 we summarize the four possible cases for the above branch predictions and their branch penalties. Note that the penalty is the sum of the time to execute the branch instruction (1 cycle), and the number of empty stages in the pipeline.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Never-Taken</th>
<th>Always-Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Not-Taken</td>
<td>1 + 0</td>
<td>1 + 3</td>
</tr>
<tr>
<td>Branch Taken</td>
<td>1 + 3</td>
<td>1 + 1</td>
</tr>
</tbody>
</table>

**Table 1.** Branch penalty using simple prediction techniques (in cycles)

Better branch prediction strategies can be employed [2] [8] to achieve about 85% correct predictions. The wrong prediction penalty can be reduced by using branch spreading. The compiler tries to re-order the compare and conditional branch instructions. If the compare instruction can be executed two or more instructions before the branch instruction, the penalty for wrong prediction is reduced to a single cycle (execution of the branch instruction itself). There are cases where the compare and branch instructions can not be spread. In [9], where a machine with the pipeline described in Figure 2 runs code generated for the Series 32000™ architecture [10], the authors show that branch spreading reduces the penalty for wrong branch prediction to 2.8 cycles. In [9], the authors also show that an average of 62.4% of the conditional branches are taken and 37.6% are not taken. Using the above, the average branch penalty is:

\[
1 + 1 \times 0.85 \times 0.624 + 2.8 \times 0.15 = 1.95\text{ cycles}.
\]

The three terms in the formula are: the cost of the branch instruction (1 cycle), the penalty associated with branches that were predicted taken and were actually taken, and the penalty associated with branches where the prediction was wrong.

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Never-Taken</th>
<th>Always-Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Not-Taken</td>
<td>1 + 0</td>
<td>1 + 3</td>
</tr>
<tr>
<td>Branch Taken</td>
<td>1 + 3</td>
<td>1 + 1</td>
</tr>
</tbody>
</table>

**Table 2.** Branch penalty while using a DINC

A DINC reduces the average number of cycles per branch as shown in Table 2. The IF stage fetches the next instruction according to the branch prediction mechanism, either from the next sequential address, or from the branch target address stored in the DINC. On a predicted-taken branch in a DINC, the IF stage does not need to wait a full cycle to let the ID stage calculate the branch target. Therefore, the branch penalty of taken branches that were predicted as such, becomes zero. Hence, the average branch penalty is: \( 1 + 2.8 \times 0.15 = 1.42 \) cycles.

Another reduction in the branch penalty can be achieved by implementing branch folding [3]. Since each instruction in the
DINC has a next address field, each instruction can easily incorporate a branch instruction by adding few more bits to specify the conditions for which the branch is actually taken. If all the branch instructions are folded into the previous instruction, the penalty for branches is reduced (see Table 3). The branch instruction is usually folded into the previous instruction. In case of a wrong prediction the four instructions in the next pipeline stages have to be flushed (one more instruction than in the non-folded case). Branch spreading reduces this penalty to 3.8 cycles [9]. The branch penalty here reduces to: 3.8x0.15 = 0.57 cycles.

### Table 3. Branch penalty while using folded branches in a DINC

<table>
<thead>
<tr>
<th>Prediction</th>
<th>Never-Taken</th>
<th>Always-Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Not-Taken</td>
<td>0</td>
<td>3.8</td>
</tr>
<tr>
<td>Branch Taken</td>
<td>3.8</td>
<td>0</td>
</tr>
</tbody>
</table>

Now that we know the branch penalty for each technique we can evaluate the performance of the processor. We have to add the number of CPI without the branch instructions, to the branch penalty in each pipeline type. The average CPI in the 6 stage pipeline is 1.22 CPI (assuming 1 cycle per branch) and the conditional branch frequency is 16.56% [9]. Thus the CPI is given by: 1.22 + 0.1656 x (branch penalty - 1). The results are presented in the following table (note that these results are without the penalty for instruction cache misses). The term RINC in Table 4 stands for Regular INstruction Cache.

### Table 4. CPU Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>CPI</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>RINC, Branch Prediction and Branch Spreading</td>
<td>1.38</td>
<td>100%</td>
</tr>
<tr>
<td>DINC, Branch Prediction and Branch Spreading</td>
<td>1.29</td>
<td>107%</td>
</tr>
<tr>
<td>DINC, Branch Prediction, Branch Spreading and Branch Folding</td>
<td>1.15</td>
<td>120%</td>
</tr>
</tbody>
</table>

### Table 5. Direct mapping into a 1-way set associative DINC

Empty sets might increase the miss rate as illustrated in the following example.

### Table 6. Mapping two procedures into a 1-way DINC

Example 2: Assuming the same DINC of Example 1, we map two procedures, A and B, into the DINC. Procedure A consists of four 2-byte instructions $I_1$, $I_2$, $I_3$ and $I_4$ in addresses 00000, 00010, 00100 and 00110, respectively. Procedure B consists of instructions $I_{a1}$, $I_{b2}$, $I_{b3}$ and $I_{b4}$, lengths 1, 3, 1 and 3 bytes, and addresses 10001, 10010, 10101 and 10110 respectively.
Clearly, after both procedures have been executed once (A and then B), all the instructions except for \( I_{A2} \) and \( I_{A4} \) are in the cache (Table 6). The instructions of the two procedures are mixed in the DINC. In the case where procedures A and B execute alternately, a miss occurs on each execution of instructions \( I_{A2}, I_{A3}, I_{A4}, I_{A6} \).

To reduce the number of empty sets that might cause misses, the index can be extracted from higher bits of the instruction address. For the above example, the index can be extracted from bits 1 to \( n+1 \) i.e., shifted by one bit (1-shift), as shown in Figure 4. Table 7 shows that this mapping results in a better cache utilization.

### Table 7. Mapping two procedures into a 1-way DINC (with 1-shift)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
<th>Set</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>( I_{A1} )</td>
<td>10001</td>
<td>( I_{B1} )</td>
<td>000</td>
<td>( I_{A1}, I_{B1} )</td>
</tr>
<tr>
<td>0010</td>
<td>( I_{A2} )</td>
<td>10010</td>
<td>( I_{B2} )</td>
<td>001</td>
<td>( I_{A2}, I_{B2} )</td>
</tr>
<tr>
<td>0110</td>
<td>( I_{A3} )</td>
<td>10101</td>
<td>( I_{B3} )</td>
<td>010</td>
<td>( I_{A3}, I_{B3} )</td>
</tr>
<tr>
<td>00110</td>
<td>( I_{A4} )</td>
<td>11010</td>
<td>( I_{B4} )</td>
<td>011</td>
<td>( I_{A4}, I_{B4} )</td>
</tr>
</tbody>
</table>

### Table 8. Mapping two procedures into a 2-way DINC with a 1-shift

<table>
<thead>
<tr>
<th>Procedure A</th>
<th>Procedure B</th>
<th>DINC</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_{A1} )</td>
<td>( I_{B3} )</td>
<td>( I_{B3} )</td>
</tr>
<tr>
<td>( I_{A2} )</td>
<td>( I_{B4} )</td>
<td>( I_{B4} )</td>
</tr>
<tr>
<td>( I_{A3} )</td>
<td>( I_{B5} )</td>
<td>( I_{B5} )</td>
</tr>
</tbody>
</table>

Figure 4. Tag and index address fields with 1-shift

We see that empty sets cause scattered misses because of intermixed procedures. To solve it, we use a non-zero shift that might cause collisions and full sets. Both effects increase the miss ratio. In the next section we introduce a novel model to estimate the optimal mapping function to minimize the miss ratio, without using extensive trace driven simulations.

### 4. EVALUATION OF DINC BEHAVIOR

Intuitively, a DINC performs optimally if the number of collisions, full sets and empty sets are minimal. To estimate these numbers we developed the following probabilistic model.

First, let us define a memory line as a group of \( L \) consecutive addresses (bytes) mapped to the same DINC set. Also, \( P(\text{Len}=i) \) is the probability that the length of a given instruction is \( i \). A collision in a 1-way set associative DINC occurs whenever two or more instructions start in the same memory line. This happens whenever an instruction \( I \) with length \( i \) starts in a memory line ML, with less than \( L-i \) occupied bytes. The instructions preceding \( I \) occupy less than \( L-i \) bytes of ML. Instruction \( I \) uses \( i \) bytes of ML, thus the instruction that follows \( I \) will also start in ML and be mapped into the same set. Assuming that on variable instruction-length machines there is an equal probability that an instruction ends at any of the \( L \) bytes in a memory line, the collision probability in a 1-way set associative DINC, \( P_c(1) \), is given by:

\[
P_c(1) = \frac{1}{L} \sum_{i=1}^{L} (L-i) P(\text{Len}=i)
\]

(1)

In a similar way, the collision probability of an \( N \)-way set associative DINC, \( P_c(N) \), is given by:
The following table shows the collision probability evaluated by equations (1) and (2) using the measured instruction length distribution of our workload, for several cache associativity schemes and different mapping functions. Note that we used only simple mapping functions employing shifts of the index. Other mapping functions can also be used but may require the use of complex hardware. The results in Table 10 show that with higher associativity, more shifts (and therefore less empty sets) can be used without getting any collisions.

TABLE 9. The UNIX workload

<table>
<thead>
<tr>
<th>Name</th>
<th>Instructions in Trace</th>
<th>Static Code Size</th>
<th>Average Instruction Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls</td>
<td>22551</td>
<td>19320 Bytes</td>
<td>3.26 Bytes</td>
</tr>
<tr>
<td>dc</td>
<td>531789</td>
<td>33780 Bytes</td>
<td>3.32 Bytes</td>
</tr>
<tr>
<td>grep</td>
<td>697463</td>
<td>16944 Bytes</td>
<td>3.20 Bytes</td>
</tr>
<tr>
<td>opt</td>
<td>430323</td>
<td>31276 Bytes</td>
<td>3.70 Bytes</td>
</tr>
</tbody>
</table>

TABLE 10. Collision probability as a function of the mapping and associativity

The probability of full sets in a DINC can be calculated using the equation for collision probability. A memory line causes a fully occupied set in an associativity of N, if in an N-1 associativity it causes a collision, and it does not cause a collision in an N-way associativity. E.g., two instructions that reside in the same set in a 2-way DINC cause a collision in a 1-way DINC, since only one instruction can fit in the single instruction slot of the set. Thus, the probability of full sets in an N-way DINC, \(P_f(N)\), is:

\[
P_f(N) = P_c(N-1) - P_r(N)
\]

In the following table we show the probability of full sets evaluated using equation (3). Note that in this table we start with associativity of 2-way. In 1-way associativity, the behavior of DINCs and RINCs is similar, each time an instruction is mapped into a specific set, this set is fully occupied, and cannot be used for other instructions without removing the previous one.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>0-shift</th>
<th>1-shift</th>
<th>2-shift</th>
<th>3-shift</th>
<th>4-shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way</td>
<td>0</td>
<td>0</td>
<td>0.273</td>
<td>0.580</td>
<td>0.790</td>
</tr>
<tr>
<td>2-way</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.214</td>
<td>0.579</td>
</tr>
<tr>
<td>4-way</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.191</td>
</tr>
<tr>
<td>8-way</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16-way</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

TABLE 11. Probability of full sets as a function of the mapping and associativity

An instruction causes empty sets whenever its length \(i\) is greater than the memory line size \(L\), and crosses a memory line boundary. In this case the following instruction is not mapped to the consecutive cache set. Our model for number of empty sets, \(E\), is:

\[
E = \sum \left( \frac{1}{L} - 1 \right) P(Len=i)
\]

Where \(MAX\) is the maximal instruction length. It is easy to see that the average number of empty sets does not depend on the associativity of the DINC. In Table 12 we use equation (4) to evaluate the average number of empty sets.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>0-shift</th>
<th>1-shift</th>
<th>2-shift</th>
<th>3-shift</th>
<th>4-shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty Sets</td>
<td>2.369</td>
<td>0.685</td>
<td>0.116</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
organizations and mapping functions. The DINC capacity was modeled with 128, 256 and 512 instructions, and simulated with 1-, 2-, 4-, 8- and 16-way set associative organization. Note that to simulate a multitasking environment, our simulator flushed the whole instruction cache every 100,000 instructions. Figure 5 shows the geometric average [12] of the miss rate over all four applications and DINC capacities.

These results match the conclusions that followed our analysis presented above. In the next section we focus on the influence of different associativity degrees on the DINC performance.

5. ASSOCIATIVITY IN DINC

In the previous section we determined the best mapping function for each associativity degree of a DINC. Usually, to choose the best associativity scheme, the designer has to consider the chip-area and the complexity of different associativity degrees [13]. We simulated DINCs and RINCs of comparable capacities to evaluate the improvement in miss rate when the associativity degree increases. The DINC was modeled with 128, 256 and 512 instructions. The RINC was modeled with 512, 1024 and 2048 bytes (average instruction length is close to 4 bytes, in our workload, so we compared a DINC of m entries with a RINC of 4 x m bytes). Both the DINC and the RINC were simulated with 1-, 2-, 4-, 8- and 16-way set associative organizations. Figure 6 presents the geometric mean of the miss rate over all the applications and cache capacities.

As expected, the miss rate in RINC decreases as the associativity degree increases [6]. The main factor is the higher flexibility in choosing the place for a new instruction. As follows from Section 4 above, associativity has greater effect on DINCs (see the slope of the DINC’s curve).

From Figure 6, it follows that for high associativity degrees, the performance of the DINC and the RINC is comparable. This is due to the negligible number of collisions, full sets and empty sets in DINCs with high associativity degrees. The fact that a 8-way is the lowest associativity degree for DINC performance comparable to a RINC, matches the results presented in Section 4, where it is the lowest associativity degree to overcome the collisions, full sets and the empty sets problems.

It should be pointed out that although the instruction capacities of the DINC and the RINC we compared, are similar, the actual area to implement a DINC is larger. In the DINC we have modeled, we decode each instruction into a vertical microcode format with a fixed length of four bytes each (similar to the instruction encoding in RISC machines [7]). This format can be used only for a subset of the machine instructions (which will be executed at the full throughput of the pipeline). In our workload, the length of more than 60% of the instructions is less than or equal to 4 bytes, most of them are 2 or 3 bytes long. By changing the encoding of all the short instructions and some of the long instructions to 4 bytes, we can achieve a high percentage of instructions decoded in 4 bytes. We assume that the number of instructions that cannot be mapped to 4 bytes is negligible, and therefore choose not to map them into the DINC. It can be shown that two more bytes can contain all the rest of the information needed in a DINC representation (the branch condition, and the branch target address). Thus the number of bytes needed for storing an instruction in a DINC is six, four bytes for the instruction, and another two for the folded branch. In the next section, we evaluate the performance of DINCs with various line sizes.

6. LINE SIZE EFFECT ON DINCS

In the DINC structures we discussed above, we assumed that the DINC incorporates single instruction lines. A common method to increase the performance of regular caches is to use a larger line size [14]. In the following we show that it is not trivial to obtain the same advantage in a DINC.

Each cache line stores several instructions and a single tag, see Figure 7 below (this is more efficient for a given cache area, when both the data and the tag area are considered [13]). Such a structure implies that only a stream of instructions that are executed sequentially can be mapped into a single line, where branches cause access to a new cache line. This limits the number of instructions in each cache line to the basic block size (i.e., the number of instructions between two consecutive branches).

As expected, the miss rate in RINC decreases as the associativity degree increases [6]. The main factor is the higher flexibility in choosing the place for a new instruction. As follows from Section 4 above, associativity has greater effect on DINCs (see the slope of the DINC's curve).

From Figure 6, it follows that for high associativity degrees, the performance of the DINC and the RINC is comparable. This is due to the negligible number of collisions, full sets and empty sets in DINCs with high associativity degrees. The fact that a 8-way is the lowest associativity degree for DINC performance comparable to a RINC, matches the results presented in Section 4, where it is the lowest associativity degree to overcome the collisions, full sets and the empty sets problems.

It should be pointed out that although the instruction capacities of the DINC and the RINC we compared, are similar, the actual area to implement a DINC is larger. In the DINC we have modeled, we decode each instruction into a vertical microcode format with a fixed length of four bytes each (similar to the instruction encoding in RISC machines [7]). This format can be used only for a subset of the machine instructions (which will be executed at the full throughput of the pipeline). In our workload, the length of more than 60% of the instructions is less than or equal to 4 bytes, most of them are 2 or 3 bytes long. By changing the encoding of all the short instructions and some of the long instructions to 4 bytes, we can achieve a high percentage of instructions decoded in 4 bytes. We assume that the number of instructions that cannot be mapped to 4 bytes is negligible, and therefore choose not to map them into the DINC. It can be shown that two more bytes can contain all the rest of the information needed in a DINC representation (the branch condition, and the branch target address). Thus the number of bytes needed for storing an instruction in a DINC is six, four bytes for the instruction, and another two for the folded branch. In the next section, we evaluate the performance of DINCs with various line sizes.

6. LINE SIZE EFFECT ON DINCS

In the DINC structures we discussed above, we assumed that the DINC incorporates single instruction lines. A common method to increase the performance of regular caches is to use a larger line size [14]. In the following we show that it is not trivial to obtain the same advantage in a DINC.

Each cache line stores several instructions and a single tag, see Figure 7 below (this is more efficient for a given cache area, when both the data and the tag area are considered [13]). Such a structure implies that only a stream of instructions that are executed sequentially can be mapped into a single line, where branches cause access to a new cache line. This limits the number of instructions in each cache line to the basic block size (i.e., the number of instructions between two consecutive branches).

As expected, the miss rate in RINC decreases as the associativity degree increases [6]. The main factor is the higher flexibility in choosing the place for a new instruction. As follows from Section 4 above, associativity has greater effect on DINCs (see the slope of the DINC's curve).

From Figure 6, it follows that for high associativity degrees, the performance of the DINC and the RINC is comparable. This is due to the negligible number of collisions, full sets and empty sets in DINCs with high associativity degrees. The fact that a 8-way is the lowest associativity degree for DINC performance comparable to a RINC, matches the results presented in Section 4, where it is the lowest associativity degree to overcome the collisions, full sets and the empty sets problems.
This causes block size is tions in a memory line, due to the variable instruction length. branch targets are not mapped to the beginning of A tics LDINC with comparable capacity. In other words, we see that the LDINC does not exploit to full advantage the longer lines showing that the RINC achieves a much better hit rate than an LDINC with comparable capacity. In other words, we see that the LDINC does not exploit to full advantage the longer lines as does a RINC (refer to [14]). There are three LDINC characteristics that cause this performance degradation. The first one is that branch targets are not mapped to the beginning of memory lines. A second problem is the irregularity of the length of n instructions in a memory line, due to the variable instruction length. This causes full sets, empty sets and more collisions (depending on the mapping function). The third problem is that the basic block size is small (4-5 instruction on average), thus long cache lines cannot be fully utilized. A possible way to tackle the first two problems is to use the compiler to tune the code, so that instructions fit the memory lines, using the following alignment algorithm.

The goal of the algorithm is to group together at most n instructions (n will be the number of instruction slots in a cache line) with a length of exactly m bytes (i.e., memory line with \( L = m \)). If the length of the set of instructions grouped together is less than \( m \), we "pad" this group with a branch instruction to the aligned address of the next memory line (this address is of course a multiple of \( m \)). Also, we do not allow grouping together a set of instructions whose length exceeds the limit of \( m \) bytes, although the number of instructions in the group is less than \( n \). Regardless of the process described above, if we encounter an instruction that is a possible target for a branch instruction, we assign this instruction to an aligned address (i.e., as a first instruction in a new group). Notice that the branch instructions that we use for the padding adjust each group size to \( m \). These instructions will not map into the LDINC due to the use of the branch folding technique [3].

This algorithm applies only for software that can be re-compiled, and benefit from the better cache performance. Existing code can still run on the LDINC correctly. To quickly evaluate the potential for the above compiler technique, we modified our workload traces (by changing instruction addresses) to produce a trace with aligned instructions. Note that a compiler tuned for the DINC structure can produce a more efficient code.

In Figure 8 we summarize the results for RINC and the two kinds of LDINC: LDINC without code alignment, and LDINC using code alignment (ALDINC). The line size \( m \) is 4x\( n \) bytes, and the mapping function uses \( \log_2 (m) \) shifts.

![Diagram](image)

**Figure 8.** Average percentage of misses in an LDINC

![Diagram](image)

**Figure 9.** Average percentage of misses in RINC and LDINC

**Figure 8** shows that the minimal number of misses is achieved with a higher number of shifts in longer cache lines. **Figure 9** shows that a RINC achieves a much better hit rate than an LDINC with comparable capacity. In other words, we see that the LDINC does not exploit to full advantage the longer lines as does a RINC (refer to [14]). There are three LDINC characteristics that cause this performance degradation. The first one is that branch targets are not mapped to the beginning of memory lines. A second problem is the irregularity of the length of \( n \) instructions in a memory line, due to the variable instruction length. This causes full sets, empty sets and more collisions (depending on the mapping function). The third problem is that the basic block size is small (4-5 instruction on average), thus long cache lines cannot be fully utilized. A possible way to tackle the first
7. SUMMARY

We have described how branch folding and spreading can be used together to greatly improve performance. These techniques can enhance the performance of existing programs without compiling them again for a new architecture. The demand for such improvement is very great because of the high cost of recompiling the billions of bytes of code running on the popular architectures, e.g., Intel-86, VAX and IBM-370.

An efficient way to implement branch folding is to use a DINC. We wanted to simplify the implementation of a DINC for variable instruction length computers, and to design a DINC that would take advantage of the performance improvement potential of the branch folding technique.

We have explained the correlation between instruction length distribution, and mapping into a DINC. This correlation enables the DINC designer to choose the best function for mapping into a DINC, using a minimum number of simulation runs. We have also shown that, even though designers should usually choose direct mapped caches [15], high associativity degrees are necessary to enable hit rates that compare favorably with those of RINCS.

Longer cache lines can also enhance the performance of a DINC, because less memory is required for tags. However, they do not enhance performance as much as might be expected. We described three factors that prevent high hit rates in an LDINC. We proposed a way to eliminate two of these three factors, by having the compiler tune the code to the cache. Code that was recompiled achieved better hit rates. However, the basic block size continued to restrict the performance of the LDINC.

Finally, we succeeded in designing a DINC whose hit rate is comparable to that of an RINC. We were able to actualize the 20% improvement in processor performance that is possible using DINCs. We suggest that further work be done with DINCs on superscalar CISC machines, to determine how DINCs can simplify the logic used to issue instructions.

8. ACKNOWLEDGMENTS

We would like to thank Raya Leviatan, Sara Bitan, and Eran Gluska from National Semiconductor for their help with the instruction tracer that provided the Series 32000 traces. Special thanks to Don Alpert from Intel and Gabriel M. Silberman from IBM for their help during the various stages of this work.

9. REFERENCES


1 Series 32000 is a trademarks of National Semiconductor Corporation.
2 UNIX is a trademark of AT&T.