

# Basic-block Reordering Using Neural Networks

Xianhua Liu, Jiyu Zhang, Kun Liang, Yang Yang and Xu Cheng

Microprocessor Research and Development Center,  
Peking University, Beijing, P.R.China, 100871  
{liuxianhua, zhangjiyu, liangkun, yangyang, chengxu}@mprc.pku.edu.cn

**Abstract.** Basic-block reordering is a compiler optimization technique which has the effect of reducing branch cost and I-Cache misses by rearranging code layout. In this paper, we present our basic-block reordering method which detects typical structures in the control-flow graph. It uses the architecture-specific branch cost model and execution possibilities of control-flow edges to estimate the possible layout costs of specific sub-structures. The layout with the minimal cost estimation would be chosen. We further investigate a new approach to use neural network to predict execution possibility for each edge. We choose a set of programs and record particular static information of the edges in the typical structures. These data includes the knowledge about the relationship between static program features and dynamic behavior, and is fed to train the neural network. In this paper, we adopted an improved back propagation neural network. The algorithm has been implemented based on a 5-stage pipeline UniCore architecture. The experiments show that it improves programs' performance well, and execution possibility of edges may be predicted using machine learning techniques.

**Keywords:** neural network, basic-block reordering, program structure, profile guided optimization

## 1 Introduction

The control-flow information is usually represented by control-flow graph during compiler optimization. The binary or assembly is finally laid as one-dimension sequence in the memory, which brings a gap between two-dimension graph-represented program and sequential one. Compilers will arrange the layout of binaries at the end of the compilation process. In modern pipelined microprocessors with hierarchical memory system, the performance of a binary program will be affected by its code layout in most cases. Whether the branch will be taken and the branch target address cannot be known until the branch reaches a late pipeline stage. This often brings a performance loss, which is more obvious in deeper pipelines. For example, in Digital Alpha 21164, the target address is available at the sixth stage and the average performance loss per branch prediction miss is 11 cycles. Modern processors have introduced various branch prediction strategies to improve branch performance. If the layout of one binary matches the processor's branch prediction strategy, the performance might be further improved. The layout of a binary may also affect instruction cache performance. The target of a branch might be laid

un-continuously with itself. When a branch is taken, if the branch instruction and its target are not in the same cache line, there may be some un-useful instructions fetched into the cache lines but never executed. This leads to instruction cache efficiency decrease. Further more, if frequently-interacted code fragments are laid close to each other, the possibility of conflict in instruction cache will be reduced.

Our algorithm analyzes the structures of the program and gives the optimal layout for each local sub-structure [23]. We combine structural analysis and machine learning techniques to guide the layout process. While achieving good experimental results, most of basic reordering methods are implemented based on profiling information of the program execution. One of the drawbacks of profile-based methods is the additional work of profile generation and selection. We attempted to find out if the profile information of one execution trace can be used to predict another execution trace of the program or the behavior of another program. Our method has two main phases. In the first phase the analyzer identifies various local structures from a set of programs and collects relevant profiling information. Then we use that to train a neural network mapping static features associated with each control-flow edge to an execution possibility prediction. In the second phase, we apply the trained neural network to predict the execution possibility for each control-flow edge, to feed our basic-block reordering algorithm based on local structural analysis.

Basic-block reordering using machine learning methods has several advantages. First, compared with profile-based methods, it can save programmers' work of producing and choosing proper profile information to guide compiler optimization. Second, traditional heuristic based methods require compiler writers to analyze many static features, so that to find out which set of features affects the execution of the programs most and how. Machine learning methods automatically select the information.

The rest of this paper is organized as follows. In Section 2, we discuss some related works. In Section 3, we introduce our architectural branch cost model and basic-block layout cost model. We also describe the structural analysis based basic-block reordering algorithm. Our neural network used to learn the mapping static program features to execution possibilities is addressed in Section 4. Section 5 shows the experimental results and the conclusion is given in Section 6.

## 2 Related Works

There are a number of techniques reducing branch costs, both hardware-based and software-based. Most of them use past program behavior to predict its future actions. One of the most popular compiler optimization techniques is suggested in [1]. In that paper, Pattis and Hansen use profile information of execution frequency of each edge to arrange the code placement. Although it is a greedy algorithm and the optimal layout cannot be guaranteed, many compilers and basic-block reordering techniques are based on it [2][4][5][6][12][15]. The idea of taking the different branch costs of the specific architecture into consideration is described in [12]. In [2], the authors reduce a limited form of the reordering problem to the Directed Traveling Salesman Problem which is NP-hard. They also present experimental results using heuristic algorithms

from training and testing on different data sets, which shows only a little reduction of the benefits from the code placement algorithms.

Compiler writers have crafted many heuristics over the years to approximately solve NP-hard problems efficiently [26]. Profile-based optimizations require effective profiles which are not deeply discussed in most of the works. The experimental results in [2] shows a possibility of using past program behavior to predict its future actions, like many hardware-based branch prediction techniques do. Meanwhile, the authors in [24] investigate an approach to uses a body of existing programs to predict the branch behavior in a new program at compile-time. In that paper, the authors use machine learning techniques and show their efficiency in predicting the branch behavior. Finding a heuristic that performs well on a broad range of applications is a tedious and difficult process. J.Cavazos induces heuristics in instruction scheduling [27], it also shows that inexpensive and static features can be successfully used in scheduling.

In this paper, we investigate ways to combine our structural analysis based method and the machine learning techniques to guide the layout process, and intend to get some hints related to program structures and behaviors.

### 3 Basic Block Reordering Using Structural Analysis

In this section, we give a briefly description to our local structural analysis based basic-block reordering method. A more detailed one can be found in [23]. As mentioned before, greedy algorithms usually cannot get the optimal layout. Modern programs are well structured to be divided into typical structures usually. Our method first identifies all local structures of a program. For each structure, we use structure cost model and the execution frequencies of control-flow graph edges to calculate the cost of each possible layout and the one has the minimal cost is chosen. Experimental results show that our method can improve performance by 7% on average which is better than the common used greedy method [23].

#### 3.1 Architecture-Specific Branch Cost Model

In our model, branch cost is defined as the cycles needed to execute the branch/jump instruction (if needed) plus the cost caused by the branch or jump. We are mainly concerned with reducing branch/jump cost, and instruction cache performance may be improved as well. In this paper, we will call branch instruction or jump instruction all as branch for convenience.

For a typical single-issued pipelined processor with branch prediction mechanism, the branch cost can be calculated as follows:  $Branch\ cost = Cost\ of\ branch \times Numbers\ of\ branches + Cost\ of\ branch\ prediction\ miss \times Numbers\ of\ branch\ prediction\ misses$ . Here the number of branches stands for the dynamic number of branches executed.

Different processors may have different branch cost models due to architectural difference. We analyze and conclude the branch cost model for UniCore-I processors.

But the method isn't limited to them and can be used on many other pipelined processors.

UniCore-I is a single-issued RISC microprocessor with five pipeline stages. Its branch prediction mechanism is to assume the fall-through path is always executed [14]. Since whether a branch will be taken is not determined yet, it continues to fetch the following instructions in the fall-through path. If the branch is taken, these instructions in pipeline will be squashed, and re-fetch right instructions. This branch prediction strategy is very common to be seen in HP PA-RISC, Alpha AXP-21064 and many other processors. In UniCore-I processor, a branch instruction itself will take 1 cycle in the pipeline. If the branch is taken, it will cost extra 2 cycles till branch target is calculated. Thus, a taken conditional branch will take 3 cycles in all and a not-taken conditional branch takes one. An unconditional jump will always take 3 cycles.

The branch cost is directly related to the edges in the control-flow graph. For a control-flow graph  $G$ , the branch cost of an edge  $e = \text{head} \rightarrow \text{tail}$  is defined as the cycles needed to go from head to tail, and is represented as  $Cost(e)$ . Given an architecture,  $Cost(e)$  is related to the characteristics of basic-block Head and tail and their relative positions in the linear space. Figure 1 shows the possible situations:



**Fig. 1.** Different Branch Cost of Different Edges in Control-Flow Graph

In Figure 1-(a), basic-block A has only one successor and can go through to B directly with no branch instructions, so the cost of edge  $A \rightarrow B$  is 0; In Figure 1-(b), A has only one successor B, but B is not laid just following A, thus there need be one jump instruction in the end of A, thus edge  $A \rightarrow B$  costs 3 cycles; In Figure 1-(c), A has two successors with B just following it, thus there should be one conditional branch in the end of A and edge  $A \rightarrow B$  costs 1 cycle while edge  $A \rightarrow C$  costs 3 cycles; In Figure 1-(d), A also has two successors B and C, but neither of them is laid just following A, so there should be two branches at the end of A and edge  $A \rightarrow B$  costs 3 cycles while edge  $A \rightarrow C$  costs 4 cycles.

In UniCore-I, for a control-flow edge  $e = A \rightarrow B$ , let  $Profit(e)$  be the profit of laying B directly following A. It is the cost of not laying B right following A minus the cost of laying B right following A, as shown in Table 1.

**Table 1.** The  $Profit(A \rightarrow B)$  in UniCore-I

|                                | Cost of not laying B directly following A | Cost of laying B directly following A | $Profit(A \rightarrow B)$ |
|--------------------------------|---|---------------------------------------|---------------------------|
| A has only one successor       | 3 cycles                                  | 0                                     | 3 cycles                  |
| A has more than one successors | 3 or 4 cycles                             | 1 cycle                               | 2 or 3cycles              |

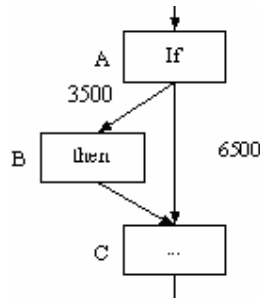
The case that  $Cost(e)=4$  is only when none of A's successors is laid directly following A, which cannot be determined before the layout is finished, and is not very common. So we assume the  $Profit(A \rightarrow B)$  to be 2 when A has more than one successors. Thus in our model, according to the number of A's successors,  $Profit(A \rightarrow B)$  is calculated as follows:

$$Profit(e) = \begin{cases} 3, & A \text{ has 1 successor} \\ 2, & Other \end{cases}$$

The final profit-weight of the edge  $Key(e)$  is calculated as follow:  $Key(e)=Profit(e) \times Frequency(e)$ .

### 3.2 Local Structural analysis and Optimization

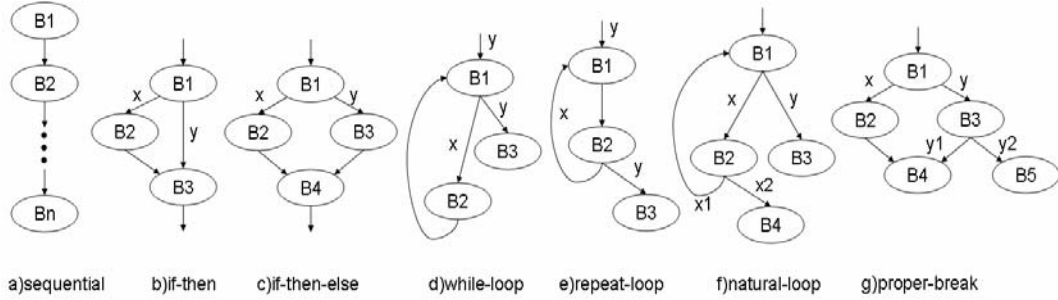
PH algorithm use profile information of execution count of each edge to arrange the code placement. Pattis and Hansen describe two code placement algorithms. The bottom-up one begins with the edge with the highest execution frequency and tries to arrange the tail node of the edge immediately following the head node. It's a greedy algorithm which cannot get the optimal layout in some cases. See Figure 2 as an example. Basic-block A, B and C construct an if-then structure, execution frequency of each edge is marked beside. Since  $Freq(A \rightarrow C) > Freq(A \rightarrow B)$  and  $Profit(A \rightarrow C) = Profit(A \rightarrow B) = 2$ ,  $Key(A \rightarrow C) > Key(A \rightarrow B)$ . Thus the program layout will be A-C in PH algorithm, not including block B in the chain. The total branch cost will be  $6500 + 3500 \times 3 + 3500 \times 3 = 27500$  (cycles). But if we lay them as A-B-C, the total branch cost will be  $3500 + 6500 \times 3 = 23000$  (cycles). This would be better than the former.



**Fig. 2.** Example of If-then Structure

Modern programs always have good structures and their control-flow graph can be divided into typical structures. Most of these structures can be represented as nine

types [9]. Among these structures, the improper interval schema doesn't always appear and has no fixed structure. The self-loop structure has only one basic-block, while case/switch structure are represented as jump tables in machine codes, so these structures aren't concerned in our work. We define 7 typical structures and analyze their optimal layout given the execution frequencies of all edges, which are shown in Figure 3.



**Fig. 3.** Seven Typical Structures in Control-flow Graph<sup>1</sup>

We will take if-then structure as an example to introduce our algorithm getting the optimal local layout.  $Freq(e)$  represents the execution count of the edge  $e$  in profile. As shown in Figure 3-(b), in if-then structure, B1 has two successors, thus the possible layout includes: B1-B2-B3 or B1-B3 (B2 is set to be a separate node). For the former layout, the branch cost is:

$$\begin{aligned} Cost_{B1-B2-B3} &= Freq(B1 \rightarrow B2) * Cost(B1 \rightarrow B2) + Freq(B2 \rightarrow B3) * Cost(B2 \rightarrow B3) \\ &\quad + Freq(B1 \rightarrow B3) * Cost(B1 \rightarrow B3) \\ &= x + 3y \end{aligned}$$

Similarly, for the latter layout, the branch cost is  $Cost_{B1-B3} = 6x + y$ . So if B1-B2-B3 is better, it demands  $Cost_{B1-B2-B3} \leq Cost_{B1-B3}$ , that is  $y \leq 2.5x$ . Similarly, if the algorithm chooses B1-B3, then it demands  $y \geq 2.5x$ . If  $y = 2.5x$ , the first layout is chosen for better instruction cache locality. The item of if-then in Table 2 shows the above results. Other structures' optimal local layouts under different frequencies of edges are also listed in Table 2.

**Table 2.** Optimal Local Layout for Each Structure under Different Conditions

| Structure    | Optimal Local Layout | Condition     |
|--------------|----------------------|---------------|
| Sequential   | Sequential           | N.A.          |
| If-then      | B1-B2-B3             | $y \leq 2.5x$ |
|              | B1-B3                | $y > 2.5x$    |
| If-then-else | B1-B2-B4             | $y \leq x$    |
|              | B1-B3-B4             | $y > x$       |

<sup>1</sup> The weight beside each edge represents the execution counts of the edge according to the profile information

|              |  |              |                                   |
|--------------|--|--------------|-----------------------------------|
| While-loop   | The other predecessor of B1's is laid directly before B1 | B2-B1-B3     | $y \leq x$                        |
|              |  | B1-B3        | $y > x$                           |
|              | Other  | B2-B1-B3     | N.A.                              |
| Repeat-loop  | B1-B2-B3   |              | N.A.                              |
| Natural-loop | The other predecessor of B1 is laid directly before B1   | B1-B3, B2-B4 | $y > x$                           |
|              |  | B1-B2-B4     | $y \leq x$                        |
|              | Other  | B1-B3, B2-B4 | $y > x$ and $x_1 < x_2$           |
|              |  | B1-B2-B4     | $y \leq x$ and $y < x_2$          |
|              |  | B2-B1-B3     | Other                             |
| Proper-break | B1-B2-B4, B3-B5  |              | $x > y$                           |
|              | B1-B3-B4   |              | $x \leq y$ and $3x + 2y_2 < 2y_1$ |
|              | B1-B3-B5, B2-B4  |              | Other                             |

Thus we can get the table of optimal local layout for each structure under different conditions. In this phase of the algorithm, local structural analysis and layout optimization is performed: The algorithm traverses the whole graph and searches for typical structures. Once a typical structure is identified, the algorithm will reorder the basic-blocks of it according to the above table and the particular execution frequencies of the edges. Let's take the weighted control-flow graph in Figure 2 for example. In that case,  $Freq(A \rightarrow B) = 6500 < 2.5 \times 3500 = 2.5 \times Freq(A \rightarrow C)$ , so the optimized layout is A-B-C.

We describe our algorithm in pseudo code as following:

```

procedure reorder (N, E)
  begin
  sort_edge(E); //Sort all edges by Profit(e)
  for e in E do //visit all edges by order of Profit(e) and link them to chains
  begin
    if A.visit = false and B.visit = false; //A, B are both not visited
      B.visit := true; A.visit := true;
      A.next := B;
      B.isTail := true; A.isHead := true;
    else if B.visit = false and A.isTail = true // B is not visited, A is tail of a chain
      B.visit := true; A.next := B;
      A.isTail := false; B.isTail := true; // B becomes tail
    else if A.visit = false and B.isHead = true // A is not visited, B is head of a chain
      A.visit := true; A.next := B;
      A.isHead := true; B.isHead := false; // A becomes its head
    else if A.isTail = true and B.isHead = true; // A, B are both visited
      A.next := B;
      A.isTail := false; B.isHead := false; //form these 2 chains
  end
  structural_analysis(N,E); //scan all chains, optimize according table 2 via structural analysis
  connect_chains(); //link all chains to form the binary
  end

```

```

procedure structural_analysis (N,E)
  begin
  for B in N (Deep-First-Order) do
    begin
      // if-then structure?
      if Succ(B)={ V1,V2}
        if Pred(V1)={B} &&Pred(V2)={B} &&Succ(V1)={V2} &&Profit (B→V1) +
Profit(V1→V2)>Profit (B→V2)
          B.next := V1; V1.next := V2; V1.visit := true;
          if Pred(V1)={B} &&Pred(V2)={B} &&Succ(V2)={V1} &&Profit (B→V2) +
Profit(V2→V1)>Profit (B→V1)
            B.next := V2; V2.next := V1; V2.visit := true;

          //test whether basic-block B and its succeeds is a typical structure, select optimal
          //layout according to Table2
          .....
        end
      end
    end
  end

```

## 4. Predictions with Artificial Neural Network

In this section, we use artificial neural network to predict the execution probability of edges in typical structures given above. Our idea is generally described as follows. A set of programs are gathered and particular static information about edges in the typical structures is recorded. Relevant dynamic behavior is associated with each edge while profiling. We have accumulated a dataset about the relationship between static program features and dynamic behaviors. This dataset is used to train the neural network to predict the behaviors of edges. In the dataset, the edges with similar static features may exist in programs not in the training set or the programs with different inputs.

This section contains two main parts. First, we introduce the static feature set extracted for the training process. Second, we describe the artificial neural network used in our prediction.

### 4.1 Static Feature Set for Prediction

To apply the neural network to our problem, the static feature set as input of neural network is firstly determined. We record the static features for each edge that are used in the algorithm above in each typical structure (see Table 3). Some features are used to identify typical structures and the edges constructing them, others are some properties related to the edges.

**Table 3.** Selected Static Features for Edges

| No. | Feature name   | Feature description  |
|-----|----------------|--|
| 1   | SS_No.         | Unique number to identify typical structure.   |
| 2   | Edge_No.       | Unique number to identify edges.   |
| 3   | I_last_of_head | The operation code of the last instruction in the basic-block of edge's head.        |
| 4   | Br_direction   | Branch direction.  |
| 5   | I_pre_last     | The operation code of the instruction before the last instruction in the edge's head |
| 6   | Is_bl          | Whether the last instruction in the edge's head is a                                 |
| 7   | Is_swi         | Whether the last instruction in the edge's head is SWI.                              |

B. Calder [24] brings neural network and static features of program in their evidence-based static branch prediction (ESP). We've referred these suggestions to choose these features. The most important difference is that, [24] considered every single branch in a program, while we are concerning edges in an identified typical structure. A lot of static information, such as whether the edge is a part of a loop, etc, has already been contained in the type of a typical structure.

We choose the feature set in Table 3 based on several criteria. First, we adopt information which would be likely predictive of behavior. This information includes two parts. One is a unique number for each typical structure. The other unique number is for edges. Second, we use encode some information related to a given edge according to classical static branch prediction methods, such as the operation code of the last instruction in the edge's head basic-block, branch direction and etc.

We profile the programs to collect information on execution counts of edges and its head basic-blocks. Since the execution counts of edge varies with programs, we normalized it by its head basic-block execution counts, that is, execution probability of edge is  $\text{execution\_counts\_of\_the\_edge} / \text{head\_basic\_block\_execution\_counts}$ . Finally, we associate the static features of each edge with the corresponding execution probability.

## 4.2 Artificial Neural Network Building

Our goal is to build a tool that can effectively predict the edge's execution probability based on static program features. It should accurately predict not only for the programs it studied, but also for other programs it is never acquainted with.

We use an improved back propagation neural network, which is not fully connected and has shortcut connections [20]. A cascade model is used to train it. The cascade training model differs from the ordinary ones in the sense that it starts with an empty neural network and then adds neurons incrementally. The basic idea of this model is that some neurons are trained separately, and the most promising candidate is inserted. Then the output connections are trained and new candidate neurons is prepared. The candidate neurons are created as shortcut connected neurons in a new

hidden layer, which means that the final network will consist of a number of hidden layers with one connect neuron in each. We choose RPROP algorithm as its internal training algorithm of this model [20]. And the candidate neurons' activation functions may be one of the followings.

The neural network we adopted has only one hidden layer. Thus, the structure is 7-X-1. Here X is the number of neurons in the hidden layer, which is determined in the learning phase. The activation function for the candidate neuron may be one of the following:

$$\begin{aligned} \text{Sigmoid: } y &= \frac{1}{1 + e^{-2 \times s \times x}}; \\ \text{Sigmoid\_symmetric: } y &= \tanh(s \times x); \\ \text{Gaussian: } y &= e^{-x \times s \times x \times s}; \\ \text{Gaussian\_symmetric: } y &= e^{-x \times s \times x \times s} \times 2 - 1; \\ \text{Elliot: } y &= \frac{\frac{s \times x}{2}}{1 + |s \times x|} + 0.5; \\ \text{Elliot\_symmetric: } y &= \frac{s \times x}{1 + |s \times x|}; \end{aligned}$$

The output function is,

$$y = \begin{cases} \tanh(s \times x) & \text{if } \tanh(s \times x) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Here x is the biased linear combination of the outputs of the hidden candidate neurons. The learning criterion is to minimizing the Mean Square Error. More details and discussions about the neural network are available in the next section.

## 5. Experimental Results and Analysis

In this section we present our experimental results. We have implemented the method described above as a post pass of GNU toolchain (gcc 3.2.1). We use integer benchmarks in SPEC2000 as our neural network training data and use several programs from MediaBench[14] as our final benchmark test data. Each program is compiled with GCC “-O2” level of optimization. The performances are measured using sim-pipeline: a cycle-accurate five-stage pipeline UniCore simulator modified from sim-outorder in simplescalar[16]. The main configuration for UniCore-I processor simulator is listed in Table 4, and the benchmark programs are briefly described in Table 5.

**Table 4.** Simulation Parameters of UniCore-I Processor

|                                 |           |
|---------------------------------|-----------|
| <b>Branch Prediction Scheme</b> | Not taken |
| <b>Level 1 I-Cache</b>          |           |

|                              |   |
|------------------------------|---|
| <b>Capacity</b>              | 8K                                      |
| <b>Associativity</b>         | 2                                       |
| <b>Block Size</b>            | 32 Bytes                                |
| <b>Replace Scheme</b>        | Round-Robin                             |
| <b>Level 1 D-Cache</b>       |   |
| <b>Capacity</b>              | 8K                                      |
| <b>Associativity</b>         | 4                                       |
| <b>Block Size</b>            | 32 Bytes                                |
| <b>Other Schemes</b>         | Round-Robin, Write back, Write allocate |
| <b>Memory Access Latency</b> |   |
| <b>First Chunk</b>           | 20 Cycles                               |
| <b>Inter Chunk</b>           | 2 Cycles                                |

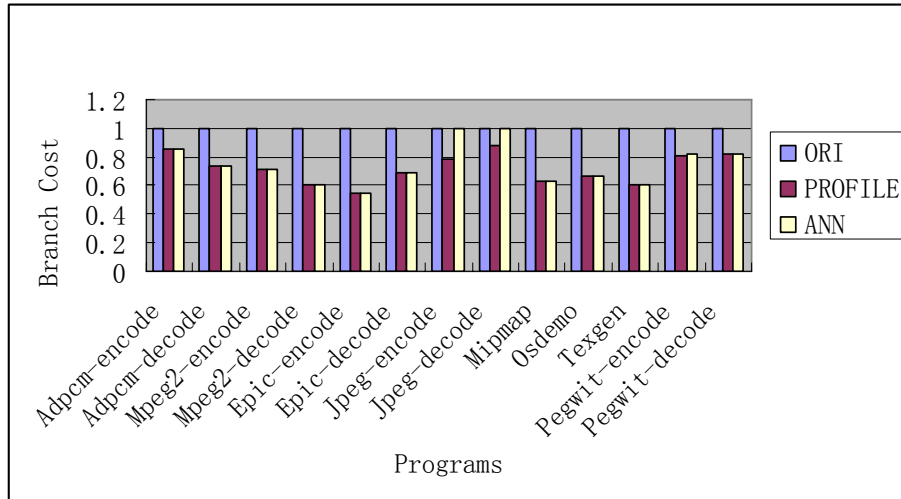
Table 5. Benchmark Description

| <b>Name</b> | <b>Description</b>  |
|-------------|---|
| adpcm       | Adaptive differential pulse code modulation, one of the simplest and oldest forms of audio coding (encode / decode) |
| epic        | An experimental image compression utility (epic / unepic)   |
| pegwit      | Public key encryption and authentication (encode / decode)  |
| jpeg        | Standardized compression method for full-color and gray-scale images (encode / decode)                              |
| mesa        | A 3-D graphics library clone of OpenGL (mipmap / osdemo / texgen)   |
| mpeg2       | A standard for high quality digital video transmission (encode / decode)  |

### Experimental Results

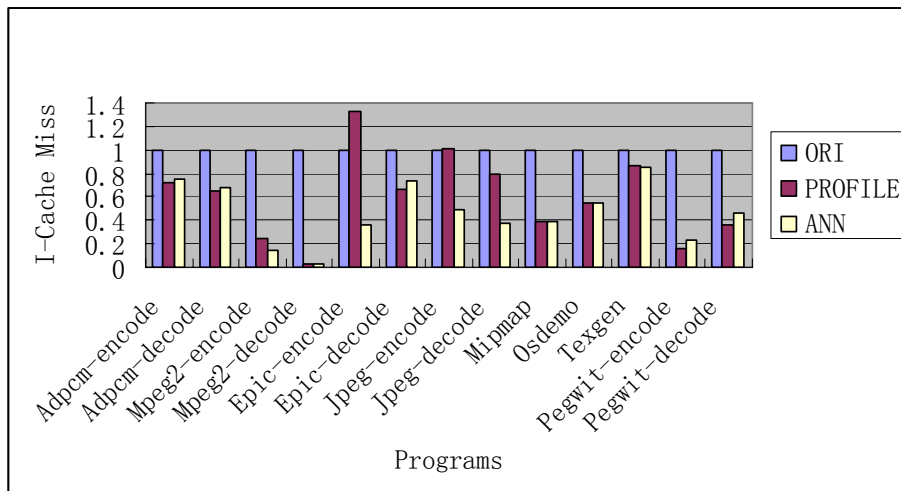
Experimental results are shown in Figure 4-6 (ORI means original program. PROFILE means using profiling optimization directly. ANN means using artificial neural network built above).

While applying the cost model mentioned in Section 3 on UniCore-I architecture, the *branch\_inst\_cost* is 1 cycle and *branch\_miss\_cost* is 2 cycles, so that the branch cost of UniCore-I is calculated as *branch instruction counts + 2\* branch prediction misses*. As shown in Figure 4, ANN can get an average branch cost reduction of 25%, which is surprisingly close results compared with PROFILE (28%).



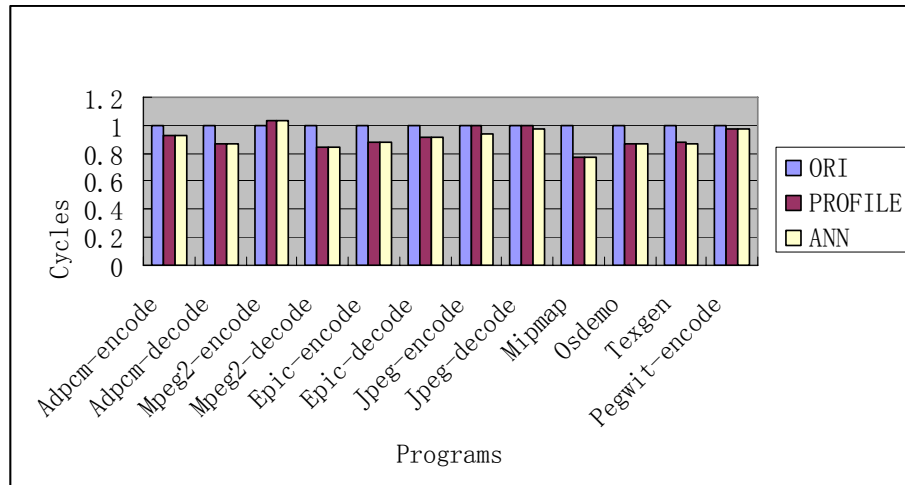
**Fig. 4.** Normalized Branch Cost

The influence of instruction cache misses is compared and shown in Figure 5. Generally speaking, both the PROFILE and ANN methods achieve a considerable improvement on I-Cache efficiency. However, sometimes ANN wins PROFILE. One possible reason might be that our basic-block reordering algorithm is not specially tuned for I-Cache and the optimal layout is only inside single structures. ANN brings some random factor into the algorithm and gets good result sometimes.



**Fig. 5.** Normalized Instruction Cache Misses

The comparisons on the whole performance is shown in Figure 6. On this aspect, PROFILE and ANN make similar improvement, 9% and 8% separately.



**Fig. 6.** Normalized Cycle Counts

As we can see above, the result of using artificial neural network to predict edge execution probability for the algorithm is sometimes even better than using profile information and sometimes worse. And on average, they are very close. The result is better than our primal assumption, especially in the situation that the training of the neural network uses different set of programs from the testing benchmarks. Future analysis is still needed in order to get further understanding.

## 6. Conclusions and Future Work

In this paper, we described our basic-block reordering method which combines program structural analysis and the neural network technique to give a proper layout for each typical structure. Experiments show that this method can improve program performance and structural analysis based feature extraction can be effective on static prediction of branch possibility. Program structural analysis can provide detailed information about the control flow structures, thus it may also provide a set of important and effective static features for other optimizations based on machine learning. Computer architecture and compiler optimization are related to many concrete applied fields. As the computer systems are becoming more and more complex, it is also harder for researchers to derive conclusions from so many factors and details. We believe applying machine learning and other techniques which are

commonly used in computer application fields upon the researches on computer system itself can further help researchers understand and optimize the system.

## References

- 1 K.Pettis, R.C.Hansen, Profile Guided Code Positioning, Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, Vol. 25, No. 6, June 1990, pp. 16-27.
- 2 S.McFarling, Program optimization for instruction caches. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.
- 3 W.Hwu and P.Chang, Achieving High Instruction Cache Performance with an Optimizing Compiler. In 16th Annual International Symposium of Computer Architecture, 1989, pp.242-251.
- 4 D. Zhu, Research on Branch Mechanism for UniCore Architecture. Ph.D. thesis, 2004.
- 5 Y. Gao, Basic Block Reordering on UniCore Architecture, master thesis, 2005.
- 6 R.Muth, S.K.Debray, S.A.Watterson, and K.De Bosschere, alto: a link-time optimizer for the Compaq alpha. *Software – Practice and Experience*, 31(1):67-101, 2001.
- 7 C.Young, D.S.Johnson, D.R.Karger, and M.D.Smith, Near-Optimal Intraprocedural Branch Alignment, Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation, June 1997, pp. 183-193.
- 8 M. Sharir, Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers, *Computer Languages*, vol. 5, nos 3/4, 1980.
- 9 S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, 1997.
- 10 UniCore32 ISA and Programming Manual. Microprocessor Research and Development Center of Peking University, 2002.
- 11 B.D.Bus, B.D.Sutter, L.V.Put,D.Chanet, and K.D.Bosschere, Link-Time Optimization of ARM Binaries. Proceedings of the 2004 Conference on Languages, Compilers, and Tools for Embedded Systems, June 2004, pp. 211–220.
- 12 B.Calder and D.Grunwald, Reducing Branch Costs via Branch Alignment. Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating System. Oct. 1994, pp. 242-251.
- 13 N.Gloy, T.Blackwell, M.D.Smith and B.Calder, Procedure placement using temporal ordering information. Proceedings of the 30th Annual International Symposium of Microarchitecture, Dec.1997, pp.303-313.
- 14 C.Lee, M.Potkonjak, W.H.Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. Proceedings of the 30th Annual International Symposium on Microarchitecture, Dec.1997.
- 15 R.Cohn, P.Goodwin, G.Lowney and N.Rubin, Spike: an optimizer for Alpha/NT executables. USENIX Windows NT Workshop, August 1997.
- 16 <http://www.simplescalar.com/>
- 17 D.A.Jimenez, Code Placement for Improving Dynamic Branch Prediction Accuracy, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), pp. 107–116, June, 2005.
- 18 H.Aydin and D.Kaeli, Using cache line coloring to perform aggressive procedure inlining. *ACM Computer Architecture News*, vol. 28, no. 1, 2000.
- 19 A.H.Hashemi, D.R.Kaeli, and Brad Calder. Procedure Mapping Using Static Call Graph Estimation. Workshop on the Interaction between Compilers and Computer Architectures, February 1997.
- 20 M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation

- learning:the RPROP algorithm. Proceedings of International Neural Networks. San Francisco,1993.
- 21 A.H.Hashemi, D.Kaeli and B.Calder,. Efficient Procedure Mapping using Cache Line Coloring. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 171-182, June 1997.
  - 22 A.Ramirez, J.L.Larriba-Pey and Mateo Valero, Software Trace Cache, IEEE Transactions on Computers, Vol. 54, No.1, pp. 22-35, Jan. 2005.
  - 23 X. Liu, Y. Yang, J. Zhang and X. Cheng. A Basic Block Reordering Algorithm Based On Structural analysis. Technical Report. 2006.
  - 24 B. Calder, D. Brunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer and B. Zorn. Evidence-based Static Branch Prediction Using Machine Learning. ACM Transactions on Programming Languages and Systems, 1997.
  - 25 S. Haykin. Neural Networks: A Comprehensive Foundation. Second Edition. Prentice-Hall, Inc,1999.
  - 26 M.Stephenson, S.Amarasinghe, M.Martin, U.M.O'Reilly, Meta Optimization: Improving Compiler Heuristics with Machine Learning. Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), June, 2003.
  - 27 J.Cavazos, J. Eliot B. Moss, Inducing Heuristics To Decide Whether To Schedule. Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI), June, 2004.