International Journal of Computer and Electrical Engineering, Vol. 2, No. 4, August, 2010
1793-8163

Performance Optimization of Hyperblock Using Predicate Compilation Technique

Sweta[1], Dr. Ranjit Biswas[2] and Prof . J.B.Singh[3], *Member, IACSIT*

Abstract—**Predicate execution provides a large number of opportunities to enhance and expose ILP in the presence of branches. However, as with speculative execution, an aggressive compiler is required to realize most of the performance advantages. Compiler Optimization and transformation techniques focus on eliminating branches from the instruction stream and overlapping the execution of multiple control flow paths using the conditional execution capabilities provided by predication. The compiler support for predicated execution is based on a new structure referred to as the hyperblock. . Hyperblocks are a generalized form of Superblock that takes advantage of both predicated and speculative execution. This paper discusses the hyperblock compilation techniques. The formation procedure of hyperblock is described first. Secondly, the extensions to traditional optimization, instruction scheduling, and register allocation techniques to enable them to work on hyperblocks. The presence of predicates introduces new challenges into the compiler backend to understand the meaning of predicates, take the advantage of the relations among predicates, and perform transformations in the presence of predicates. Finally, a set of four new optimizations designed specially for improving the performance of predicated code.**

Index Terms—**Coalescing, compiler backend ,Predicate compilation,**

I. Introduction

The most basic compiler transformation utilizing predicate execution is if-conversion. The traditional approach applied to if-conversion to entire loop to enable vectorization or modulo scheduling of loops with conditional branches .This could also be extended to handle certain control structure , such as hammocks, in non loop portions of the code . The major problem with this approach is that if conversion is an all or nothing transformation . With the large number of branches and corresponding control flow path present in nonnumeric applications, a more flexible strategy that efficiently supports selective if conversion is required. To support such a flexible method, hyperblock is introduced .

A hyperblock is a collection of connected basic blocks in which control may only enter through the first block, referred to as the entry block. Control flow may leave from any number of blocks in the hyperblock. All control flow between basic blocks in a hyperblocks is removed via if –conversion[1]. The goal of hyperblocks is to intelligently group basic blocks from many different control flow paths into a single manageable block for compiler optimization and scheduling.

Hyperblock are formed using a five step procedure : region identification , loop backedge coalescing, block selection , tail duplication , and if –conversion.

```
        Insect = wordct = charct = token =0 ;
        for(; ;)
        {

A:              if(--(fp)-> cnt < 0)
C:              c=filbuf(fp),
        else
B:              c=*(fp)->ptr ++;
D:              if (c==EOF) break;
E:              charct ++;
F:              if((' '<c) && (c<0177))
                {
                        if (! token)
                        {
K:                              wordct ++ ;
H:                              token ++ ;
                        }
                             continue ;
                        }
G:              if ( c = = '\n')
I:                      linect ++ ;
J:              else if (( c!=' ' ) &&
L:                      (c!='\t')) continue;
M:              token = 0;
                }
```

Figure1.1 Source code for the inner loop of wc

A running example is utilized throughout this section to illustrate hyperblock formation. The example choosen is the inner loop from the benchmark *wc*. The pre-processor of C source code for the loop segment is shown in 1.1. The example firstly contains a loop that accounts for a large fraction of the bench mark execution time. Secondly, the loop has a nontrivial control structure, which presents a challenge to all branch handling strategies[13].

The purpose of *wc* is to count the number of characters,words,and lines in an input file .A character buffer is processed in the loop and re-filled as necessary until the end-of-file marker is encountered .The corresponding assembly code and control flow graph for the loop segment are present in fig 1.2. The control flow graph is augmented with the execution frequencies of each control transfer for the measured run of the program . The basic blocks are consistently identified by letters A through M in both figures. The loop is characterized by small basic blocks and a large percentage of branches. Overall, the loop segment contains 13 basic blocks with a total of 34 instructions. Out of the 34 instructions,14 are branches, 8 conditional, 5 unconditional and 1 subroutine call .

[1]Research Scholar , Shobhit University, s weta_verma @ y ahoo.com
[2]Professor CSE/IT , ITM Gurgaon
[3]Professor CSE/IT, Shobhit University

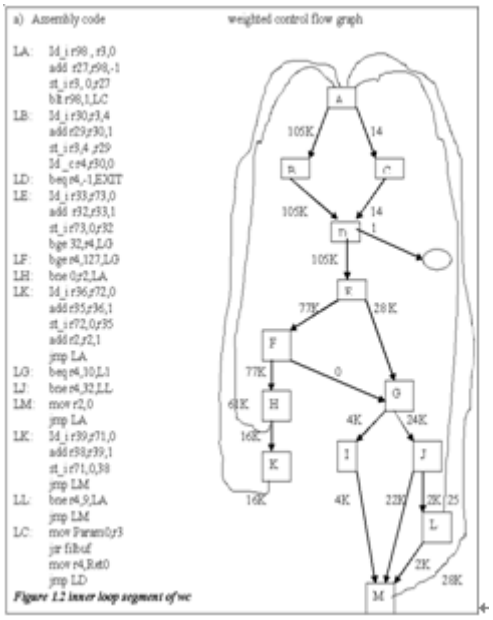


Figure 1.2 inner loop segment of wc

II. Procedure Involved For Hyperblock Formation

Step1-Region Identification : The blocks for a hyperblock are chosen from regions in the control flow graph . A region is a group of basic blocks with a single entry block that dominates all blocks in the region . Typical regions are loop bodies ,intervals, if-then-else conditionals and nested combinations of these.

The region serve as outer boundaries for hyperblock formation. The compiler attempts to identify the largest regions as possible under two constraints. First , a basic block may only resides in a single region. Second , the region may contain no internal cycles. The second constraints is later relaxed to support *loop peeling*. For ***wc*** example, the innermost loop body is identified as a region .

Step 2–Backend coalescing : The second step of hyperblock formation applies only to loop regions All loop backedges of loop regions are coalesced into a single backedge. This is done because if- conversion can only remove non-loop branches. By coalescing all the backedges , the control logic that determines the particular backedge that is traversed becomes a candidate for elimination via if-conversion[1]. A single branch back to the loop header is taken whenever any of the backedges were taken in the original loop.

Examination of the control flow-graph for the *wc* example in fig 1.2 shows that the loop contains four loop backedges, i.e the branches from basic blocks H,K,L and M to basic block A. These branches currently cannot be eliminated with if-conversion . However , these branches can be retargeted to a new block N , as shown in figure 1.3 . Block N simply contains an unconditional branch back to block A. The loop region now contains only a single loop backedge. Additionally , the four branches to block N are now candidates for elimination with if-conversion .

Step 3-Block Selection: The third step of hyperblock formation is choosing a set of basic blocks to combine into a hyper block . Blocks are selected based on two high-level, possibly conflicting goals. First , including more blocks can potentially improve performance by eliminating branches among the included blocks. Second, including too many blocks is likely to result in an overall performance loss due over saturation of processor resources or increased dependence height. These conflicting goals must be addressed by the block selection algorithm .

Blocks are selected by enumerating execution paths through the region .An execution path is a path of control flow from the entry block to an exit block in the region .A priority is calculated for each path to determine its relative importance . Paths are included from highest priority to lowest priority based upon the estimated available execution resources and the characteristics of the path .The final set of selected blocks is then the union of all blocks along paths chosen for inclusion .

The path priority function is a combination of four elements: *path execution frequency , number of instructions on the path , path dependence height , and hazard conditions on the path*. Execution frequency is used to give paths with higher execution frequency a higher priority . In general , execution frequency is used to exclude paths of control which are not often executed . Removing infrequent paths eliminates dependence constraints for optimization and scheduling associated with these paths .Also, the demand for resources is reduced by omitting these paths. The number of instruction along a path is used to give higher priority to paths with fewer instructions . Longer paths utilize more machine resources and are likely to reduce the overall performance of the hyperblock if they are combined with the shorter The dependence height of a path is used to give paths with larger dependence height a lower priority. When multiple paths are merged together in a hyperblock , the dependence height of the resultant hyperblock is the maximum across all paths. Therefore, the overall performance of a hyperblock can be reduced by merging a path with a very large relative dependence height.

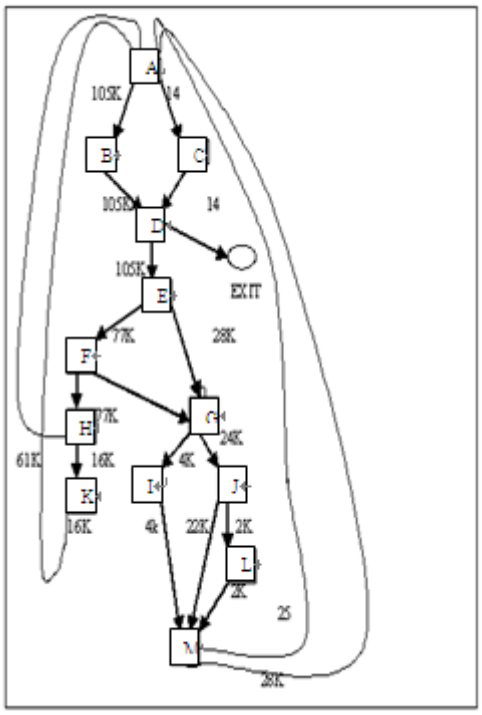


Fig 1.3 a) Before Coalescing

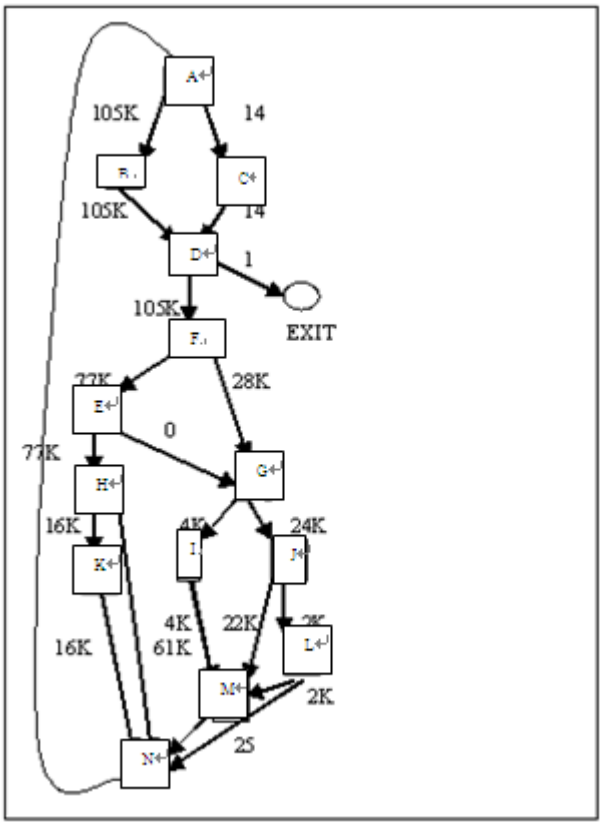


Fig 1.3 b) After Coalescing

Finally, any hazard conditions that exist along a path are used to give the path lower priority. Hazard conditions include procedure calls and unresolved memory stores (typically pointer update)[6]. Hazard conditions limit the effectiveness of optimization and scheduling for the entire hyperblock since the compiler must make conservative assumptions regarding the hazards to ensure correctness.

The path priority function is defined more precisely by the following three equations:

$$Dep\_ratio_i = 1.0 - (dep\_height_i / max(dep\_height_j)) \quad 1 <= j <= N \quad 1.1$$

$$op\_ratio_i = 1.0 - (num\_ops_i / max(num\_ops_j)) \quad 1 <= j <= N \quad 1.2$$

Equation (1.1) calculates the ratio of a particular path's dependence height with respect to the path with the largest dependence height in the region. In order to make similar dependence heights more favorable, this ratio is subtracted from one. Correspondingly, the ratio of the number of operations along a path in the region is calculated by equation(1.2) . These two equation are used to gauge the height and resource dominance of each path through the region .

The overall priority is calculated by equation (1.3). The priority is the product of two terms . The first term is the probability of the path which is scaled by a hazard multiplier. The hazard multiplier is used to reduce the probability of paths that contain a hazardous instruction . Currently , a value of 0.25 is used for any path containing a subroutine call or an unresolved memory store .For paths containing no hazards , a value of 1.0 is used. The second product term is sum of the previously computed dependence and operation ratios along with a constant term, K. The constant term is used to indicate a base contribution of the path probability. In this manner, a path with the largest dependence height and number of operations still may have a non zero priority .Currently , the value of K is set to 0.1.

As previously mentioned, after the priorities for all paths are calculated , the paths are sorted in priority order and considered for inclusion from highest to lowest priority[4]. The algorithm used for block selection is presented in fig 1.4 . Paths are included in the hyperblock provided that they do not violate any of the following three conditions.

1) First, the additional resources required by a path may not cause the total number of resources required by a path may not cause the total number of resources required by the hyperblock to exceed the estimated available resources.
2) Second, the dependence height of a path may not exceed the dependence height of the highest priority path ($path\_height_i$) by more than a predefined fraction.
3) Finally,the priority of a path must be within some fraction of the priority for the last included path. This restriction prevents disparate low priority paths from being included in a hyperblock consisting of high priority paths .

The final selection of blocks that are actually selected for inclusion are calculated by taking the union of all blocks along the selected paths.

```
/*  Predefined variables for block selection  */
ISSUE_WIDTH=1 to 8
RES_MULTIPLIER=2
MAX_DEP_GROWTH =3
MIN_PATH_PRIORITY_RATIO =0.10

block selection (region)
{
enumerate all paths in region
calculate priority of each path
sort paths from largest to smallest priority
/* Initialization of loop variable */
avail_resources=ISSUE_WIDTH X dep_height X RES_MULTIPLIER
used_resources= 0
last_priority=0.0
sel_paths = 0
for(i=1 to num_paths)
{    /* Check if there enough resources available to include the path */

if ((num_ops_i + used_resources) > avail_resources)
{       continue    )
/*  prevent paths with a large relative dependence heights from being included
*/
if( dep_height_i  > (dep_height_1 x  MAX_DEP_GROWTH))
{                  continue            )
/* Do not include paths with a small relative priority to that of the last included
path */
if(  priority_i < (last_priority x MIN_PATH_PRIORITY_RATIO))
{       continue    )
/* Include the path in the hyperblock  */
sel_path = sel_paths U path_i.
used_resources = used_resources + num_ops_i.
last_priority =priority_i.
}
sel_blocks = all blocks contained within sel_paths
return sel_blocks
}
```

Fig 1.4 Block selection algorithm

The block selection algorithm[3] utilizes a simplified scheme to model processor resources. Resources are modeled by keeping track of the estimated number of available instruction slots. Currently, instruction slots are not classified by allowable instruction types. Therefore , each instruction under consideration may be placed in any available slot. The available number of instruction slots is calculated by multiplying the issue width of the target processor by the dependence height of the highest priority path. In addition, the number of available resources is increased by a padding factor referred to as the *RES_MULTIPLIER*.

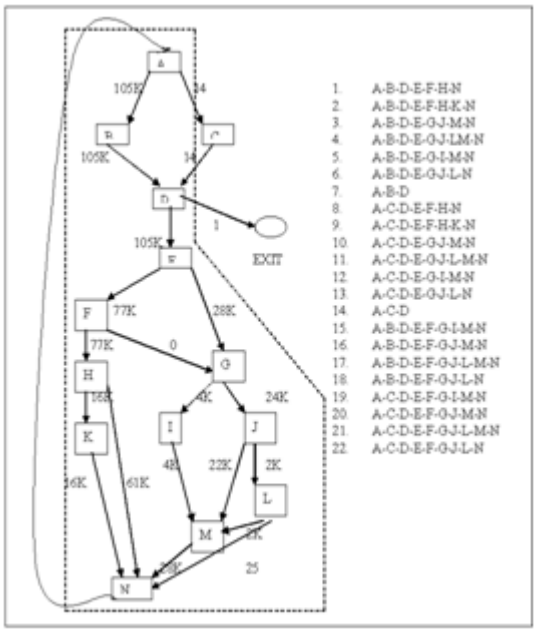


Fig 1.5 block selection applied to the inner loop segment of wc

A padding factor of 1.0 constraints the selection algorithm not to increase the schedule length of the highest priority path due to resource demands of other paths. In practice, this was found too restrictive. For many cases, increasing the schedule length of the highest priority path by a modest margin is profitable because more paths can be overlapped. The value of *RES_MULTIPLIER* is set to 2.0 .

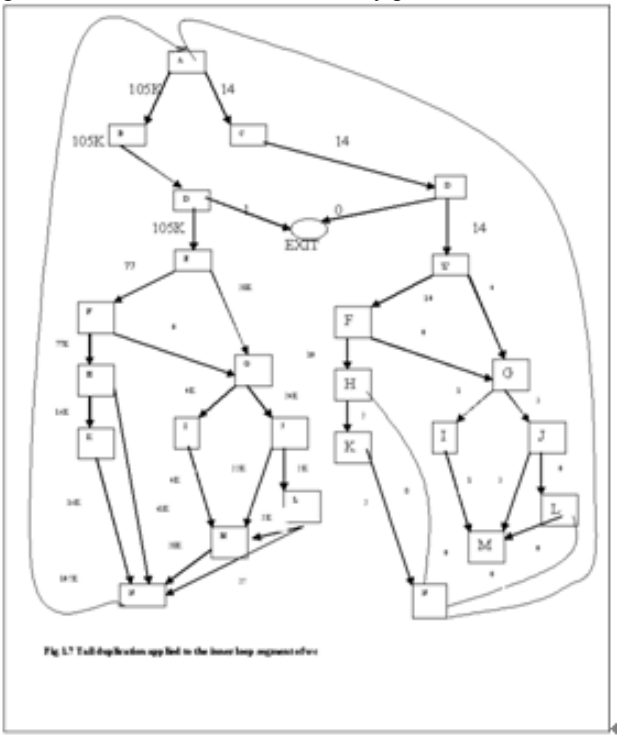
The application of block selection algorithm to ***wc*** example is illustrated in fig 1.5. In the right hand portion of the figure, the execution paths are enumerated in priority order . This loop region contains 22 unique paths , with the path A-B-D-E-F-H-N having highest priority. Path 1-7 are chosen for inclusion by the block selection algorithm. After path 7, the priority value for the renaming paths drops dramatically due to their low execution frequency .Additionally, Block C contains a hazardous instruction (a subroutine call) so the priorities of all paths which contain block C is further reduced . The blocks which are actually selected for inclusion are then calculated by taking the union of all blocks from the selected paths .The result of block selection is that all blocks with the exception of block C are chosen for the hyperblock. With this strategy, some paths which were not chosen may indeed be included in the hyperblock .For this example, path 15-18 are actually selected since all the blocks which lie along those paths are chosen . In reality these paths could be excluded if desired , but little advantage is gained by doing this.

Step 4-Tail Duplication : In order to make the eventual hyperblock be single entry,control flow from non selected blocks to selected blocks (other than entry block) must be eliminated . Such paths of control are referred to as side entry points into the hyperblock. In fig 1.5 ,a side entry point exists from block C to block D . Tail duplication is used to remove all side entry points of a hyperblock.

```
tail_duplication(set_of_bb)
{
        Let set_of_bb be the blocks that are selected for hyperblock formation
and bb1 is the entry of the hyperblock .
done=false
while(not done)
{   done=true
for each basic block in set_of_bb, bb {
if( bb ? bb1)
mark each bb that has predecessor not in set_of_bb
}
for each basic block in the set_of_bb, bb
{   if ( bb is marked )
{ if bb is not duplicated yet
duplicate bb
else
use previously duplicated bb
change all incoming flow arcs of basic blocks not in set_of_bb
from pointing to bb to duplicate bb .
done= false
                        }
                }
        }
}
```

Fig 1.6 An algorithm for tail duplication

The tail duplication algorithm (fig1.6) transforms the control flow graph by first marking all blocks which have side entry points . Then All selected blocks that may be reached from a marked block without passing through entry block are also marked . Finally, all the marked blocks are duplicated and the control flow arcs corresponding to the side entry points are adjusted to transfer control to the corresponding duplicate blocks. Note that blocks are duplicated at most one time regardless of the number of side entry points.



The ***wc*** example after tail duplication is shown in fig 1.7. The set of blocks which must be duplicated is identified by first marking the target of side entry point, namely, block D .Then all selected blocks in which control can reach from block D without passing through block A are marked. The set of reachable blocks contains blocks EF,G,H,I,J,L,M and N. Tail duplication proceeds by replicating block D and all of the reachable blocks. Lastly, the C-D control arc is adjusted to C-D to remove the side entry.

Step 5-If Conversion: The final phase of hyperblock format is if-conversion .If- conversion removes all control

flow among the blocks selected for the hyperblock using the conditional execution. However, explicit branches remain to handle all control flow which exists the hyperblock. In the current implementation, the RK if conversion algorithm (Fig 1.8) is utilized. The RK if – conversion algorithm first calculates the localized control dependence information among the selected basic blocks. Control dependences are maintained as a set of edges in the control flow graph which determines the execution condition of a particular basic block. The control dependence information is localized because only control flow among the selected blocks is considered. All control dependences resulting from branches not in the region or branches which exists the hyperblock are ignored for the purpose of calculating control dependences. This strategy minimizes the number of control dependences represented with predicates to only those branches which are targeted for elimination.

```
if_conversion(set_of bb)
{     compute post denominator (set_of_bb)
      compute control dependencies (set_of_bb)
      decompose control dependencies(set_of_bb)
      augmented K(set_of_bb)
      for each basic block,bbi
      {
       use the predicate register assigned in R
for each forward conditional flow arc use K to determine which predicates are
dependent on this arc. And add the predicate define operation.
remove all forward branches
predicate this basic block with the assigned register
      }
}
```

Fig 1.8 RK algorithm for if-conversion

Once the control dependence information is calculated , one predicate register is assigned to represent each unique set of control dependences. Therefore, all blocks which share a common set of control dependences will be executed under the same predicate. Predicate comparison instructions[2] are inserted into all basic blocks which are the source of the control dependence edges associated with a particular predicate. The predicate compare condition is determined by the branch condition specified by a particular control dependence edges. After the predicate comparison instructions are inserted, all instructions in each selected block, including the newly inserted predicate comparisons, are conditioned under the predicate assigned to their block. Finally all conditional, unconditional branches from selected blocks to other selected blocks are removed. The predicate code is placed linearly in the final hyperblock using a topological sort of the original hyperblock control flow graph.

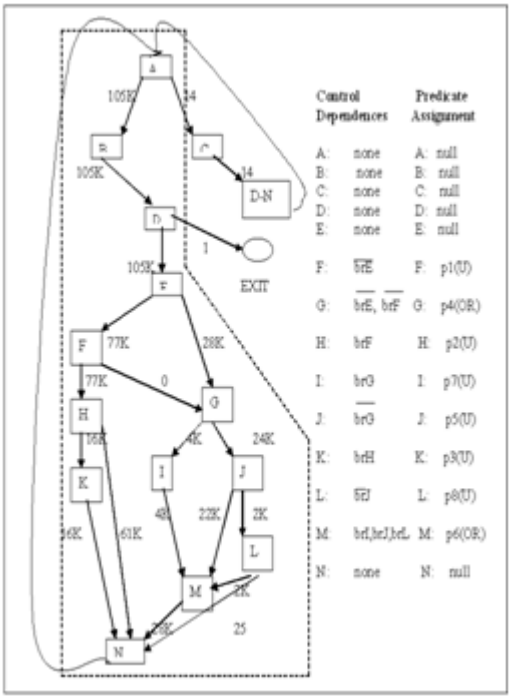


Fig 1.9 Localized control dependence calculation and predicate assignment for the inner loop segment of wc

The if- conversion step performed on the ***wc*** example is illustrated in fig 1.9 and fig 1.10.The calculation of the localized control dependence information and the predicate assignment are shown in fig 1.9. Blocks A,B,D,E and N have no local control dependences. Therefore, these blocks will always be executed if the hyperblock is not exited prematurely through a side exit and do not require predicates. The remaining blocks are control dependent on the edges specified in figure. Control dependences are denoted by indicating the branch from which they originate. True and Complement conditions are used to distinguish the left hand and right-hand control flow arcs out of a particular block, respectively. For example, the control dependence for block **J** is ***brG***, indicating the right – hand edge leaving block **G**. The example hyperblock contains eight unique sets of control dependences, thus eight predicate are required. The mapping of control dependences to predicates and the assignment of predicates to basic blocks are also shown in fig 1.9.

III. Predicate Dependencies

Unconditional predicates are used for predicates which have a single edge in their control dependences sets. For example, predicates p1, p2,p3, p5, p7and p8 are unconditional. On the other hand, OR-type predicates are used for predicates which have multiple edges in their control dependence sets .OR-type predicates are necessary with multiple edges since the predicate should beset to 1 if either edge is traversed . Predicates p4 and p6 must be OR- type in the example. In reality, OR-type predicates could be used exclusively for if - conversion. However, OR-type predicates require explicit clearing for proper use. With conditional predicates, explicit clearing is not required; thus, they are used whenever possible to reduce the number of necessary clears.

To illustrate the insertion of predicate comparison instructions, consider the calculation of predicate p4, which is the predicate for block G. The control dependence set for block g is ***{brE,brF}***; thus , comparison instructions must be placed in blocks which originate the control dependence edges, namely, blocks E and F. From fig 1.2, the compare conditions are deprived from conditional branches which terminates these blocks . Therefore, both comparison will utilize a ***pge*** instruction to correspond with the ***bge*** instruction .The final code after if-conversion , presented in Figure 1.10, shows the two ***pge*** instructions which define predicate p4,as OR-type. Note also that OR-type predicates require explicit clearing before they are defined of referenced .Thus ,the ***pclr*** instruction is placed at the top of the hyperblock.

```
Loop:  pclr p4,p6
       Id_i  r98,r3 ,0
       add r27 ,r98,-1
       st_i r3,0,r27
       blt r98,1,LC
       Id_i r30,r3,4
       add r29 , r30 ,1
       st_i, r3 ,4, r29
       Id_c r4 ,r30,0
       beq r4,-1,EXIT
       Id_i r33,r73,0
       add  r32 , r33 ,1
       st_i  r73,0,r32
       pge p4(OR), p1(U) ,32,r4
       pge p4(OR), p2(U) ,r4,127 (p1)
       peq p3(U),-,0 ,r2 (p2)
       peq p6(OR),p5(U) ,r4,10(p4)
       peq p7(OR),-, r4,10(p4)
       peq p6 (OR) ,p8(U),r4,32(p5)
       Id_i r36,r72,0,(p3)
       add r35,r36,1(p3)
       st_i r72,0,35(p3)
       add r2,r2,1(p3)
       st_i r71,0,r38(p7)
       peq p6(OR),-,r4,9 (p8)
       mov r2 ,0 (p6)
       jmp Loop
```

Fig 1.10 Inner Loop segment of wc

If-conversion is completed by associating instruction in each basic block of the hyperblock with the appropriate predicate and subsequently removing all internal control flow. The predicates of each instruction are derived directly from their original basic blocks and the predicate assignment given in fig 1.9 .For example the ***Id_i, add*** , and ***st_i*** instruction originally in block I , are conditioned under predicate p7 in the final hyperblock code . When control flow is removed, both conditional and unconditional branches are eliminated. ***Tn*** the final hyperblock for the ***wc*** example ,all but three branches are removed . The remaining branches are the two infrequent branches which exit the hyperblock(highlighted in fig 1.10) and an unconditional loop-back branch at the bottom of the hyperblock.

REFERENCE

[1] .Sweta Verma, Ranjit Biswas, J.B. Singh "Modified Architectural Support for predicate execution of Instruction level parallelism", International Journal of Computer and Electrical Engineering (IJCEE),April 2010 ,Vol 2,No.2 page 208-211.

[2] V.Aho,R.Sethiand J.D.Ullman,"Compilers: Principles, Techniques and Tools", Addison-Wesley Publishing Company, 1986.

[3] R.P.Colwell et al.,"A VLIW Architecture for a Trace Scheduling Compiler ", IEEE rans. On Computers , August 1988

[4] Subramanian Rajagopalan , Sreeranga P. Rajan, Sharad Malik, Sandro Rigo , Guido Araujo , and Koichiro Takayama;" A Ratargetable VLIW Compiler.Framework for DSP with Instruction Level Parallelism", IEEE Transactions on Computer –Aided esign Of Integrated Circuits and Systems, Vol. 20, No.11, November 2001.

[5] Peng-Sheng Chen,Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee,"Interprocedural Probabilistic Pointer Analysis", IEEE Transaction on Parallel and Distributed Systems, Vol.15, No. 10, October 2004.

[6] "An Architecture Framework for Introducing Predicated Execution into mbeddedMicroprocessors",Daniel A. Connors,David I. August,Kevin M. Crozier,And Wenmei W. Hwuand Jean-Michel Puiatti

[7] Anantaraman A, Seth K, Patil K, Rotenberg E, Mueller F (2003) "Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems". In: Proceedings of the international symposium on computer architecture, June 2003

[8] Berg C, Engblom J, Wilhelm R (2004) ,"Requirements for and design of a processor with predictable timing". In: Proceedings of the dagstuhl perspectives workshop on design of systems with predictable behavior

[9] Deverge J, Puaut I (2005)," Safe measurement-based WCET estimation".In Proceedings of the Euromicro international workshop on WCET analysis .

[10] Fisher JA, Faraboschi P, Young C (2005)" Embedded computing: a VLIW approach to architecture, compilers, and tools". Kaufmann, Los Altos

[11] .Cvetanovic and Kessler ," Performance analysis of the Alpha 21264-based Compaq ES40 system. Notes on Proceedings of the 27th ACM International Symposium on Computer Architecture, 2000. Vancouver, Canada.

[12] . Kunle, Basem, Lance, Ken, Kunyung "The Case for a Single-Chip Multiprocessor "Pages 1-2 Computer Systems Laboratory Stanford University

[13] .Mikko H.,Lipasti and John Paul Shen, "Exceeding the Dataflow Limit via Value Prediction"Departmen of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh PA, 152 13

[14] Joseph A. Fisher and Stefan M. Freudenberger, "Predicting Conditional Branch Directions" From Previous Runs of a Program Pages 85-87