On the Benefits of Speculative Trace Scheduling in VLIW Processors

Manvi Agarwal and S. K. Nandy
CADL, SERC
Indian Institute of Science,
Bangalore, India

Jos van Eijndhoven and S. Balakrishnan
Philips Research Laboratories
Eindhoven, The Netherlands

Abstract

VLIW processors are statically scheduled processors and their performance depends on the quality of the compiler's scheduler. We propose a scheduling scheme where the application is first divided into decision trees and then further split into traces. We have developed a tool "SpliTree" to generate traces automatically. Using dynamic branch prediction for selecting the root of the decision tree from which the traces are scheduled using our scheme, we obtain approximately 1.4x performance improvement over that using decision trees for Spec92int benchmarks simulated on TriMedia^{TM} processor.

Keywords: VLIW processor, scheduling, speculative trace scheduling, ILP.

1 Introduction

"Very Long Instruction Word Processor", widely known as *"VLIW"* processor is a paradigm for simple hardware and high compute capacity. In a VLIW processor the micro-architectural details are exposed to the compiler and the compiler generates schedules to exploit maximum *Instruction Level Parallelism* (ILP) present in the code. The two main methods of scheduling in VLIW processors are: *basic block scheduling* and *extended basic block scheduling*. Basic block scheduling is limited in its scope to exploiting ILP because of the small size of the basic blocks. (4-5 interdependent operations on an average in each basic block.) In extended basic block scheduling, groups of basic blocks are scheduled as a single unit. Extended basic block scheduling can be categorized into following: *trace scheduling, superblock scheduling, hyperblock scheduling* and *decision tree scheduling*. Although the scope of optimization is enlarged, these methods still suffer from the ineffective use of the VLIW issue slots to launch operations, as explained in the later text of the paper.

In this paper we propose a new scheduling scheme which ensures minimal issue slot wastage. The application is first divided into decision trees and then further split into traces by the tool *SpliTree* developed by us. Traces of the application are formed with the help of profile information of the application. All the decision points are removed from the body of the trace and extra code is inserted at the tail to check for correct conditions. Removal of decision points from the body of the trace assists the compiler to perform optimizations which are not possible otherwise. Our scheme achieves a gain in schedule length and overall improvement in performance. The traces are executed speculatively and hence the penalty paid is the number of cycles to the error exit jump, which is often at the end of the trace.

The rest of the paper is organized as follows: section 2 gives an overview of the scheduling techniques for VLIW processors. In section 3 we explain our speculative trace scheduling scheme and give results of the simulations. We summarize the contribution of the work in section 4 and draw conclusions.

2 Related Work in Scheduling in VLIW Processors

Scheduling is the process of generating a sequence of micro-operations of the code so that they can be loaded on the underlying hardware units for execution. As mentioned earlier, two methods of

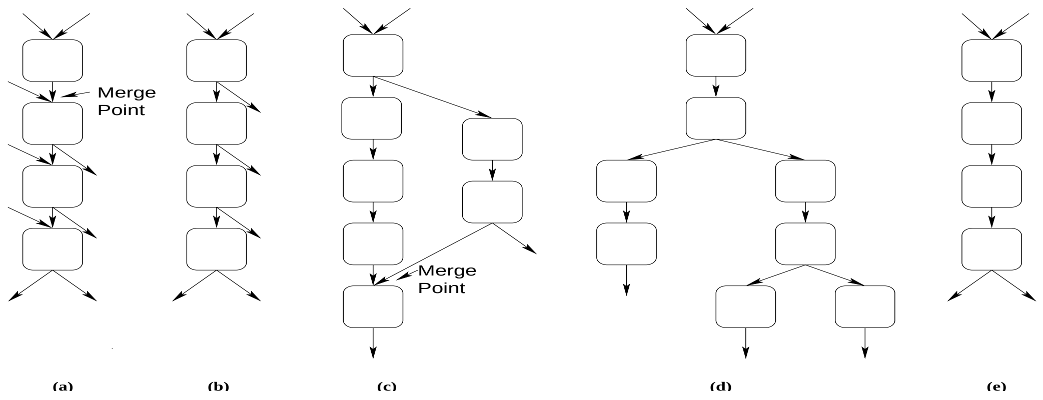

Figure 1: Types of Extended Basic Block Scheduling Scopes: (a) Trace, (b) Superblock, (c) Hyperblock, (d) Decision tree and (e) Traces for Speculative Trace Scheduling

scheduling are: *Basic block scheduling* and *Extended basic block scheduling*. The scheduling scopes used in extended basic block scheduling schemes are illustrated in figure 1. Various extended basic block scheduling schemes are elaborated below.

In *trace scheduling* [4], the compiler picks the most likely path of execution and schedules it for execution. Using a trace, it is possible to expose available ILP because several basic blocks are included in it which can be scheduled in parallel on the underlying VLIW processor. Side entries as well as side exits are allowed in traces because of which book-keeping of the operations, which have been moved across basic blocks, is required. In superblock scheduling the overhead of book-keeping is obviated as elaborated below.

Superblock scheduling [8] is similar to the trace scheduling except that it does not allow any side entries. Traces are formed along with tail duplication past fork points. There is only a unique entry point as opposed to trace scheduling which has multiple entry points. This scheduling scheme does not permit code motion past fork points which makes book-keeping unnecessary and gives an advantage over trace scheduling. A drawback of superblock and trace scheduling is that both the scheduling schemes execute only one path of the application. Selection of the wrong path for execution based on profile information leads to wastage of processor cycles. *Hyperblock scheduling* and *decision tree scheduling* schedule multiple paths in one scheduling scope.

Hyperblock scheduling [7] is different from trace and superblock scheduling in that multiple paths are scheduled in a single unit. Hyperblock scheduling uses *predication* to form scheduling scopes. *Predication* is the conditional execution of instructions based on the value of boolean operand which is known as the *predicate*. As shown in figure 1, the hyperblock can contain multiple paths combined together by if-conversion and tail duplication. Basic blocks containing procedure calls and unresolvable memory accesses are not included in a hyperblock. It is a single entry structure with multiple side exits. The process of if-conversion transforms the control dependency to data dependency and hence optimizations can be performed on the hyperblock which are not possible in trace scheduling.

Decision tree scheduling [5] is another method of extended basic block scheduling and is similar to superblock scheduling due to the absence of join points and side entries. Each leaf of a decision tree ends in a procedure call or jump to a different tree. There are no side exits from the interior basic blocks of a decision tree and there is only one entry point which is the root of the decision tree. Predication can be employed in decision trees similar to hyperblock scheduling, to perform compiler optimizations and hence exploit more ILP.

Control operations in all the scheduling scopes discussed above are either predicated or delayed. In the case of delayed branch operations, the scheduler has to find appropriate operations to fill the delay slots of the branches. If the scheduler is unable to find these operations, it fills them with *"nops"*. The issue slots thus get wasted which otherwise

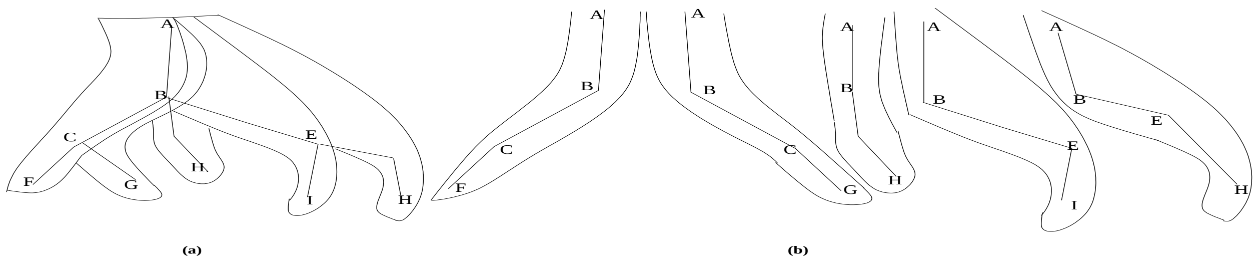

Figure 2: (a) A Decision Tree (b) Traces of the Decision Tree

could be used to schedule operations on the functional units. Due to the presence of control operations in the decision tree, required code optimizations cannot be performed because the operations following a branch cannot be scheduled earlier than the branch. This can be avoided by using guarded or predicated execution in which the control dependency is converted into data dependency with the help of predicates. With the help of predicate registers operations are scheduled as soon as their data dependency is met. The value of the predicate registers determines whether the result would be committed or masked. Though efficient schedules can be generated with higher code density, effectively this leads to the wastage of issue slots of the VLIW processors. These issue slots could instead be used to issue operations of the taken path. We propose a speculative trace scheduling scheme in this paper in which there is minimal issue slot wastage and efficient schedules are generated with high code density. The scheduling unit in our scheme is a single entry and single exit structure which we call *"probable execution trace"*. Henceforth, *"probable execution trace"* will be referred to as *"trace"*. In the following section we explain our scheduling scheme in more detail.

3 Speculative Trace Scheduling

Speculative trace scheduling scheme consists of two phases. In the first phase of the compilation process the application is divided into decision trees. After this phase an intermediate file is obtained which is a tree file and it contains the application code divided into several decision trees. In the second phase of the compilation process, the tree file is transformed into a trace file i.e. each decision tree in the tree file is split into its corresponding traces. These trace files are then scheduled on the underlying VLIW processor using list scheduling. The decision trees are split into traces in the manner shown in figure 2. Path ABCF forms one trace as shown in figure 2. Similarly all the possible paths in the decision tree are split into corresponding traces. During the formation of the traces, the decision points are removed from the body of the trace and extra operations are inserted at the tail to check for correct control flow conditions. The operations in the trace do not face ordering constraints during scheduling because of the removal of decision points from the body of the trace. The operations are scheduled as soon as their data dependency is met. During the formation of the traces, each trace is annotated with the probability of execution of the path included in it based on the profile information of the application (if available). If the profile information of the application is not available then equal probability is given to all paths of the decision tree. No delay slots are allocated for the branch operations at the end of the trace because branch prediction is employed to predict branch direction when the trace is executed. Gain in schedule lengths is achieved and code density is increased in the schedules generated by this scheme. As the check for the correctly executed trace is done only at the end of the trace, the penalty paid on a trace misprediction is the length of the trace. The pro-

```
a = 0;
b = 1;
sum = a + b;
for (i = 0; i <= 6; i++)
{
    b++;
    sum = sum + b;
}
printf("Sum = %d\n",sum);
```

Figure 3: Pseudo code for the Decision Tree shown in figure 4

```
__main_DT_1: (* DT_1, BB:2 line 11 *)
tree (12)
        (* BB:2, line 11, 0x21aca0 *)
        1 rdreg (4);
        2 rdreg (33);
        3 rdreg (35);
        4 rdreg (34);
        5 rdreg (36);
        6 iaddi (1) 5;
        7 iadd 5 4;
        8 iaddi (1) 7;
        9 iaddi (1) 3;
        10 ileqi (6) 9;
        (* End of BB:2, line 9, 0x21aca0 *)
        if  10  (0.857143) then
                 12 wrreg (36) 6 after 5;
                 13 wrreg (34) 8 after 4;
                 14 wrreg (35) 9 after 3;
                 gotree {__main_DT_1}
        else   (10)
                (* BB:3, line 14, 0x21d4e0 *)
                15 st32d (0) 1 2;
                16 st32d (4) 1 8;
                17 uimm (__main_DT_2);
                18 wrreg (5) 2;
                19 wrreg (6) 8;
                20 wrreg (2) 17;
                (* End of BB:3, line 14, 0x21d4e0 *)
                gotree {_printf}
        end (10)
endtree
```

(a)

```
__main_DT_1:
        (* cycle 0 *)
        IF r1    iaddi(0x1) r36 -> r36                 (* alu/Op6 *),
        IF r1    iaddi(0x1) r35 -> r35                 (* alu/Op9 *),
        IF r1    uimm(__main_DT_1) -> r8,
        IF r1    iaddi(0x1) r36 -> r37                 (* alu/Op7 *),
        IF r1    iadd r0 r33 -> r5  (* WR *)          (* alu/OP18 *);

        (* cycle 1 *)
        IF r1    igtri(6) r35 -> r7                   (* alu/Op10 *),
        IF r1    iadd r34 r37 -> r6                    (* alu/Op8 *).
        IF r1    iadd r34 r37 -> r34,
        IF r1    uimm(__main_DT_2) -> r37              (* const/Op17 *),
        IF r1    nop;

        (* cycle 2 *)
        IF r7    iadd r0 r37 -> r2  (* WR *)          (* alu/Op20 *),
        IF r1    ijmpf r7 r8,
        IF r7    ijmpi(_printf),
        IF r7    h_st32d(4) r6 r4                      (* dmem/Op16 *),
        IF r7    h_st32d(0) r33 r4                     (* dmem/Op15 *);

        (* cycle 3 *)
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop;

        (* cycle 4 *)
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop;

        (* cycle 5 *)
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop,
        IF r1     nop;
```

(b)

Figure 4: (a) Decision tree as generated by the TriMedia compiler. (b) Schedule of the decision tree in (a) generated by the TriMedia scheduler

cessor has to roll back to the previous checkpoint state in the event of a misprediction with the help of additional hardware support. A set of shadow registers can be maintained along with the working set of registers in the hardware. The state of the processor at the end of previously executed correct trace is stored in the shadow registers. In the case of a correct prediction, working registers are committed into the shadow registers and execution of the new trace proceeds. On a misprediction the working registers are discarded and the state of the processor is retrieved from the shadow registers and execution of the next trace starts. The memory writes of the current trace can be labeled pending till the check for the correct trace is made. If the executed trace turns out to be correct, pending memory operations are marked committed. On a misprediction these pending memory writes are discarded.

3.1 Exploiting ILP

The schedules generated by our scheme have higher code density as compared to the schedules generated using decision trees. We explain this *vis a vis* the scheduler used by TriMedia tool set. Figure 4(a) shows the decision tree for the *"for loop"* of the pseudo code shown in figure 3 generated by the TriMedia compiler and figure 4(b) shows the corresponding scheduled code generated by the scheduler (no use of loop-unrolling capabilities of the compiler was made to generate the schedule of 4(b)). The number in the parenthesis of the *"if"* condition (figure 4(a)) shows the probability with which that condition is taken. As seen in figure 4(b), the scheduler has scheduled the whole tree by using predicated execution. All the operations are *"if guarded"*. Register r1 is hardwired register of TriMedia with value 1. Register r7 is the masking register whose least significant bit (LSB) determines whether the corresponding results would be committed or masked. If the value of LSB of r7 is 1 then the results are taken into account and if it is 0 then the results are masked and the execution proceeds. The total number of cycles taken by the processor to execute this schedule is 6. As evident from the figure, last 3 cycles do not issue any operation but they have to be included because of the branch operations which have a delay of 3 cycles. 4 issue slots are seen wasted in third cycle of the schedule to schedule the "else" part (with probability of execution as 0.14) of the decision tree with register r7 as their predicate register since both the

```
__main_DT_1b:
tree(12)
      (* overall probability of the trace __main_DT_1b is 0.857143 *)
      (* source line: 11 *)
      (* successors: __main_DT_1b (0.857143), __main_DT_1c (0.142857) *)
      1 rdreg (4);
      2 rdreg (33);
      3 rdreg (35);
      4 rdreg (34);
      5 rdreg (36);
      6 iaddi (1) 5;
      7 iadd 5 4;
      8 iaddi (1) 7;
      9 iaddi (1) 3;
      10 ileqi (6) 9;
      (* source line: 9 *)
      (* successors: none *)
       12 wrreg (36) 6 after 5;
       13 wrreg (34) 8 after 4;
       14 wrreg (35) 9 after 3;
        if 10 then
              gotree {__main_DT_1}
        else (10)
              gotree {_error}
        end (10)
 endtree
```

(a)

```
__main_DT_1b:
       (* cycle 0 *)
      IF r1   iaddi(0x1) r36 −> r36           (* alu/Op6 *),
      IF r1   iaddi(0x1) r35 −> r35           (* alu/Op9 *),
      IF r1   iaddi(0x1) r36 −> r7            (* alu/Op7 *),
      IF r1   uimm(__main_DT_1) −> r37,
      IF r1   nop;

      (* cycle 1 *)
      IF r1   igtri(6) r35 −> r8              (* alu/Op10 *),
      IF r1   iadd r34 r7 −> r34              (* alu/Op8 *),
      IF r1   nop,
      IF r1   nop,
      IF r1   nop;

      (* cycle 2 *)
      IF r1   nop,
      IF r1   ijmpf r8 r37,
      IF r8   ijmpi(__error),
      IF r1   nop,
      IF r1   nop;
```

(b)

Figure 5: (a) Trace of the most probable path of the decision tree shown in figure 4(a) generated by *SpliTree*. (b) Schedule of the trace shown in (a) generated by the TriMedia scheduler using branch prediction.

"if" and the "else" part are scheduled in predicated execution. Figure 5(a) shows the trace of the most probable path of the tree shown in figure 4(a). The head of each trace contains the overall execution probability of the trace and does not have control operations in the body. The scheduled code of the trace in figure 5(a) is shown in figure 5(b). If the branch prediction is accurate then the trace takes only 3 cycles to execute the same part of the code as opposed to 6 cycles taken by the corresponding tree. On a trace misprediction, roll back operations are performed and the penalty paid is the length of trace, which in this example is 3 cycles. In the schedule of figure 5(b) there is no issue slot wastage in scheduling the second path of the decision tree of figure 4(a). As apparent from the figures the schedule length has shrunk by 3 cycles by using speculative trace scheduling.

3.2 Simulation and Results

The simulation environment used for the project is Philips TriMedia SDE version 2.0 tool set, which is a Philips proprietary software tool. The TriMedia compiler *"tmcc"* breaks down the code into several decision trees depending on the application and generates tree files in the intermediate format, known as *"trees code"* in the terminology of TriMedia. These tree-files are converted into trace-files with the help of our tool *SpliTree***.** *SpliTree* takes as input these tree files and generates trace-files with all the trees split into their corresponding traces. While generating these traces *SpliTree* calculates the overall probability of the execution of the trace based on the profile information (if available) obtained from the previous runs of the application. If the profile information is not available then equal probability is given to all paths. Each trace is annotated with this probability. The trace label is in accordance with the label of the last basic block included in it. These traces are then scheduled on the underlying hardware units with the help of TriMedia scheduler *"tmsched"*. *"tmsched"* at the time of scheduling, consults machine description file to generate proper schedules. Since a trace is devoid of control operations in its body, there is no overhead of idle processor cycles as illustrated in figure 5. Figure 4(a) shows the tree code generated by the *tmcc* whose schedule is given in figure 4(b). The code of the most probable trace of the same tree is shown in figure 5(a) with its schedule in figure 5(b). The number of branch delay slots is 0 cycles in our schedule because dynamic branch prediction is employed to predict the branch direction at the end of the trace when the trace is executed. To account for the reduction of branch delay slots in the case of branch prediction, we have modified

	No Branch Prediction	Branch Prediction
Decision Trees	Case 0	Case 1
Probable Execution Traces	Case 2	Case 3

Figure 6: Scheduling Space of Decision Trees and Probable Execution Traces

the machine description of TriMedia by changing the branch delay to 0 cycles.

In order to clearly show the efficiency of the speculative trace scheduling scheme proposed in this paper, we cover the scheduling space of both decision trees and probable execution traces, with and without branch prediction. This is pictorially depicted in figure 6. The expression for the execution time of the application, "ET_{tree}" for Case 0 is given by:

$$ET_{tree} = \sum_{\forall trees} E_{tree} * L_{tree} \qquad (1)$$

where, "E_{tree}" is the execution count of the decision tree, "L_{tree}" is the schedule length of the tree and can be expressed as $L_{tree} = \sum_{path=1}^{n} L_{path} * p_{path}$. "$L_{path}$" is the length of each path of a decision tree and "p_{path}" is the probability of execution of the path. The expression for the execution time, "ET_{ptree}" of the application in Case 1 is given by:

$$ET_{ptree} = \sum_{\forall trees} E_{tree} * (L_{tree} + MP_{tree}) \qquad (2)$$

where "MP_{tree}" is the effective penalty for mispredicted tree. The expression for calculating "MP_{tree}" is: $MP_{tree} = R * MispredictionPenalty$ where "R" is the next PC misprediction rate of the branch predictor and misprediction penalty for each tree is equal to the number of pipeline stages between the fetch and the execute unit of the processor. Execution time, "ET_{trace}" of the application in Case 2 is given by:

$$ET_{trace} = \sum_{\forall traces} E_{trace} * L_{trace} * p_{trace} \qquad (3)$$

where "L_{trace}" is the schedule length of the trace, "E_{trace}" is the execution count of trace and "p_{trace} " is the probability of the execution of the trace. Execution time, "ET_{ptrace}" of the application in Case 3 is given by:

$$ET_{ptrace} = \sum_{\forall traces} E_{trace} * p_{trace} * (L_{trace} + MP_{trace}) \qquad (4)$$

where "MP_{trace}" is the effective misprediction penalty of the trace and is given by $MP_{trace} = R * MispredictionPenalty$. "$R$" is the next PC misprediction rate of the branch predictor and misprediction penalty is equal to the length of the trace. The results for Cases 1, 2 and 3 are normalized with respect to that of Case 0 and are reported in Table 1.

As already mentioned in earlier sections, additional hardware (which is not present in TriMedia) is necessary to nullify the execution of wrongly predicted traces. The branch predictor for the VLIW processors used in this work is the one proposed by Jan Hoogerbrugge in [3]. The rate of branch misprediction depends on the implementation of the branch predictor as well as on the application. If a lot of branch operations are present in an application and the behavior of the branches change frequently then the rate of branch misprediction is high for such an application. Results have been provided for Spec92 benchmarks. We used Spec92 benchmarks to evaluate our results because these are adequate to quantify the results for embedded applications.

As can be seen in Table 1, a gain in performance is achieved in all the three cases as compared to decision trees with delayed branches of TriMedia. Case 1 results have been reported by Jan Hoogerbrugge [3] and we have reproduced them in this paper for the sake of comparison with our speculative trace scheduling scheme. Performance gain in the case of branch prediction is obvious considering the fact that branch delay slots are reduced to zero. Case 2 results (shown in column 2 of Table 1) give the theoretical advantage of trace scheduling (unpredicted traces) over decision tree scheduling. These results are produced to show the efficiency of trace scheduling as compared to decision tree scheduling. The advantage is obtained due to the removal of control operations from the body of the trace because of which the operations are moved higher up in the schedule and issue slots are utilized more efficiently. Decision trees with branch predic-

Table 1: Performance improvement relative to delayed branches in Trimedia for three cases: predicted branches in trees, split traces (no branch prediction and branch prediction in traces.

Benchmark	Scheduling Schemes		
	Predicted Trees	Unpredicted Traces	Predicted Traces
008.espresso	1.1688	1.2336	1.5387
022.li	1.2266	1.0825	1.3631
023.eqntott	1.1652	1.1348	1.4009
072.sc	1.0913	1.1111	1.3677
Average	1.1629	1.1405	1.4170

tion perform better than unpredicted traces because of the absence of branch delay slots in the former. A significant gain is seen in the case of predicted traces (column 3 of Table 1) as compared to predicted trees. This is due to two reasons: 1) branch delay slot reduction and 2) the removal of decision points from the body of the trace because of which ordering constraints are absent for scheduling operations.

The performance achieved by our scheduling scheme is approximately 1.417 times the original TriMedia scheduling scheme which is based on decision trees. The performance of predicted traces is approximately 1.2 times the performance of predicted trees (column 1 and 3 of Table 1). There is code growth due to replication of code for forming traces. However, the performance gain is considerable to offset the disadvantage of code expansion. For long traces, the misprediction penalty will be high. Considering the fact that the intermediate checkpoints will be beneficial for such cases, long traces can be artificially split into smaller traces in accordance with the scheme. Moreover in embedded applications traces are not too long and this is true of the benchmarks compiled.

4 Conclusion

The performance of the VLIW processors can be improved considerably by dividing the application into number of traces and speculatively scheduling them according to their probability of execution (obtained by profiling the application). The performance obtained by us is approximately 1.417 times the original TriMedia performance using the scheduling scheme presented in this paper. We have shown that by annotating traces according to their probability of execution (obtained by profiling the application) and scheduling them according to this probability the number of mispredictions incurred is minimal.

References

[1] Jaime H. Moreno *et al.*, *"Scalable Instruction Level Parallelism through Tree Instructions"*, IBM Research Report, `http://www.research.ibm.com`

[2] Jan Hoogerbrugge *et al.*, "Instruction Scheduling for TriMedia," *Journal of Instruction Level Parallelism,* 1, 1999.

[3] Jan Hoogerbrugge, "Dynamic Branch Prediction for a VLIW Processor", In *Proceedings of the 2000 International Conference on Parallel Architecture and Compiler Techniques*, Philadelphia, PA, Oct. 2000.

[4] John R. Ellis, *"BULLDOG: A Compiler for VLIW Architectures"*, PhD thesis, Yale University, 1985.

[5] Peter Y. T. Hsu *et al.*, "Highly Concurrent Scalar Processing", In *Proceedings of the 13th International Symposium on Computer Architecture,* 14(2):386 - 395, Tokyo, June, 1986.

[6] Sanjeev Banerjia *et al.*, "Treegion Scheduling for Highly Parallel Processors", In *Proceedings of Euro-Par'97,* pp. 1074-1078, Passau, Germany, Aug. 1997.

[7] Scott A. Mahalke *et al.*, "Effective Compiler Support for Predicated Execution using the Hyperblock", In *Proceedings of the 25th Annual International Symposium on Microarchitecture,* pp. 45-54,Portland, Oregon, USA, Dec. 1-4, 1992.

[8] W. W. Hwu *et al.*, "The Superblock: An Effective Structure for VLIW and Superscalar Compilation", *Journal of Supercomputing,* pp. 229-248, 1993.