PPCREF Project - High level design

Software Pipelining

Branch Optimizations

IBM Israel - Science and Technology
Matam, Haifa 31905, Israel

Software Pipelining for gcc

1. INTRODUCTION

Software pipelining is a scheduling technique which attempts to overlap the execution of distinct loop iterations, by creating a new loop body with instructions taken from several consecutive iterations.

First, loops which are candidates for pipelining are identified. As before, the body of such a loop is scheduled by the inter-block scheduler. Until now, only *local motions*, where the instructions scheduled belonged to the body of the loop (i.e the first iteration) were done. Software pipelining enables the scheduler to do *pipelined motions* where the instructions scheduled may belong to successive iterations.

In a pipelined motion two instructions are moved:

- an instruction is brought into the body of the loop from the k'th ($k > 1$) iteration (this is the instruction scheduled).
- its corresponding copy from the loop, initially belonging to the $k - 1$ iteration, is moved to a prologue p_{k-1}, outside the loop.

To ensure the correctness of a pipelined motion, additional conditions are imposed on the control flow of the region and on the data dependencies of its instructions.

After the scheduling, the loop may contain instructions from the first k iterations. In the case when pipelined motions have been performed (i.e. $k > 1$), the non-speculative execution of the instructions moved outside the loop must be assured. This is done by generating appropriate prologues and epilogues for these iterations

Care must be taken to ensure that the global information used by subsequent passes of the compiler is updated to account for changes resulting from the software scheduling.

This document describes and explains the enhancements to the inter-block scheduler of gcc that implement software pipelining. The main additions to the inter-block scheduler are:

- identification of candidate loops for pipelining
- computation of control flow information for pipelined motions
- computation of data dependencies for pipelined motions
- handling of pipelined motions by the list scheduling algorithm
- generation of prologues and epilogues
- updating of global information used by gcc

2. PIPELINED REGIONS

The inter-block scheduler splits a function into *regions* (i.e. inter-block scheduling ranges). A region may be an inner loop, a basic block or the whole procedure.

Only a region which is an inner loop is a candidate for pipelining. Such a loop is pipelined if it has a single latch and the number of its iterations is controlled by a BCT (branch on count) instruction.

In addition, in order to not adversely affect the compilation time, only a loop having the number of instructions under a given threshold (by default 50 instructions) is pipelined.

3. CONTROL FLOW INFORMATION

The support for inter-block motions in the original inter-block scheduler is briefly described. A candidate for scheduling may be moved from a block only into one of its dominators. In order to assure the correctness of a move, *split blocks* where live information is checked are computed. *Update blocks* where the live information is updated after a move are also computed. For non-speculative moves, the lists of split and update blocks are empty. Associated with a move is its *probability*, used to compare it with other moves.

For a pipelined motion, the block containing the copy of the candidate for scheduling, is the *source block*. The *target block* is the block being currently scheduled. Other blocks referred to below are the *header block*, the header of the loop, and the *latch block*, the latch of the loop.

From the point of view of the control flow, a pipelined motion is equivalent to two instruction moves, one from the source block to the header block and the second from the latch block to the target block. The conditions imposed by the inter-block scheduler on the control flow graph for these two moves are:

- the header block must dominate the source block (as the header of a loop dominates all the blocks in the loop, this condition is always fulfilled).
- the target block must dominate the latch block.
- the live information on the split blocks of the source block relative to the header block must allow the move of the instruction from one to another.
- the live information on the split blocks of the latch block relative to the target block must allow the move of the instruction from one to another.

The probability of a pipelined motion is computed by multiplying the probabilities of the two moves. Similarly, its update blocks are the union of the update blocks of the two moves.

Example: The loop in Figure 1 contains the blocks *b1*, *b2*, *b3*, and *b4*. The block *b1* is the header block, and the block *b4* is the latch block.

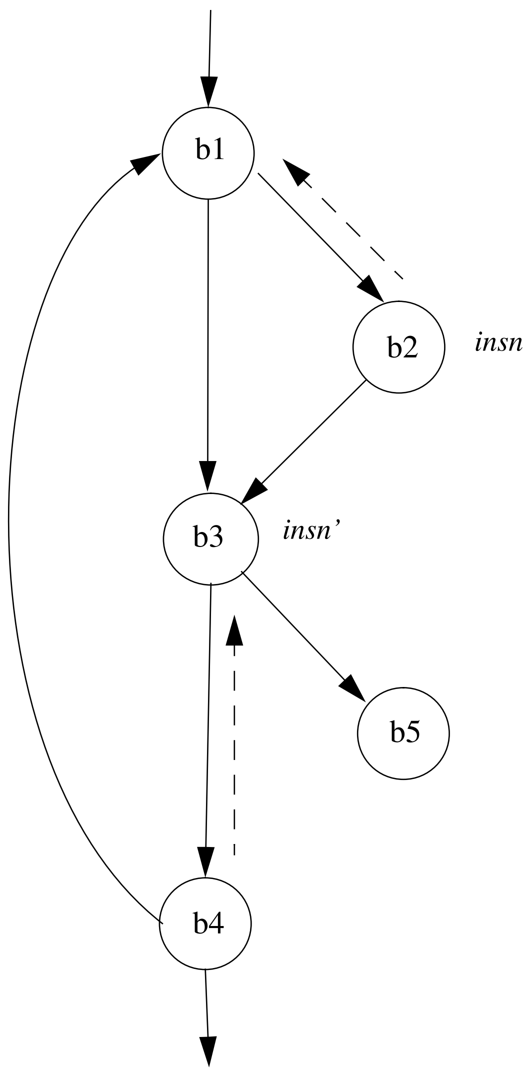

Figure 1

Let *insn* be an instruction in the block *b2*, whose copy from the second iteration *insn'* becomes a candidate for scheduling during the scheduling of block *b3*. Block *b3* then becomes the target block and block *b2* the source block. We have:

dominate (target, latch) = dominate (b3, b4) = true

split_blocks (header,source) = split_blocks (b1, b2) = {b3}

split_blocks (target,latch) = split_blocks (b3,b4) = {b5)

update_blocks (header, source) = update_blocks (b1, b2) = {b2}

update_blocks (target, latch) = update_blocks (b3,b4) = {b4}

As the block *b3* dominates the block *b4*, this pipelined motion is possible when the registers defined in the instruction *insn* are not live at the beginning of the blocks *b3* and *b5*. We assume that this conditions is fulfilled and the instruction *insn'* is scheduled in block b3. The instruction *insn* is moved from the block *b2* in a prologue. After the move the live information is updated at the beginning of the blocks *b2* and *b4*.

We assume further that during the scheduling of the block *b4*, the instruction *insn"* (the copy of the instruction *insn* in the third iteration) become a candidate for scheduling. Now the block *b4* is the target block and the block *b3* is the source block. In this case:

dominate (target, latch) = dominate (b4, b4) = true

split_blocks (header,source) = split_blocks (b1, b3) = empty

split_blocks (target,latch) = split_blocks (b4,b4) = empty

update_blocks (header, source) = update_blocks (b1, b3) = empty

update_blocks (target, latch) = update_blocks (b4,b4) = empty.

All conditions required for this pipeline motion are fulfilled. On completion of the motion, the list of update blocks is empty and hence there is no need to update the live information.

As we have seen, the analysis needed for pipelined motions is based on the analysis performed on inter-block motions. The problem which we still need to resolve is that the original scheduler has only information (about the domination, probability, split-path and update-path) that is relative to the current target. It can only handle moves into the current target. To overcome this limitation, we must maintain similar information relative to the header of the loop. This information needs to be computed only once, as it does not depend on the current target being scheduled. In fact, the original scheduler computes this information when the current target is the header block, but subsequently overwrites it. Now, if the pipelining is performed, this information is preserved in a separate data structure.

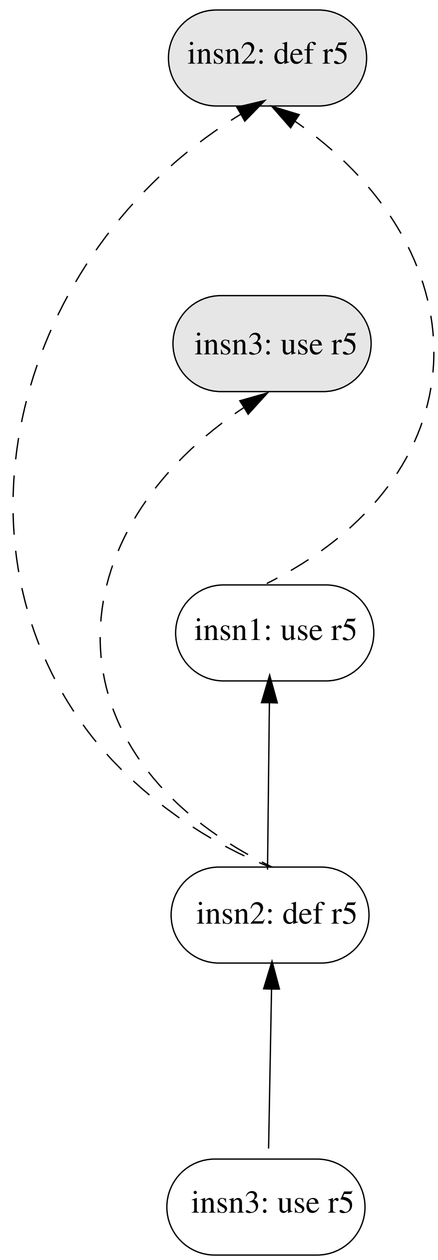

Figure 2

4. DATA DEPENDENCIES INFORMATION

In order to handle the data dependencies imposed by a pipelined motion, we now compute two types of forward dependencies for an instruction:

- *real dependencies* which are the dependencies within the same iteration (these dependencies are those computed by the original inter-block scheduler);

- *SP dependencies* which are the dependencies to the next iteration (i.e. inter-iteration dependencies);

and two types of counters:

- a *real counter* which counts the number of dependencies which must be resolved in order to move an instruction in the same iteration (these counters are computed by the original inter-block scheduler).
- an *SP counter* which counts the number of dependencies which must be resolved in order to move the instruction in the previous iteration (this counter is the sum of the real and SP dependencies having the instruction as target).

We need to extend the mechanism for computing data dependencies between instructions in the same iteration (i.e real dependencies) in order to also compute the dependencies between instructions in different iterations (i.e SP dependencies). After describing the computation of data dependencies in the original scheduler, a method for computing the inter-iteration dependencies is presented.

The original inter-block scheduler first computes the backward dependencies. For simplicity the process is described only for dependencies introduced by registers. During a forward traversal, lists for pending uses and pending definitions are maintained for each register. For each instruction seen, its dependencies are updated according to the contents of the two lists. The lists are also updated. This process is detailed below.

For a register *r*, its list of pending uses is denoted by *reg_last_uses(r)* and its list of pending definitions is denoted by *reg_last_defs(r)*. Let *insn* be the current instruction being analyzed. A register *r1*, defined in *insn*, introduces dependencies between *insn* and the instructions from the *reg_last_uses(r1)* and *reg_last_defs(r1)*. All the entries from *reg_last_defs(r1)* and *reg_last_uses* (*r1*) are then discarded, and *insn* is inserted in the list *reg_last_defs(r1)*. A register *r2,* used in *insn,* introduces dependencies between *insn* and the instructions in *reg_last_defs(r2)*. Finally *insn* is inserted in the list *reg_last_uses(r2*).

A straightforward way to compute the inter-iterations dependencies is to traverse the loop twice and to apply the usual procedure for computing the data dependencies. Conceptually, we may consider this as a traversal of the loop unrolled twice. The pending lists at the beginning of the second traversal are exactly the pending lists at the end of the first traversal (while at the beginning of the first traversal these lists were empty). This change in the initialization of the pending lists introduces additional data dependencies between instructions which are exactly the inter-iterations dependencies.

Example: Figure 2 illustrates a simple example of the process described above. The loop is formed from a single block (which is the header and the latch of the loop). Besides the BCT instruction (not shown in the figure) there are three instructions *insn1*, *insn2*, and *insn3*. The first and the third instruction use the register *r5*, the second defines it.

At the beginning of the first traversal *reg_last_defs*(*r5*)=*empty* and *reg_last_uses*(*r5*) = *empty.* At the end of this traversal, the dependencies, drawn with continuous lines in Figure 2, are found.

At the end of the first traversal, *reg_last_defs*(*r5*)={*insn2*} and r*eg_last_uses*(*r5*)={*insn3*}. These lists are used at the beginning of the second traversal. During this traversal, new dependencies, drawn with dashed lines in Figure 2(b), are found. These are the inter-iterations dependencies.

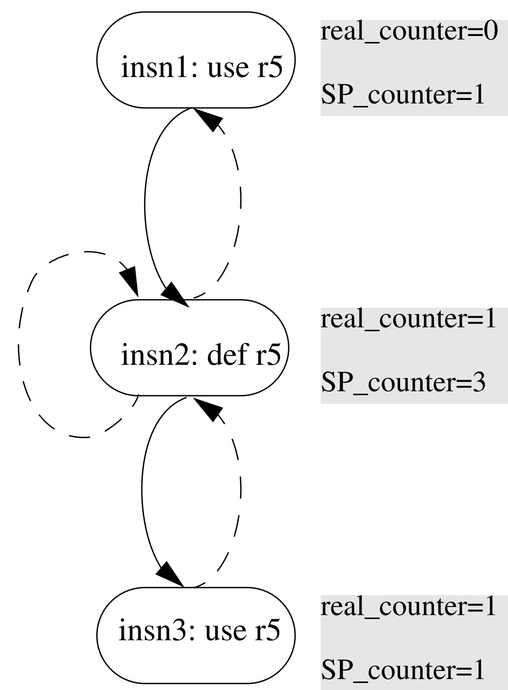

Figure 3

The final data dependency graph with forward dependencies, and the values of the two counters, appears in Figure 3. The SP dependencies are drawn with dashed lines.

5. LIST SCHEDULING

In a pipelined motion, an instruction is brought from the k iteration ($k > 1$), and its corresponding copy from the loop (from the $k-1$ iteration) is moved to a prologue p_{k-1}. The prologue holds all the instructions from the $k-1$ iteration which were moved outside the body of the loop. In practice, a copy of the instruction scheduled is created in the prologue and the original instruction is moved to its new location in the body of the loop. This means that during the scheduling, the original instruction from the body of the loop is used to represent all its copies from successive iterations.

An iteration counter *insn_iter,* specifies which copy it represents (i.e. to which iteration it belongs) for the instruction scheduled. At the beginning of the scheduling, *insn_iter* is zero for each instruction.

As in the original inter-block scheduler, for each instruction, a counter *insn_counter,* is maintained during the scheduling. An instruction may be subject to either a local or pipelined motion when its counter is zero. At the beginning of the scheduling, the counter of the instruction is initialized to the value of its real counter.

After an instruction has been scheduled, the following actions are performed:

- the counter of the instruction is set to the value of its SP counter.
- the counter of an instruction *insn1* pointed to by a real dependency of the instruction scheduled, *insn*, is decreased only if *insn_iter*(*insn1*) - *insn_iter*(*insn*) = 0.
- the counter of an instruction *insn1* pointed to by an SP dependency of the instruction scheduled, *insn*, is decreased only if *insn* = *insn1* or *insn_iter*(*insn1*) - *insn_iter*(*insn*) = 1.
- the iteration counter of the instruction scheduled is incremented.

An instruction may be scheduled only if it satisfies the conditions imposed by the data dependencies (i.e the instruction counter is zero), and the conditions imposed on the control flow of the region (which are dependent of the type of the motion involved). The iteration counter of the instruction chosen to be scheduled is used to differentiate between a local or a pipelined motion: when it is different from zero a pipelined motion will be performed.

Example (Figure 3): A loop is formed from a single block, which is the header block, the latch block, the source block and the target block all rolled into one. The conditions imposed on the control flow graph are always fulfilled. In order to schedule an instruction, we need only check that its counter is zero.

Besides the BCT instruction (not shown) there are three instructions *insn1*, *insn2*, *insn3*. The SP dependencies are drawn with dashed lines. It is assumed that only instructions from the first three iterations may be brought into the body of the loop, thus only two prologues *p1* and *p2* are necessary. The state at the beginning of the scheduling is shown in Figure 4 (a). For each instruction there is a couple (*insn_counter*, *insn_iter*) with information about the instruction.

At the start, the only instruction which may be scheduled (in a local motion) is *insn1*. The changes after its scheduling are insn_counter(insn1)=1, insn_counter(insn2)=0 and insn_iter(insn1) = 1. The loop and the relevant data structures after the motion are shown in Figure 4 (b).

The only instruction which may be scheduled (in a local motion) is *insn2*. The changes after its scheduling are insn_counter(insn1)=0, insn_counter(insn2)=2, insn_counter(insn3)=0 and insn_iter(insn2)=1. The data structures and the loop after the motion are shown in Figure 4 (c).

Now there are two candidates for scheduling, *insn3* and *insn1* (which belongs to the second iteration). We assume that *insn1* is scheduled (in a pipelined motion). The changes after its scheduling are insn_counter(insn1)=1, insn_counter(insn2)=1 and insn_iter(insn1) = 2. A copy of the instruction *insn1* is created in *p1*. The data structures and the loop after the motion are shown in Figure 4 (d).

The only instruction which may be scheduled (in a local motion) is *insn3*. The changes after its scheduling are insn_counter(insn2)=0,insn_counter(insn3)=1 and insn_iter(insn3)=1.The data structures and the loop after the motion are shown in Figure 4 (e).

The only instruction which may be scheduled is *insn2* (which belongs to the second iteration). The changes after its scheduling are insn_counter(insn1)=0, insn_counter(insn2)=2,

```
p1:
p2:
L1:
    insn1   (0, 0)
    insn2   (1, 0)
    insn3   (1, 0)
    BCT L1
```
(a)

```
p1:
p2:
L1:
    insn1   (1, 1)
    insn2   (0, 0)
    insn3   (1, 0)
    BCT L1
```
(b)

```
p1:
p2:
L1:
    insn1   (0, 1)
    insn2   (2, 1)
    insn3   (0, 0)
    BCT L1
```
(c)

```
p1: (insn1)
p2:
L1:
    insn2   (1, 1)
    insn1   (1, 2)
    insn3   (0, 0)
    BCT L1
```
(d)

```
p1: (insn1)
p2:
L1:
    insn2   (0, 1)
    insn1   (1, 2)
    insn3   (1, 1)
    BCT L1
```
(e)

```
p1: (insn1, insn2)
p2:
L1:
    insn1   (0, 2)
    insn3   (0, 1)
    insn2   (2, 2)
    BCT L1
```
(f)

Figure 4

insn_counter(insn3)=0 and insn_iter(insn2)=2. A copy of the instruction *insn2* is created in *p1*. The data structures and the loop after the motion are shown in Figure 4 (f).

Now there two candidates for scheduling *insn3* (which belongs to the second iteration) and *insn1* (which belongs to the third iteration). We assume that *insn1* is scheduled in a pipeline motion. The changes after its scheduling are insn_counter(insn1)=0, insn_counter(insn2)=1 and insn_iter(insn1)=3. A copy of the instruction *insn1* is created in *p2*. The data structures and the loop after the motion are shown in Figure 5 (a).

The computation of priorities must take into account the case where the instructions from the ready list belong to different iterations. New heuristics are introduced:

- between two instructions, the one from a previous iteration is preferred.
- each time an instruction is brought from a successive iteration its critical path (the main heuristic) is divided by 2.

- instructions having critical path less than a given threshold are not scheduled.

These new heuristics ensure that the pipelining process will eventually terminate.

6. PROLOGUES AND EPILOGUES

After the scheduling, the loop may contain instructions from the first k iterations. In the case where pipelined motions have been performed (i.e. $k > 1$), the non-speculative execution of the instructions moved outside the body of the loop must be ensured.

The instructions from an iteration j ($1 \le j < k$) which must be executed before entering the loop are in the prologue p_j generated during the scheduling. An epilogue e_j, containing all the instruction from the iteration j not in p_j, is built. Th epilogues are built during a traversal of the loop: if an instruction belongs to the l iteration ($1 \le l \le k$), copies of it are generated in the epilogues e_i, where $1 \le i \le k - l$. At the end of each prologue p_j, a BCF instruction with the target e_j is inserted to test the counter register which controls the number of iterations of the loop.

Finally the prologues p_1,..., p_{k-1} (in this order) are inserted in the code before the loop, and the epilogues e_{k-1},...., e_1 (in this order) are inserted in the code after the loop.

Example: For the loop and its associated data structures in Figure 5 (a), the prologues *e1* and *e2* are generated. The final code generated appears in Figure 5 (b).

7. GLOBAL INFORMATION

In gcc the information about basic blocks (first instructions, last instructions, registers live at beginning of blocks) is computed once, before the scheduling. This information is used and updated by subsequent passes and must be accurate after the pipelining. How this accuracy is enforced is described below.

The information is kept in arrays of size equal to the number of basic blocks in the program, the information for the block i being found at index i. As the pipelining creates new basic blocks (the prologues and the epilogues), new arrays to keep the information for the basic blocks must be created (the original arrays can not be extended). The information from the old arrays is copied into the new ones. As subsequent code assumes that the code in block i is followed by the code in block $i + 1$ care must be taken to insert the information about the new basic blocks at appropriate places in the new arrays. This is done by using an auxiliary data structure which for each basic block points to the block which must precede it in the code.

After the scheduling of a region, the information about the new basic blocks (first instructions, last instructions, registers live at beginning) is computed. The computation of live information at the beginning of the new blocks (i.e. prologues and epilogues) is explained below. First the live information at the end of a new block is computed using the live information at the beginning of its successors. Then the block is scanned and the live information at its beginning is computed. The order in which the new basic blocks are processed is important. A block must be processed only after its successors. This means that for a pipelined loop, the live information is computed first for its epilogues, then for the loop, and finally for its prologues.

```
p1: (insn1, insn2)
p2: (insn1)
L1:
    insn3  (0, 1)
    insn2  (1, 2)
    insn1  (1, 3)
    BCT L1
e2: (insn3, insn2)
e1: (insn3)
```

(a)

```
    insn1
    insn2
    BCF L2
    insn1
    BCF L3
L1:
    insn3
    insn2
    insn1
    BCT L1
L3:
    insn3
    insn2
L2:
    insn3
```

(b)

Figure 5

Another problem concerns the debugging information, which associates with each instruction a line note. After a copy of an instruction is created, a copy of the line note associated with the instruction is also created for possible use in debugging.

Branch Optimizations in gcc

1. INTRODUCTION

The goal of the project is to implement a branch optimization pass in gcc. The optimizations are mainly oriented to systems based on the PowerPC 604. The transformations to be implemented are:

- gluing
- down code motion
- branch swapping

Gluing and branch swapping have been implemented in the XL family of compilers. These transformations were designed to eliminate delays caused by branches in systems that do not have dynamic branch prediction (RS6000, POWER2 etc.). In such cases, measurements have shown noticeable speedup.

In the case of the PowerPC 604, which has dynamic branch prediction, only the first two transformations are useful. By moving a few non-branch instructions between two close branches, we can

```
            BEFORE:                                AFTER:


            BC  CR, LABEL                          BC  CR, LABEL
            B   OUT                                 INSN1
            .......                                 INSN2
                                                    B   NEW_OUT
                                                    .......
           ........                                 ........
OUT:                                       OUT:
            INSN1                                   INSN1
            INSN2                                   INSN2
            ..........                 NEW_OUT:
            ..........                              ..........
```

Figure 1

better exploit the superscalar features of the CPU. This is the basic idea behind the first two transformations. We simply increase the distance between any two branches. Experiments (on contrived examples) verify the usefulness of these techniques for 604-based systems.

Each of the transformations is applicable to a pair of close branches instructions.

2. GLUING

The gluing transformation is applicable to the case where a conditional branch is close to an unconditional one (a *BC-B* pair). A few instructions from the target of the unconditional branch are copied between the two branches. This transformation is illustrated in Figure 1.

The number of instructions to be copied depends on the number of instructions already existing between the two branches. The optimal number of instructions is determined by the tuning.

```
          BEFORE:                             AFTER:

LOOP:                             LOOP:
          .......                             ........
          .......                             ........
          INSN1                               BC CR, LABEL1
          INSN2                               INSN1
          BC  CR, LABEL                       INSN2
          BCT  LOOP                           BCT LOOP
          ........                            ........
          ........                            ........
                                              B  LABEL
                                  LABEL1:
                                              INSN1
          ........                            INSN2
LABEL:                            LABEL:
```

Figure 2

```
            BEFORE:                                AFTER:


                                                   B  LOOP1

LOOP:                                   LOOP:

            ..........                             BC  CR, LABEL

                                         LOOP1:



            .........                              ........

            BC  CR, LABEL                         BCT LOOP

            BCT  LOOP                              BC  CR, LABEL
```

Figure 3

The transformation should be performed carefully so as not to create a new pair of close branches. A sufficient number of instructions should be left between the target of the unconditional branch and the first branch following it.

3. DOWN CODE MOTION

This transformation is applicable when two conditional branches are in close proximity. It is illustrated below for the case where the first conditional branch is a regular conditional branch (*BC*) and the second is a branch on count register (*BCT*). The code may be improved by moving a few of the instructions preceding the first conditional branch to a location between the conditional branches (i.e. moving them down). The complete transformation is illustrated in Figure 2.

An instruction may be moved down only if there is no chain of data dependencies between it and the first conditional branch of the pair. It should be noted that in the context of gcc, it is easy to determine which instructions are suitable for such motions as the backward data dependencies built during the second pass of scheduling are still available.

It is natural to require the second conditional branch to be a *BCT*. This pair of branches usually occurs at the end of a loop. As a consequence, the code in the most frequent path is improved. The transformation works even if the *BCT* instruction is replaced by an ordinary conditional branch.

Again, the optimal number of instructions to be moved between the two branches is determined by tuning.

4. BRANCH SWAPPING

Although it seems that branch swapping yields no results for systems based on the PowerPC 604, it still useful for POWER and POWER2 systems. This transformation is applied to *BC-BCT* pairs. As in the down code motion transformation, requiring the last of the two branches to be a *BCT*, usually results in such a pair occurring at the end of a loop. Consequently the code on the most frequent path is improved. The transformation is illustrated in Figure 3.