Code Reordering on Limited Branch Offset

YU CHEN and FUXIN ZHANG
Chinese Academy of Sciences

Since the 1980's code reordering has gained popularity as an important way to improve the spatial locality of programs. While the effect of the processor's microarchitecture and memory hierarchy on this optimization technique has been investigated, little research has focused on the impact of the instruction set. In this paper, we analyze the effect of limited branch offset of the MIPS-like instruction set [Hwu et al. 2004, 2005] on code reordering, explore two simple methods to handle the exceeded branches, and propose the bidirectional code layout (BCL) algorithm to reduce the number of branches exceeding the offset limit. The BCL algorithm sorts the chains according to the position of related chains, avoids cache conflict misses deliberately and lays out the code bidirectionally. It strikes a balance among the distance of related blocks, the instruction cache miss rate, the memory size required, and the control flow transfer. Experimental results show that BCL can effectively reduce exceeded branches by 50.1%, on average, with up to 100% for some programs. Except for some programs with little spatial locality, the BCL algorithm can achieve the performance, as the case with no branch offset limitation.

Categories and Subject Descriptors: D.3.4 [**Processors**]: Code generation

General Terms: Algorithms

Additional Key Words and Phrases: Code reordering, Godson Processor, link-time optimization

ACM Reference Format:

Chen, Y. and Zhang, F. 2007. Code reordering on limited branch offset. ACM Trans. Architec. Code Optim. 4, 2, Article 10 (June 2007), 23 pages. DOI = 10.1145/1250727.1250730 http://doi.acm.org/10.1145/1250727.1250730

1. INTRODUCTION

The ever-increasing gap in performance between memory and processor introduced the need for fully utilizing the existing memory hierarchy. Since the 1980's, code reordering has gained popularity as an important way to improve the spatial locality of the programs without introducing any hardware cost. Code reordering successfully reduces the instruction cache miss rate and the execution time of the programs, as well as power consumption.

This research was supported by the National Science Foundation Project of China.
Authors' address: Y. Chen and F. Zhang, Computer Architecture and System Laboratory, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080.

The reordering effect is influenced by memory hierarchy and the microarchitecture of the processor. Another point that may have been ignored is the instruction set. A flexible instruction set will provide great support for code reordering, while some limitations of the instruction set, such as a short branch offset, will decrease the benefits.

Our study is based on Godson 2C [Hwu et al. 2005] processor, the first general-purpose microprocessor developed in China. It is a 64-bit, 4-issue, out-of-order execution RISC processor. It implements MIPS-like instruction set, which includes a MIPS-III instruction set and several newly defined instructions. The 16-bit branch offset of the branch instructions in MIPS-III sets limitations to reordering optimization.

The rest of this paper is organized as follows: in Section 2, we discuss the previous work on code reordering. Section 3 describes the limitation of branch offset on code reordering of SPEC2000 integer benchmark and discusses the factors that lead to exceeded branches. In Section 4, we give two simple methods to handle the exceeded branches. We then propose the bidirectional code layout (BCL) algorithm to reduce the number of exceeded branches and analyze the performance improvement. Finally Section 5 concludes.

2. RELATED WORK

Many reordering algorithms have been proposed to improve instruction cache performance.

Early work, such as McFarling [1989], proposes a basic block-remapping algorithm. They use basic block execution frequency as the criterion for reordering and incorporate instruction exclusion to increase instruction cache hit rate.

Pettis and Hansen [1990] use edge execution frequency as a reordering criterion (the PH algorithm) after showing that the hottest basic blocks do not necessarily account for the hottest execution trace. The edges between the basic blocks contain the local information of the execution path, so the algorithm surpasses the previous work by exploiting more spatial locality. Using basic block reordering, procedure ordering, and procedure splitting, the proposed algorithm builds a call graph by traversing edges in decreasing edge weight order with a closest-is-best strategy. Pettis and Hansen's work serves as the basis for many later code-reordering algorithms [Gloy et al. 1998; Cohn and Lowney 2000; Ramirez et al. 2005].

Hashemi et al. [1997] take a new approach to reduce cache conflict misses by proposing a cache-line coloring algorithm. The algorithm keeps track of the cache blocks (colors) used by each procedure to take into consideration the call graph, procedure size, and cache line size. Their work can accurately map procedures in a "popular call graph", even if the size of the graph is larger than the size of the instruction cache.

Unlike Hashemi et al. [1997], Kalamatianos et al. [1998] take the higher-order generation conflict misses into consideration. Their work uncovers conflict misses between procedures, many procedures away on a call chain, as well as on different call chains. The proposed technique exploits procedure-level temporal interaction, using a structure called a conflict miss graph (CMG). In CMG, every

edge weight is an approximation of the worst-case number of misses between two competing procedures. The edge weights are used for color-based mapping to reduce conflict misses between procedures lying either in the same or in different call chains.

Both Hashemi et al. [1997] and Kalamatianos et al. [1998] work at procedure level. They avoid basic block reordering because this step introduces extra code and complicates compilation. However, basic block reordering is advantageous, as suggested in both papers: it will produce more compact procedure body and facilitate the coloring step by reducing the actual number of colors required by each procedure.

Parameswaran et al. [2001] take a heuristic approach when relocating the selected sections of code within the main memory to reduce the conflict misses. Code is reordered by examining the execution frequency of basic blocks and by placing code segments with high execution frequency next to each other within the cache. Parameswaran et al. [2004] improve the work of Parameswaran et al. [2001] by proposing a methodology that can be applied for configurations with cache line size or associativity greater than one.

Parameswaran et al. [2001; 2004] work on basic block level. Their work is different from the PH algorithm in that they first place the basic blocks in the cache, then map them into main memory. In order to reduce conflict misses, the blocks are spread apart in the memory. The gaps will be filled by later blocks, but the memory size required by the program is inevitably increased. In addition to the issues of memory size, this approach will not fully exploit the fetch efficiency of the processors. When hot blocks are packed together, it is possible that the fetch of one hot block will fetch in the first several instructions of the next hot block. However, when these blocks are far away in memory, the fetch width will not be fully utilized.

The code-reordering techniques reviewed so far all belong to static time optimizations. Recently, researchers have been exploring other optimizing opportunities through runtime optimization [Bala et al. 2000; Lu et al. 2004]. The speedup obtained by the method proposed by Bala et al. [2000] is comparable to that of the best static optimization. Because the analysis and optimization of the program both incur runtime overhead, it is hard to achieve an additional improvement as static optimization has achieved. Lu et al. [2004] utilize the performance counter of itanium-64 to reduce the overhead of program analysis, obtaining a speedup of about 4%, in the best case.

These previous works mainly focus on the benefit that code reordering can introduce. In this paper, we present the influence of the instruction set on reordering optimization and our way of dealing with the problem.

3. EXCEEDED BRANCHES IN CODE REORDERING

For the experiments presented in this paper, we use the Godson link-time optimizer GLTO [Chen et al. 2006], which is ported from alto [Muth et al. 1999] (the link-time optimizer for Compaq Alpha). The improved PH algorithm already implemented in alto serves as the basic reordering algorithm for our experiments. The branch offset in Godson ISA (MIPS-like instruction set) is

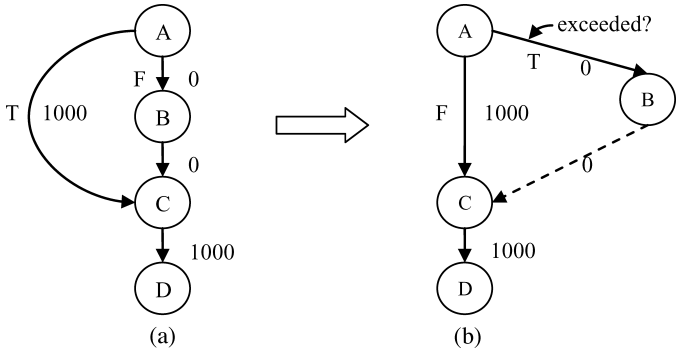

Fig. 1. The typical sequence of blocks being reordered. (a) The original sequence of blocks; (b) reordered blocks.

a 16-bit long signed immediate, which can reach 32 K (2^{15}) instructions both forward and backward.

3.1 Exceeded Branches in Code Reordering

The basic idea of code reordering in algorithms, such as the PH algorithm, is to keep the hottest codes compact together, which represents the hottest path in execution. The hot codes are packed into the first several pages while the cold or never-executed codes are pushed far away.

Figure 1 shows a typical sequence of blocks being reordered. During reordering, the frequently executed blocks A, C, and D would be laid in a straight line, and block B would be pushed out of the way. Thus the original graph (a) will be changed to (b). To ensure the correctness of the program, block B should still be connected with blocks A and C. For blocks B and C, one jump instruction will be introduced (the dashed line in Figure 1). As for blocks A and B, the branch in block A should ensure the connection between them. The problem comes from the fact that the distance between A and B can be so large that the branches of the instruction set cannot afford such a long jump.

Those branches going out of offset bound in code reordering are defined as exceeded branches in this paper. The problem of laying out all blocks with no exceeded branches to occur is equivalent to the "bandwidth problem," which is proved to be NP-complete.

3.2 Factors Related with Exceeded Branches

The branch offset of the target ISA and the reordering algorithm both influence the number of exceeded branches. Besides, there are several factors inherent in the program itself that have an effect on the number of exceeded branches.

3.2.1 *Total Number of Instructions.* If the total number of instructions of the whole executable is less than 32 K, no branches will go out of bound no matter how the executable is reordered. This is a sufficient condition. Since a branch can reach 32 K both forward and backward, any instruction is within reach.

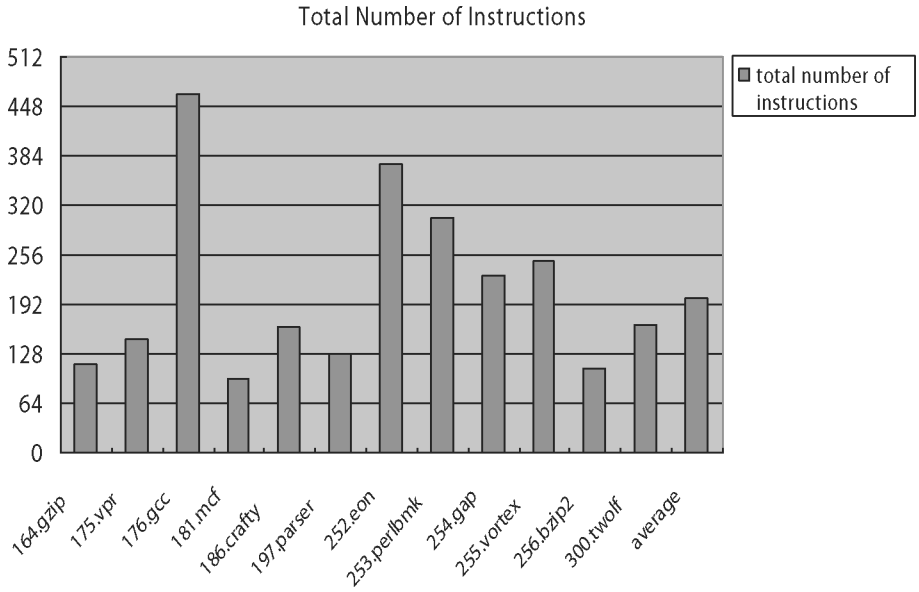

Fig. 2. The total number of instructions (K) of SPEC2000 integer benchmark.

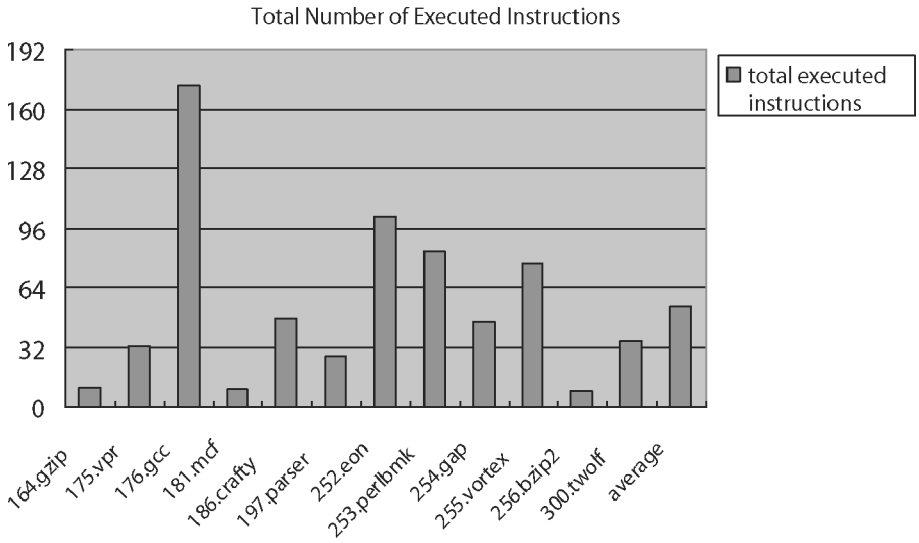

Fig. 3. The total number of executed instructions (K) of SPEC2000 integer benchmark.

As shown in Figure 2, the number of the instructions for the SPEC2000 benchmarks ranges from 96 K (mcf) to 468 K (gcc), all exceeding 32 K. The sufficient condition is not satisfied; thus, it is possible to have branches exceed the limit.

3.2.2 *Total Number of Executed Instructions.* Figure 3 gives the total number of executed instructions of each SPEC2000 benchmark with reference input. Note that one instruction will be counted only once no matter how many times

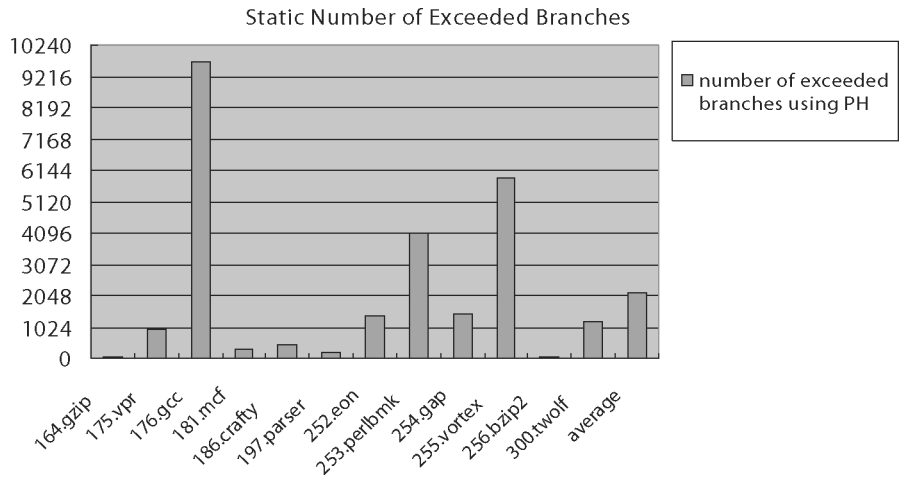

Fig. 4. Static number of branches exceeding the limit.

it is repeated in execution. As shown in Figure 3, except gzip, mcf, parser, and bzip2, other benchmarks all have executed more than 32 K instructions. The worst case is gcc, with a number of instructions in excess of 160 K.

The large number of executed instructions does not directly lead to the branch offset limits being exceeded. It indicates the chance of having exceeded branches. According to algorithms, such as the PH algorithm, the total number of executed instructions represents the distance from the hottest instruction to the nearest nonexecuted instruction in layout. If more than 32 K instructions are executed, the distance from the hottest to the first nonexecuted will exceed the 16-bit signed offset. Thus, a hot branch with a cold target will exceed the offset limit when the distance is greater than the branch can reach.

3.2.3 *Interleaving of Hot and Cold Code.* Another important factor that leads to exceeded branches is the interleaving of the hot and cold code in the executable. In Figure 1, if block B is also executed many times (not much fewer than A and C), it will not be pushed out so far away. When this factor is combined with the second factor, many branches go out of limit. The hot blocks are laid at the beginning of the binary and the cold blocks will be laid behind them in algorithms similar to the PH algorithm. Then, the distance between the hot code and the cold one is really large, as in the gcc benchmark.

Figure 4 reports the static number of exceeded branches when the programs are reordered by the PH algorithm. Although for small-size programs, such as gzip, mcf, parser, and bzip2, only a small number of branches exceed the limit, the problem for large-sized programs like gcc, perlbmk, and vortex is quite serious. Although the code size and static executed instructions of eon are really large, not too many branches exceed the limit. This is because in eon there are not so many interleaving of hot and cold code. On average, there are more than 2048 exceeded branches in MIPS-like instruction set, while in Alpha (21-bit branch offset) no branches go out of limit. The exceeded branches have to be dealt with to keep the correctness of the programs.

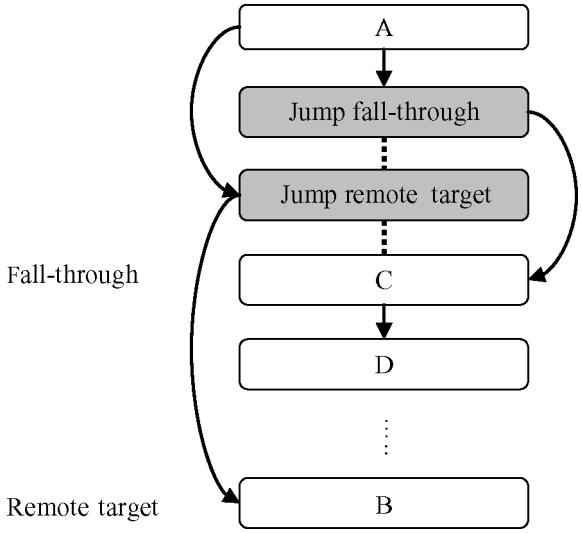

Fig. 5. The direct jumping method. Arrows connect two successive blocks in the control flow. Heavy dashed lines connect two spatially neighboring blocks with no relationship in the control flow. Light dashed lines indicate the presence of blocks not displayed in the schema.

4. CODE REORDERING ON LIMITED BRANCH OFFSET

In this section, we present two simple methods to handle the exceeded branches. We then propose the bidirectional code layout (BCL) algorithm to reduce the number of exceeded branches.

4.1 Direct Jumping Method

The most direct way to handle the exceeded branches is explained in Figure 5. We insert another two unconditional jumps between the original branch block A and its fall-through neighbor block C. The unconditional jump in MIPS ISA has a 26-bit offset that is long enough for most applications on 32-bit platform.

The first jump (jump fall-through) is inserted directly after block A and the target of the jump is the fall-through block C. The second jump becomes the new target of the original branch and its own target is block B. Since the jump instruction bears a long enough offset, the correctness of the program is ensured.

However, we can see this method significantly hurts performance. The move of block B to a distant place means the A to C path is more frequently executed. When we compensate for the exceeded branches in this way, the additional jump to block C will be executed as many times as is the A to C path. The continuous control-flow transfer instructions (a branch followed by a jump) are not preferred by most microprocessors, since they all refuse to issue two control-flow transfer instructions at the same cycle. This means a decrease in issue efficiency and overall performance.

Furthermore, the insertion of the two jump blocks (4 instructions, 2 jumps, and 2 delay slots) pollutes the cache, since they are not real effective

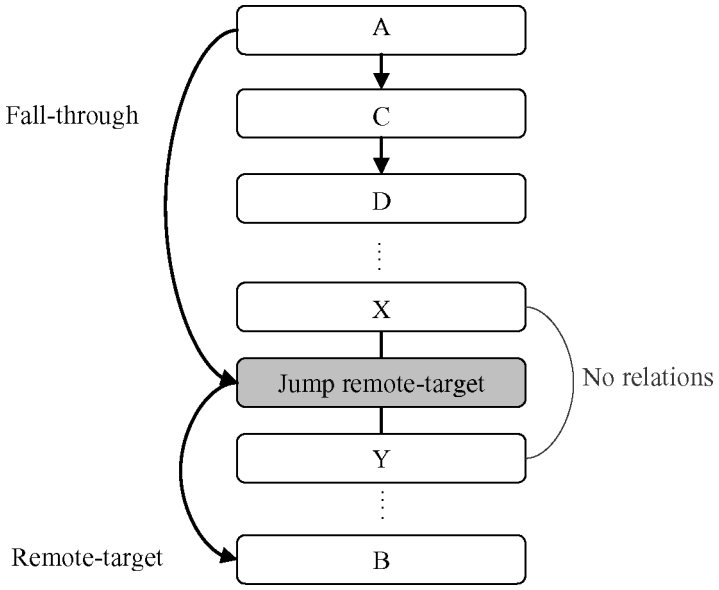

Fig. 6. Distant jumping method.

instructions, especially the latter jump to the remote target. When the branch block A and the fall-through block C are next to each other, the fetch of block A is likely to fetch in the first several instructions of block C; however, when inserted with four instructions, the fetch efficiency is decreased.

The effect of the direct jumping method on SPEC benchmarks will be given together with the later algorithms in Section 4.4.

4.2 Distant Jumping Method

An improved approach is described in Figure 6. Instead of adding two jumps, we only use one jump to get to the real target. The jump is inserted at some point far away from the branch, but still within reach. This keeps block A and its fall-through block C close together; thus no jumps are needed to ensure their relation. When we insert the jump, we would try to choose two consecutive blocks with no relations, such as one block with an unconditional jump and its following block (blocks X and Y, as shown in Figure 6). In this way, the inserted jump will not require any additional jumps for keeping the relation of the two blocks.

This method overcomes the shortcomings of the direct jumping method. On the one hand, the most frequently executed blocks are still kept together, so fetch efficiency is not decreased. On the other hand, the jump to the remote target is moved away to some cold part of the program, thus keeping the code density high and the instruction cache miss rate low, as a result.

However, the two methods share the same limitation. The insertion of jumps will increase the code size, leading to some branches exceeding the limit, which, in turn, require more jumps. A more efficient method is to change the reorder algorithm to make fewer out-of-bound branches occur.

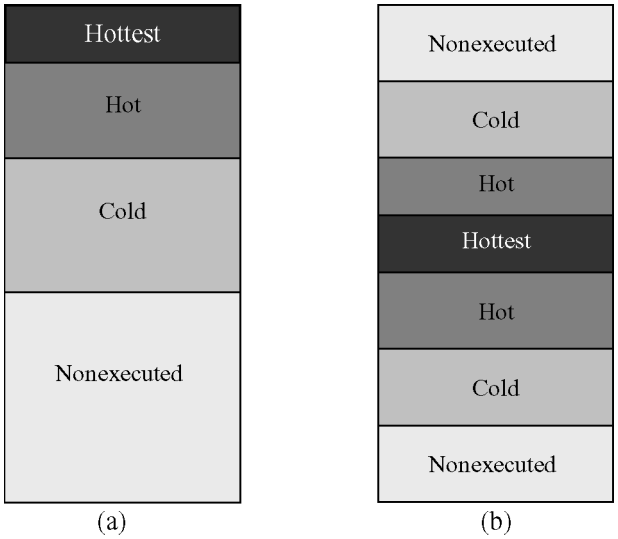

Fig. 7. The code density layout of reordered programs. (a) Code layout of previous algorithm; (b) code layout of BCL algorithm.

4.3 Bidirectional Code Layout (BCL) Algorithm

Previous algorithms in the line of the PH algorithm all put the hottest code at the very beginning of the code segment, the less hot code following it and the coldest part at the very end. These algorithms treat the code space as a unidirectional linear space. As shown in Figure 7a (the color indicates the hotness), the hotness of the code decreases as it is reaches its end.

This method works quite well if the branches never exceed the bound, as SPEC2000 benchmarks on Alpha ISA with the PH algorithm. However, on MIPS-like ISA, as we stated before, there would be a certain number of branches going out of limit. The situation further degrades for larger programs with drastic interleaving of hot and cold code.

One solution is to treat the code space as bidirectional instead of unidirectional. With such a view, we set the hottest part at the midpoint of the entire code space and the less frequent code extending both forward and backward, as shown in Figure 7b. In this way, we exploit the fact that branches have a signed offset that can reach both forward and backward. By laying out code in a bidirectional fashion, the distance between the hot and cold code is greatly reduced and the number of exceeded branches is greatly reduced.

We build the BCL algorithm by extending the improved PH code reordering algorithm already implemented in alto. Improvements include the strategy of sorting the chains, the careful management of chains to avoid cache conflicts, and the bidirectional layout. BCL is composed of three steps, basic-block chaining, super chain forming, and chain layout.

4.3.1 *Basic-Block Chaining.* BCL algorithm is based on the profile information collected through running the instrumented programs with training input. The profile information contains the execution count of basic blocks and

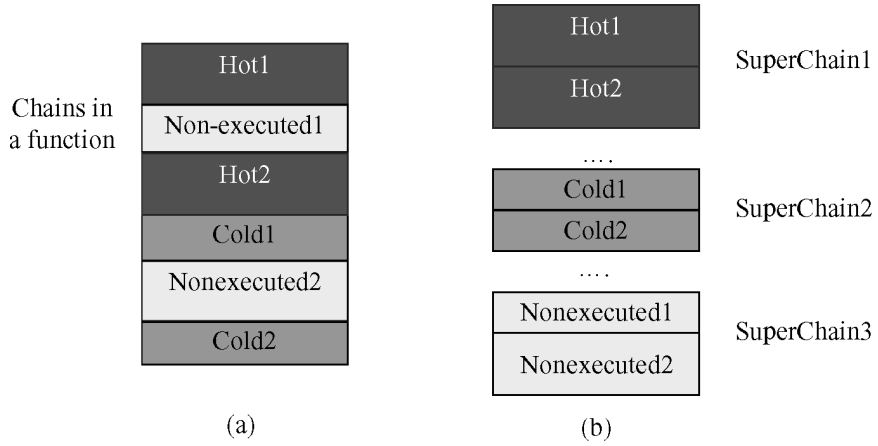

Fig. 8. Super chain forming.

edges. With this information, BCL sorts edges in descending order and, each time, selects the edge with the highest count and connects the two basic blocks.

The chains will be extended just as in the PH algorithm. When the last block of the chain is the head of the processing edge, the tail block of the edge will be connected to the chain; when the first block of the chain is the tail of the processing edge, the head block of the edge will be connected to the chain.

The chaining process stops when all the executed edges have been dealt with.

4.3.2 *Super Chain Forming.* This step is the same as in the improved PH algorithm implemented in alto. After the first step, in some functions, the hot chains are disconnected, such as two hot loops in the same function, with a cold part in between. The goal is to connect these separate chains in the same function together to improve spatial locality. Besides hot chains, the nonexecuted chains of the same function will also be connected together and the rest of the function will be chained together.

The view of a function before super chain forming is shown in Figure 8a. The hot, cold, and nonexecuted chains are interleaved. After this step, the view is like Figure 8b. The hot chains are connected together for the sake of spatial locality. The cold chains are connected together so as to reduce the chance of occurrence of exceeded branches. If the cold chains of one function are scattered in the layout, it will even be possible that branches from one cold chain to the other will go out of bound.

4.3.3 *Chain Layout.* This final step lays out the chains with the aim of minimizing the number of exceeded branches. Instead of simply laying out the chains sequentially according to the execution frequency as done in the PH algorithm, we adopt the following steps:

1. Sort the chains. Existing layout algorithms generally sort the chains simply according to the execution frequency of the hottest block or the hottest

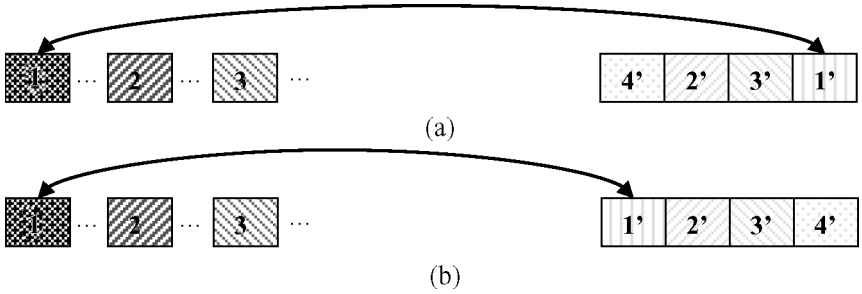

Fig. 9. Sorting super chains according to the hotness of the related chains.

edge in the chain. This causes the problem that chains with equal frequency are sorted in no particular order. BCL sorts the chains of equal or similar frequency by taking into consideration the position of its related parts. Two chains are related if there is a branch from a basic block in one chain to another basic block in the other chain.

We divide all the chains into several subsets according to the execution frequency. Chains in the same set have similar execution frequency. The set with the hottest chains are sorted first. When sorting the colder chains in one set, the sorting measurement will be adjusted to the hot degree of its related chains, i.e., if there is a related hot chain already sorted; the chain being processed will be given a higher priority.

In Figure 9, each small rectangle represents a super chain. The super chains marked with the same number are related chains, i.e., Chain1 and Chain1′ are related; Chain2 and 2′ are related; Chain3 and 3′ are related; Chain4′ is a separate chain. The color of the rectangles indicates the hotness of the super chain. Chain1 is hotter than 2; 2 is hotter than 3. Assume that the 1′, 2′, 3′, and 4′ all have not been executed. In existing algorithms, Chain1′, 2′, 3′, and 4′ will be sorted with no particular order—maybe 4′, 2′, 3′, and 1′ in (a) with quick sort algorithm. When sorted like this, Chain1′ will be processed after Chain4′, 2′, and 3′ in code layout. Then, even if we lay the code bidirectionally, Chain1′ will be far away from Chain1.

The BCL algorithm sorts the chains with similar frequency according to the hotness of their related chains. Thus, the order will be 1′, 2′, 3′, and 4′, as shown in Figure 9b. This technique succeeds in keeping the related chains relatively close to each other (Chain1 and 1′) while pushing those isolated chains (Chain4′) away. Since the adjustment is done within the same set, it will not affect the performance.

2. Select the hottest chain from the sorted chains and lay it next to the prior chain selected.
3. Repeat step 2 until the code size of all the selected chains reaches the limit size N.

Analysis shows that the number of static instructions that account for 95% of runtime instructions ranges from 303 (mcf) to 26,072 (gcc), with an average of 5627. Except for gcc, the number for most programs is no more

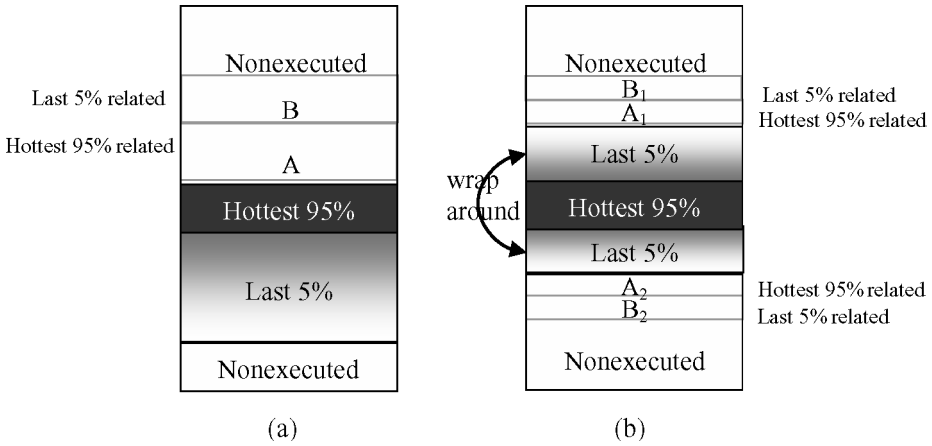

Fig. 10. The different layout of the last 5% executed code.

than 8000, which will occupy about 32 KB of code space. In order to keep the hottest code compact, we define N as shown in Formula (1) to be the size of static instructions, accounting for 95% runtime execution frequency or 32 KB, whichever is less. Experimental results show good performance when N is chosen this way.

$$N = \text{minimum (static size of hottest 95\% runtime instructions, 32 KB)} \quad (1)$$

The argument for choosing N as 95% of the size of executed instructions, even if the static size of executed instructions is less than 32 K, is as follows. As pointed out in Section 3.2, the total number of static instructions (not the executed instructions) within 32 K is a sufficient condition. However, if the static size of all executed instructions is less than 32 KB, there are still chances for branches to exceed the limit, just as gzip and mcf. The size of the last 5% runtime instructions are usually much larger than that of the hottest 95% runtime instructions. Our experimental results show that the last 5% range from 4.5 times (crafty) to 148 times (vpr) larger than the hottest 95%.

As shown in Figure 10, if we just layout the 100% executed code sequentially, with the later nonexecuted code laid bidirectionally, the layout will be like Figure 10a. However, if we start bidirectional layout from the last 5% of the code, we can wrap the last 5% executed code around the hottest 95% as shown in Figure 10b. Figure 10a is an unbalanced layout; it does not fully utilize the code space. The nonexecuted related with the hottest (region A in Figure 10a) will be laid upon the hottest 95% code. Then, the distance from the nonexecuted related with the 5% code (region B in Figure 8a) to the last 5% code will be larger than in Figure 8b. As shown in Figure 10b, the distance from region B_1 and B_2 to their related parts will be shorter. Start bidirectional layout earlier for the last 5%, guarantees a more balanced layout.

4. After the first N size of hottest code is laid out, we calculate the following values:

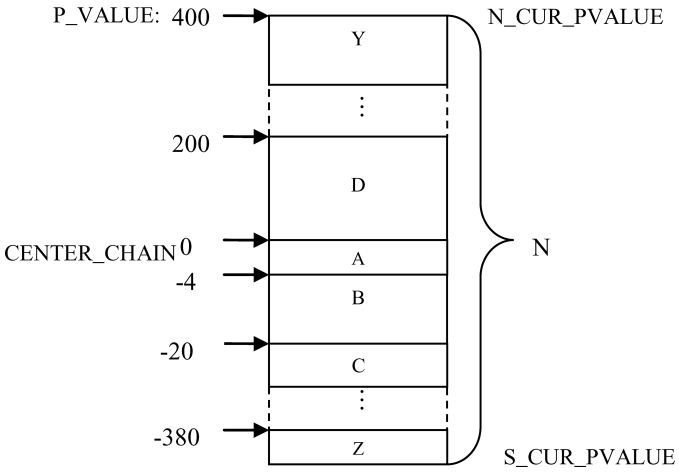

Fig. 11. The first N size of reordered code.

(1) CENTER_CHAIN
The chain residing in the middle point of the limit size N. Chain A is the CENTER_CHAIN (see Figure 11). CENTER_CHAIN is used to define the relative positions of the other chains in the bidirectional layout.

(2) P_VALUE
The signed position value of each chain is defined as the distance from the first instruction of the chain to the first instruction of the CENTER_CHAIN. Process upward (north direction) and assign each chain a positive P_VALUE (Chain D's P_VALUE is 200), and downward (south direction) to assign each chain a negative P_VALUE to indicate the direction (Chain B's P_VALUE is -4). The P_VALUE is used for measuring the distance between chains in the later layout and help to measure cache conflicts between chains.

(3) N_CUR_PVALUE and S_CUR_PVALUE

$$\text{N\_CUR\_PVALUE} = \text{sum of the size of chain Xi, (where Xi's P\_VALUE} > 0) \quad (2)$$

$$\text{S\_CUR\_PVALUE} = -\text{sum of the size of chain Xi, (where Xi's P\_VALUE} <= 0) \quad (3)$$

As shown in Formulas (2) and (3), N_CUR_PVALUE is the total size of all the positive position chains, while S_CUR_PVALUE is the total size of all the nonpositive position chains. These two values are used to determine which side is shorter in length. When processing a chain with no relation or conflict with any previous chains, the chain will be laid on the shorter length side to implement a balanced layout.

5. Select the next chain X from the sorted chains. There are two cases:
   (1) Chain X has no relation with any chain processed. Compare N_CUR_PVALUE and S_CUR_PVALUE and put it on the side with smaller size.

(2) The chain X has n (n > 0) related chains, i.e., with chain X_1, X_2, X_3 ....X_{n-1} and X_n. Calculating the following distances.

$$N\_DISTANCE_i = (N\_CUR\_PVALUE - P\_VALUE \text{ of } X_i), (i = 1, 2 \ldots .n) \quad (4)$$

$$S\_DISTANCE_i = (P\_VALUE \text{ of } X_i - S\_CUR\_PVALUE), (i = 1, 2 \ldots .n) \quad (5)$$

$$N\_DISTANCE = \text{sum of } N\_DISTANCE_i, (i = 1, 2 \ldots .n) \quad (6)$$

$$S\_DISTANCE = \text{sum of } S\_DISTANCE_i, (i = 1, 2 \ldots .n) \quad (7)$$

These values indicate the distance between X and its related chains. They are used for determination of which side chain X should be laid.

(3) If chain X and its related chain X_i both have execution frequencies larger than zero, determine if they will conflict when mapped in cache. Considering two related chains X and X_i, the measuring process proceeds as follows:

a. Calculate the memory positions of the chains
Assume P_VALUE for X and X_i are P_{first} and P_{ifirst}. They are the positions of the first instruction of chain X and X_i. With the chain size, we can get the positions of the last instruction of chain X and X_i as P_{last} and P_{ilast}. These values are based on the position of the CENTER_CHAIN, so they are relative memory positions not absolute ones. However, they are sufficient for cache conflict analysis.

b. Calculate the cache positions of the chains
Here we assume that we are dealing with a direct-mapped cache with size M.

As the P_{first}, P_{last}, P_{ifirst}, and P_{ilast} can be negative, for the sake of convenience, we add a large value V onto the negative position value to change it to positive. As for later mod calculation, V should be a multiple of M. If the result is still negative, repeat adding V onto it until it becomes positive. Thus, the calculation of the positive value P'_{ifirst} is as Formula (8). If the P_{ifirst} is positive, the value of k is 0, otherwise k is greater than 0. In our implementation, we set V 0x10000000, which is large enough for all the SPEC benchmarks. The calculations of P'_{last}, P'_{ifirst}, and $P'_{ilast}$are done in the same way.

$$P'_{1first} = P_{ifirst} + kV \ (k >= 0, V \bmod M = 0) \quad (8)$$

With all the positive values of the chains P'_{first}, P'_{last}, P'_{ifirst}, and $P'_{ilast,}$We calculate the cache positions C_{first}, C_{last}, C_{ifirst}, and C_{ilast} of the chains by mod the corresponding P values with cache size M as shown in Formula (9). Again these positions are also relative values simply used for conflict analysis.

$$C_{first} = P'_{first} \bmod M, \quad C_{last} = P'_{last} \bmod M$$
$$C_{ifirst} = P'_{ifirst} \bmod M, \quad C_{ilast} = P'_{ilast} \bmod M \quad (9)$$

c. Calculate the conflicting area
If either of the following conditions is satisfied, the two chains are in conflict.

$$C_{first} <= C_{ifirst} <= C_{last} \quad (10)$$

$$C_{ifirst} <= C_{first} <= C_{ilast} \quad (11)$$

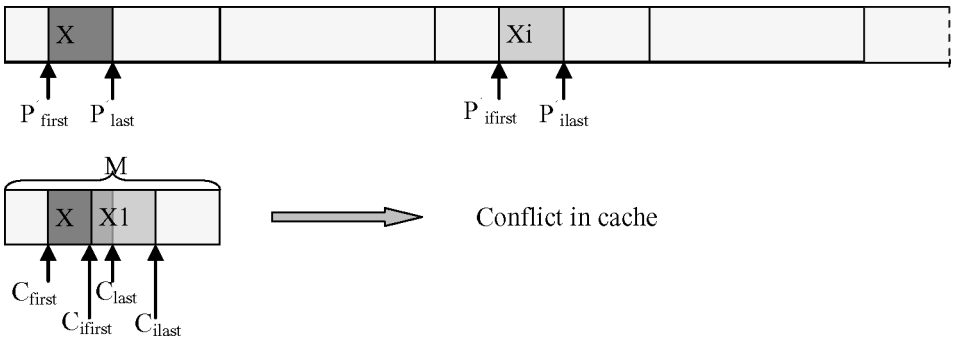

Fig. 12. Cache conflict analysis.

For example, in Figure 12, the first condition is satisfied and the two chains will be in conflict when mapped in the cache.

(4) If the two chains conflict, choose the side as follows,

```
/* chain X or all related have never been executed, no cache conflict will occur */
if (chain X or all Xi have zero execution frequency based on profile)
        compare N_DISTANCE and S_DISTANCE, choose the smaller side;
else if {
        if (both sides are cache conflict free)
                choose the smaller DISTANCE side;
        else if (only one side indicates cache conflict)
                choose the other side;
        /*both sides indicate cache conflict*/
        else
                choose the smaller DISTANCE side;
        }
```

As stated in the pseudocode above, the major goal is to keep a balanced layout. Furthermore, since we lay the chains bidirectionally, we have a greater chance to avoid cache conflict compared with the original PH algorithm. When one side indicates cache conflict, we can choose the other side. When both sides indicate cache conflict, we would choose the smaller DISTANCE side. We consider distance as the major issue because the cache conflict detected at this stage will not be as harmful as inserting rarely executed instructions into the relatively hot area.

Finally, update the N_CUR_PVALUE, S_CUR_PVALUE, and assign the P_VALUE of the new incorporated chain.

6. Repeat step 5 until all the chains have been processed.

4.4 BCL Performance Evaluation

4.4.1 *Evaluation Environment Setup.* To evaluate the BCL algorithm, we build the evaluation environment as follows. First, all the benchmarks are separately reordered in three ways: the PH with direct jumping method, the PH

with distant jumping method, and BCL algorithm. The benchmarks are then run with reference input five times on the real machine 500 MHz Godson 2C processor (64 K Icache, 64 K Dcache on chip, 8 M off-chip secondary cache, 512 M memory). We run the programs five times and use the median value as the result. The test environment for ideal branch offset support is built on Sim-Godson simulator, which is a simulator derived from simplescalar [Burger and Austin 1997].

Because our focus is the exceeded branches, we want to see how much benefit our algorithm can bring compared with the results if we increase the branch offset to tolerate reordering from the ISA point of view. However as the MIPS-like ISA is 32-bit fixed-length, we cannot simply extend the branch offset to be 21 bits. Experimental results show that, on average, 90.8% (static count) / 92.3% (dynamic count) exceeded branches are those comparing the value of one register with zero. Thus, we just use the available instruction format: 6 bits for opcode, 5 bits for one register, and the left 21 bits for the offset. After that is done, 7.2% (static count) / 6.3% (dynamic count) of previous exceeded branches go out of bound. As the ratio suggests, fewer branches exceed the limit ($7.2\% < 1 - 90.8\%, 6.3\% < 1 - 92.3\%$), that is, because some of the previous exceeded branches are caused by the insertion of the jump instructions.

The remaining exceeded branches are treated with distant jumping method. Because of its small percentage in program execution time, we assume that this portion will not make a great difference. The 7.2% exceeded branches will bring about 309 (2148 * 7.2% * 2) instructions. The size of these instructions is about 1.2 KB. When calculating the ideal benchmark size, we have eliminated this part from the resulting benchmarks. In this way, the size of the program is ideal. We also modify the simulator to support these 21-bit offset branch instructions and then run the programs on the simulator to see the effect of 21-bit branch offset.

The algorithm was evaluated through various measurements: number of exceeded branches, code size, instruction cache miss rate, average length of continuous instructions before control-flow transfer, and program execution time on real machine.

4.4.2 *Experimental Results*

4.4.2.1 *Number of Exceeded Branches.* Table I shows that the number of exceeded branches is greatly reduced. For the small size programs, such as, gzip, mcf, and bzip2, all exceeded branches are eliminated by using the BCL algorithm. For larger programs, such as, vpr, crafty, parser, vortex and twolf, more than 50% of exceeding branches are eliminated.

The reduced branches for certain programs, such as, gcc and perlbmk, are relatively low. There are two reasons. First, these programs have more hot spots interleaved with cold spots. Second, these programs do not have such obvious spatial locality like other programs. The execution spots are scattered all over the program. More chains have to be laid first before the cold chains are processed. The distances between some of the hot and cold spots are too far away for BCL to make a significant difference.

Table I. Number of Exceeded Branches with the PH and BCL Algorithm

Spec2000	Number of Exceeded Branches using PH	Number of Exceeded Branches using BCL	Reduction Rate (%)
164.gzip	64	0	100.0
175.vpr	965	396	59.0
176.gcc	9704	5901	39.0
181.mcf	295	0	100.0
186.crafty	428	207	51.6
197.parser	222	39	82.4
252.eon	1420	783	44.9
253.perlbmk	4097	2541	38.9
254.gap	1426	732	48.7
255.vortex	5882	1802	69.4
256.bzip2	62	0	100.0
300.twolf	1217	476	60.9
Average	2148.5	1073.1	50.1

4.4.2.2 *Code Size.* Code size is not very important for systems with large memory, but very meaningful for those with limited memory, like embedded systems. For very large programs, which suffer more from insufficient cache capacity (capacity miss) than from insufficient associativity (conflict miss), the reduction of code size is also meaningful.

The code size is affected by the reordering algorithm. If any branch exceeds the limit, long jumps have to be inserted to ensure code correctness. In MIPS-like ISA, each branch is followed by a delayslot (since we want to run the benchmarks on a real machine, all the benchmarks support delayslots); thus, inserting one jump actually introduces two instructions. The insertion of jumps may again cause some of the branches to go beyond limit, which, in turn, introduces more jumps. Experimental results show that the BCL algorithm reduces the code size by 1.6% (14.2 KB), on average, when compared to the original PH algorithm with direct jumping method and by 0.9% (7.8 KB) when compared to the PH algorithm with distant jumping method.

To compare the code size in detail, we measure the original code size, using the four following methods: the PH algorithm with direct jumping method, the PH algorithm with distant jumping method, BCL algorithm, and the PH algorithm with ideal branch offset support. As we can see from Figure 13, the code size is increased even using ideal branch offset. It is because many cases are like Figure 1, namely, code reordering twists the fall-through block B and the target block C of the branch block A, at the same time block C is the successive block of B, the twist detaches C from B, thus a jump is required to maintain the correctness. The code size of programs using BCL algorithm is less than the average code size using ideal branch offset and the PH algorithm with distant jumping method, except for very large programs with little spatial locality.

4.4.2.3 *Icache Miss Rate.* The introduction of additional jumps not only affects the code size, but also the instruction cache miss rate. Especially for large programs, the executed instructions occupy a huge space, as the inserted

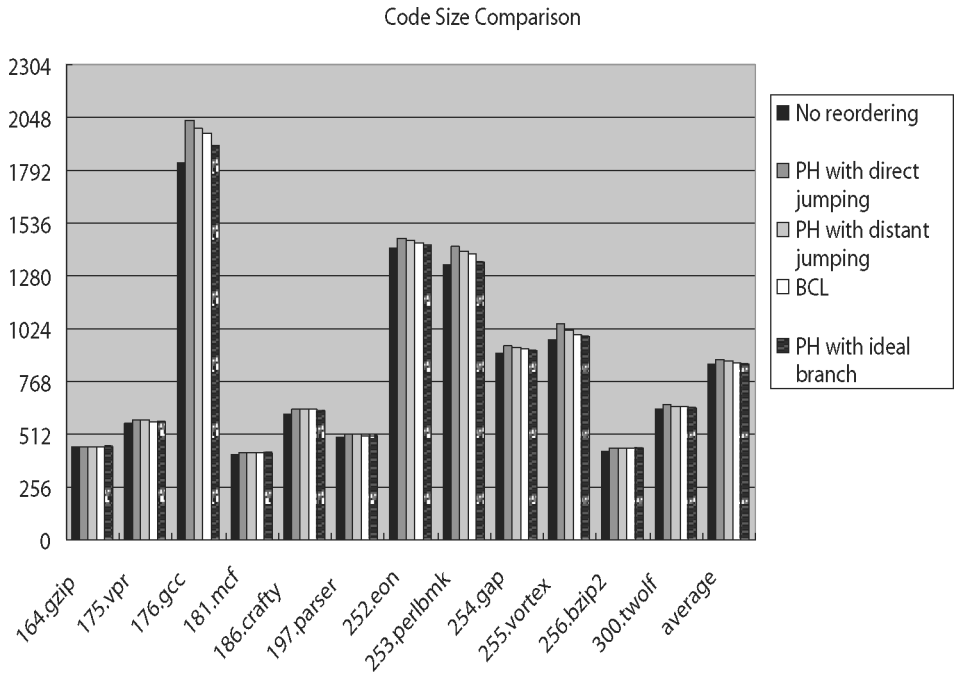

Fig. 13. Code size (KB) comparison.

jumps have to be placed within the reach of the branch, and many jumps will be put in a relatively hot area. As most of these jumps will never be executed, they pollute the cache area and force the executed instructions to be moved far away. As a result, the elimination of exceeded branches reduces the cache miss rate.

Besides the effect of eliminating additional jumps, the chain layout strategy we used in BCL algorithm also reduces the icache miss rate. In the original method applied by the PH algorithm, each chain can only relocate in one direction. When we view the code space as bidirectional, each chain has two possible positions to make a better choice. Thus, we create more opportunities for one chain to avoid cache conflict with its related chains.

Unlike the previous measurement, the icache miss rate is influenced by the cache configuration. We explore the effect of the algorithm on different cache size and cache line size.

Figures 14 and 15 show the impact of different reordering strategies with various cache sizes and cache line sizes. When exploring the effect of cache size, we assume the cache line to be 32 bytes and when measuring the effect of cache line size, we make the assumption that the cache is 32 KB large.

The results in Figures 14 and 15 show that even the PH with direct jumping method reduces the cache miss rate. The use of distant jumping method further reduces the miss rate. The BCL reordering algorithm surpasses the PH algorithm and is close to the ideal branch offset support. The elimination of cache polluting instructions shows the largest improvement for small cache size with big line size.

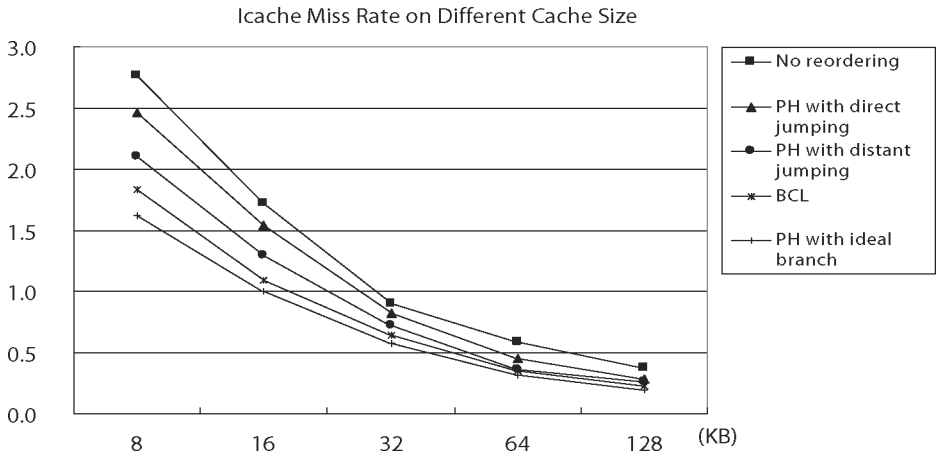

Fig. 14. Icache miss rate (icache misses per k instructions) on different cache size.

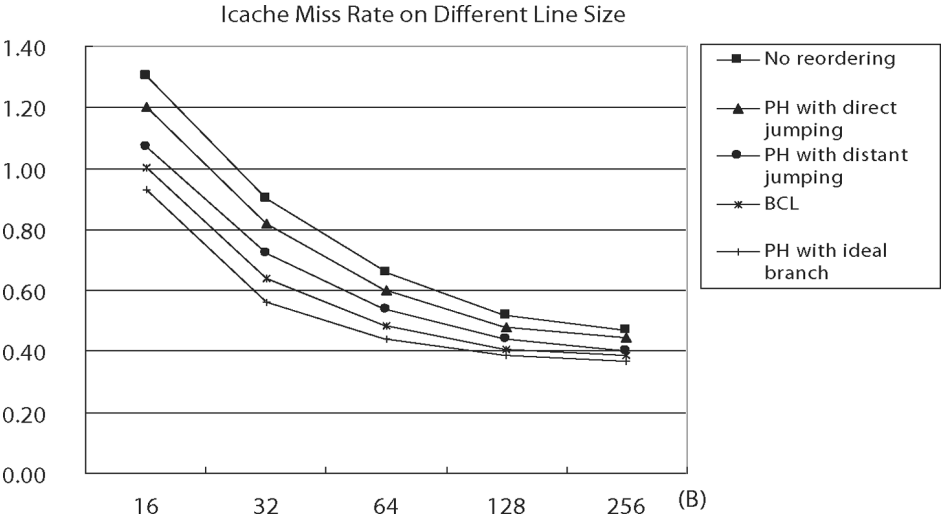

Fig. 15. Icache miss rate (icache misses per k instructions) on different cache line size.

4.4.2.4 *Average Length of Continuous Instructions* . In Figure 16, we examine the instructions per branch (IPB) of the original and reordered programs. This measurement is meaningful for fetch efficiency and efficient issue width. We have not examined perlbmk because it forks another process and cannot be simulated on a single-thread simulator. It is the same for all the simulation.

As the result shows, the direct jumping method decreases the IPB by 3.1%, on average, with the largest drops in gcc, eon, vortex, and twolf. This is because of the jump fall-through instructions inserted between the branch block and its fall-through block (see Figure 5). These inserted jump instructions frequently interrupt the control flows.

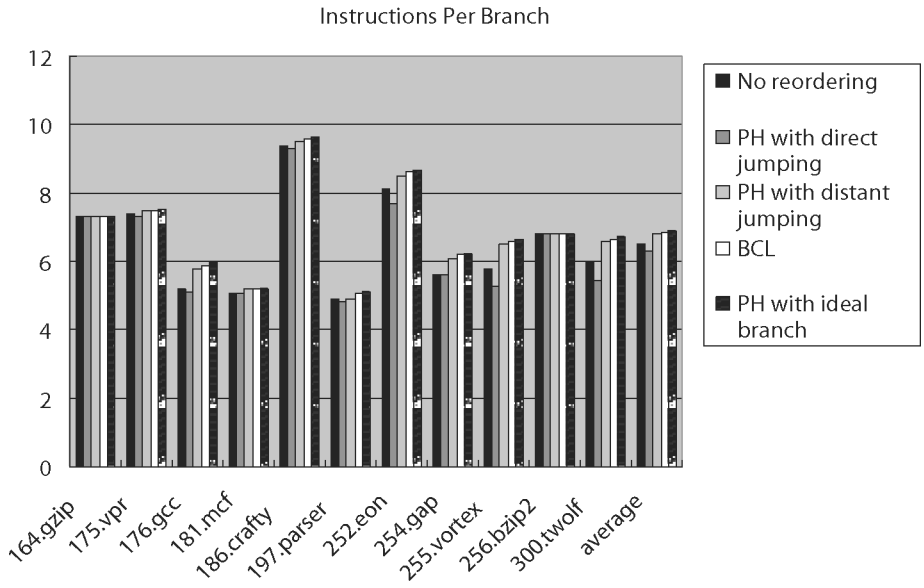

Fig. 16. Instructions per branch.

The BCL algorithm brings limited improvement compared with the distant jumping method. The improved programs are gcc, crafty, parser, eon, gap, vortex, and twolf. Other programs show no improvement. As for ideal branch, gcc and eon can further benefit from it. Because of the large size and little spatial locality, ideal branch is needed for these programs to eliminate all exceeded branches, so as to increase the IPB. The remaining programs react more or less the same to both methods.

Overall, the BCL algorithm increases the IPB by 0.9, 5.1, and 7.2%, respectively, compared to the PH with distant algorithm, original program and the PH with direct jumping. For large programs with little spatial locality, the ideal branch will bring some benefit.

4.4.2.5 *Program Execution Time.* We compare the spec ratio (representing the execution time) of the original benchmark, the benchmark using the PH algorithm with direct jumping method, using the PH with distant jumping method, using BCL algorithm, and using the PH with ideal branch offset support. We use the same 21-bit offset as for Alpha ISA. The spec ratio of ideal branch offset is calculated by multiplying the spec ratio using BCL with the IPC ratio between ideal branch support and the BCL algorithm.

As we can see from Figure 17, the BCL algorithm improves the performance by 4.0%, on average, when compared to nonreordered cases. Even using direct jumping, the spec ratio is 1.6% higher than nonreordered, which proves the effectiveness of code reordering. When using the distant jumping method, the spec ratio is 2.7% higher than the nonreordered case. The additional 1.1% (2.7%–1.6%) is brought by moving additional jumps to cold areas. BCL further improves the ratio by 1.3% by removing additional jumps, decreasing the icache

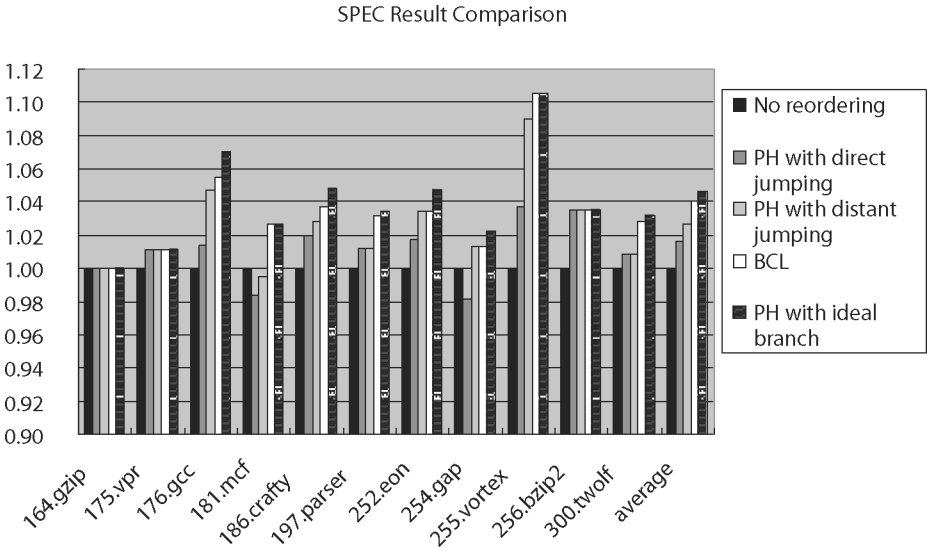

Fig. 17. Spec result comparison with reference input (normalized to no reordering result).

miss rate, and increasing the IPB. The biggest improvement for BCL compared to distant jumping is achieved on gcc, mcf, vortex, and twolf.

For ideal branch offset, several programs will be further improved, such as gcc, crafty, parser, eon, gap, and twolf. Since the average improvement is 0.6%, the BCL algorithm is already achieving a satisfactory effect compared to ideal branch offset.

4.4.3 *Evaluation Result Analysis.* As we can see from the above experimental results, our BCL algorithm manages to surpass the previous PH algorithm by striking a balance on the following four issues:

1. Distance of related code. This is the most remarkable improvement we have made compared to any previous code layout algorithm. With the cold portion laid out both forward and backward according to the position of its related code, the distance between the related parts is decreased.

   Besides laying out the chains, when sorting the chains, we also take the distance of related chains into consideration. By adjusting the order of similar frequent chains, the chains related with hot chains will be put nearer to hot chains, and those independent chains will be pushed far away. In this way the number of exceeded branches will be minimized. Because the algorithm only changes the position of the nonhot codes and keeps the hottest codes together, this method will not decrease performance.
2. Icache miss rate. When laying out the code, the cache miss rate is also taken into consideration. The first N size of chains, which is the hottest part of the program, is laid together so that conflicts among these chains are eliminated. Later chains have two potential positions to relocate. Thus, there are more opportunities to reduce the cache conflict misses between the chain and its related chains.

3. Code size. Unlike previous reordering research [Parameswaran and Henkel 2001, 2004], our approach does not increase the memory size to reduce the cache conflict misses. We borrow the closest-is-best idea from the PH algorithm, ensuring that the code size will not increase too much. Considering the limited branch offset, we propose the BCL algorithm to further reduce the number of exceeded branches so as to keep the code size from increasing when reordering.
4. Control transfer of the code. As the instruction cache becomes larger and larger, the cache miss rate becomes more tolerable, while control-flow transfer has received more interest than before [Ramirez et al. 2005]. In BCL algorithm, we pay special attention to reducing control-flow transfers. The direct jumping method adds control-flow transfer to the original program. The distant jumping method is proposed to overcome this problem. The BCL algorithm further reduces the number of exceeded branches, which leads to a reduction in inserted jumps and helps to reduce additional control-flow transfers.

5. CONCLUSIONS

Previous code reordering algorithms mainly focused on reducing instruction cache miss rate and later on reducing control-flow transfers in the context of multi-issue processors. Yet for larger benchmarks and system applications, such as OSes and databases, applying reordering algorithm might cause the branch offset to exceed the limit on some instruction sets, such as MIPS.

This paper explores two simple ways (direct jumping and distant jumping) to handle the exceeded branches and proposes the bidirectional code layout (BCL) algorithm to minimize the number of exceeded branches. The algorithm carefully keeps a balance on four main issues: the distance of related blocks, the instruction cache miss rate, the memory size required, and the control-flow transfer of the code. Results show that the BCL can effectively reduce the number of exceeded branches by 50.1%, on average. For programs exhibiting high spatial locality, the reduction can reach up to 80–100%. BCL improves program spatial locality, decreases cache miss rate, control-flow transfer, and execution time of the program. When compared with the ideal case, experimentation shows that, except for some large programs, BCL can achieve ideal performance.

ACKNOWLEDGMENTS

We thank Brad Calder, Dean Tullsen, Tom Conte, and the anonymous referees for their detailed feedback. Many thanks to the University of Arizona for their work on alto. We also thank our laboratory members, Yan Tang, Qiong Zou, and Xiaojing Zhu, without your help, it would have been impossible for us to complete this work in such a short period of time. This research is supported by the National Science Foundation of China (60325205), 863 Hi-Tech Research and Development Program of China (2002AA110010) and Knowledge Innovation Engineering Project of Chinese Academy of Sciences (KGCX2-109).

REFERENCES

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, British Columbia (June 18–21). 1–12.

BURGER, D. AND AUSTIN, T. M. 1997. The simpleScalar tool set, version 2.0. In *Computer Architecture News*, (June), 13–25.

CHEN, Y., ZHU, X. J., ZOU, Q., AND LIU, L. 2006. GLTO design and analysis. *Journal of Computer Research and Development 43*, 8 (Aug.), 1450–1456.

COHN, R. AND LOWNEY, P. G. 2000. Design and analysis of profile-based optimization in Compaq's compilation tools for Alpha. *Journal of Instruction Level Parallelism 2*, 5 (May), 1–25.

COHN, R., GOODWIN, P., LOWNEY, G., AND RUBIN, N. 1997. Spike: An optimizer for Alpha/NT executables. In *USENIX Windows NT Workshop*, August.

GLOY, N. 1998. Code Placement using Temporal Profile Information. PhD thesis, Harvard University, Cambridge, MA.

HASHEMI, A. H., KAELI, D. R., AND CALDER, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, Las Vegas, Nevada (June). 171–182.

HWU, W. W., ZHANG, F. X., ET AL. 2004. Godson processor project data. Beijing: Institute of computing technology, Chinese Academy of Sciences, Beijing.

HWU W. W., ZHANG, F. X., AND LI, Z. S. 2005. Microarchitecture of the Godson-2 processor. *Journal of Computer Science and Technology 20*, 3 (Mar.), 243–249.

KALAMATIANOS, J. AND KAELI, D. R. 1998. Temporal-based procedure reordering for improved instruction cache performance. In *Proceedings of the 4th International Conference on High Performance Computer Architecture*, Las Vegas (Feb.). 244–253.

LU, J. W., CHEN, H., YEW, P. C., AND HSU, W. C. 2004. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism 6*, 4 (Apr.), 332–341.

MCFARLING, S. 1989. Program optimization for instruction caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA (Apr.). 183–191.

MUTH, R., DEBRAY, S., AND WATTERSON, S. 2001. Alto: A link-time optimizer for the Compaq Alpha. *Software Practice and Experience 31*, 1 (Jan.), 67–101.

PARAMESWARAN, S. AND HENKEL, J. 2001. I-CoPES: Fast instruction code placement for embedded systems to improve performance and energy efficiency. In *Proceedings of International Conference on Computer Aided Design*, San Jose, CA (Nov.). 635–641.

PARAMESWARAN, S. AND HENKEL, J. 2004. REMcode: Relocating embedded code for improving system efficiency. *IEE Proc.-Comput. Digit. Tech. 151*, 11 (Nov.), 431–435.

PETTIS, K. AND HANSEN, R. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY (Jun.). 16–27.

RAMIREZ, J., PEY, L., AND VALERO, M. 2005. Software trace cache. *IEEE Transactions on Computers 54*, 1 (Jan.), 22–35.

Received August 2005; revised January 2006 and June 2006; accepted November 2006