Speculative Trace Scheduling in VLIW Processors

Manvi Agarwal and S.K. Nandy
CADL, SERC,
Indian Institute of Science,
Bangalore, INDIA
{manvi@rishi.,nandy@}serc.iisc.ernet.in

J.v.Eijndhoven and S. Balakrishnan
Philips Research Laboratories,
Eindhoven, The Netherlands
{jos.van.eijdhoven,srinivasan.balakrishnan}@philips.com

Abstract

VLIW processors are statically scheduled processors and their performance depends on the quality of schedules generated by the compiler's scheduler. We propose a new scheduling scheme where the application is first divided into decision trees and then further split into traces. Traces are speculatively scheduled on the processor based on their probability of execution. We have developed a tool ***"SpliTree"*** *to generate traces automatically. Using dynamic branch prediction for scheduling traces our scheme achieves approximately 1.4x performance improvement over that using decision trees for Spec92 benchmarks simulated on TriMedia*[tm].

1. Introduction

"Very Long Instruction Word Processor", widely known as *"VLIW"* is a paradigm for simple hardware and high compute capacity. In a VLIW processor the micro-architectural details are exposed to the compiler and latter generates schedules to exploit maximum *Instruction Level Parallelism (ILP)* present in the code. 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 of exploiting ILP because of small size of basic blocks. (4-5 interdependent operations on an average in each basic block.) In extended basic block scheduling, groups of basic block scheduling can be categorized into following: *trace scheduling, superblock scheduling, hyperblock scheduling and decision tree scheduling.* All these scheduling schemes suffer from the drawback of issue slot wastage as explained later in the text of the paper.

In this paper we propose a new scheduling scheme which ensures minimal issue slot wastage. In our scheme, 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 carved out with the help of profile information of the application. Based on profile information, each trace is annotated with the probability of its execution. 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. Using dynamic branch prediction for predicting root of each trace our scheme achieves a gain in schedule length of the trace. The proposed speculative trace scheduling scheme minimizes the number of mispredictions by scheduling traces based on their probability of execution.

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. We summarize contribution of the work in section 4 and draw conclusions.

2. Related Work

Scheduling is the process of generating a sequence of micro-operations that provide appropriate control to the functional units for their execution. As mentioned earlier, two methods of scheduling are: *Basic block scheduling* and *Extended basic block scheduling*. Some of the

Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)
1063-6404/02 $17.00 © 2002 **IEEE**

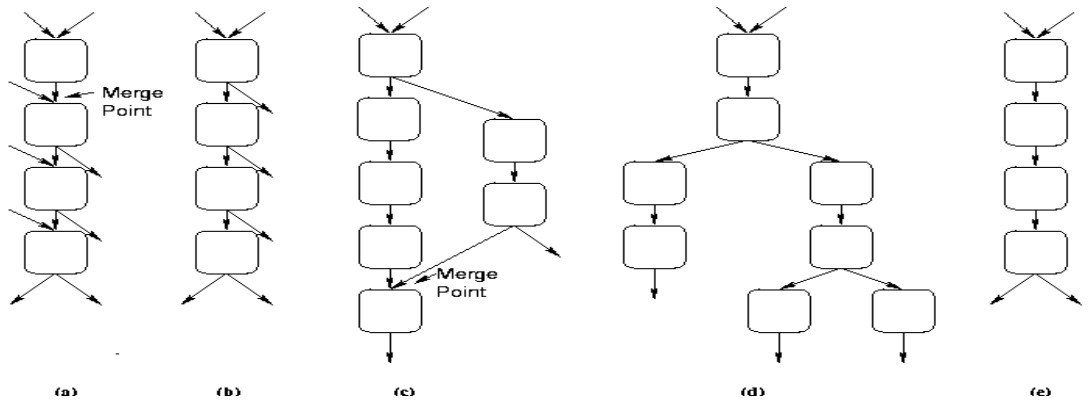

Figure 1. Types of Extended Basic Block Scheduling Scopes: (a) Trace, (b) Superblock, (c) Hyperblock, (d) Decision Tree and (e) Traces of Speculative Trace Scheduling

extended basic block scheduling techniques are *trace scheduling, superblock scheduling, hyperblock scheduling* and *decision tree scheduling*. The scheduling scopes used in these scheduling schemes are illustrated in figure 1.

In *trace scheduling* [3], 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 as a single unit. Side entries as well as side exits are allowed in traces because of which book-keeping of operations moved across basic blocks is required. In superblock scheduling overhead of book-keeping is obviated as elaborated below.

Superblock scheduling [8] is similar to 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 single entry point as opposed to trace scheduling which has multiple entry points. This scheduling scheme does not permit code motion past fork points, which renders book-keeping unnecessary and is 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 wrong path for execution based on profile information leads to wastage of processor cycles.

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* involves conditional execution of instructions based on the value of boolean operand which is known as *predicate*. As shown in figure 1, a 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 with 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, scheduler has to find appropriate operations to fill the delay slots of the branches. If it is unable to find these operations, it fills them with *"nops"*. Issue slots thus get wasted which otherwise could be used to schedule operations on the functional units. Due to the presence of control operations in a decision tree, required code optimizations cannot be performed because operations following a branch cannot be scheduled earlier than the branch. This can be evaded by using guarded or predicated execution in which control dependency is converted into data dependency with the help of predicates and the operations are scheduled as soon as their data dependency is met. Value of the predicate registers determines whether the result would be considered or discarded. Though efficient schedules can be generated with higher code density, effectively this leads to wastage of issue slots of VLIW processors. These issue slots could instead be used to issue operations along the correct path. We propose speculative trace scheduling scheme as an alternative 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, whenever we use

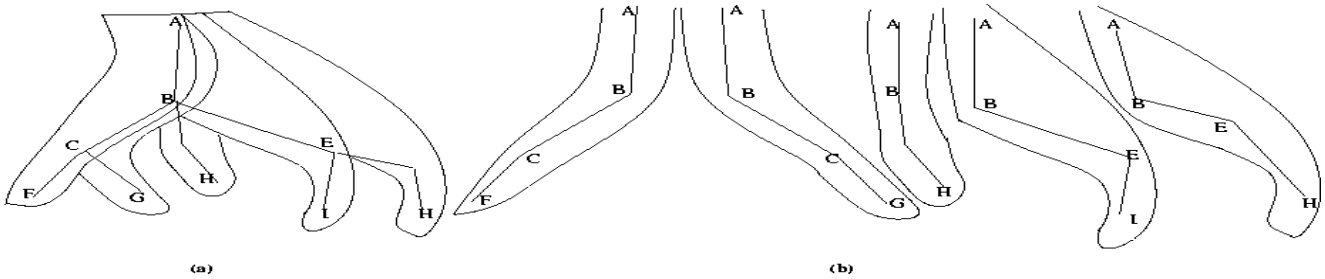

Figure 2. (a) A Decision Tree (b) Traces of the Decision Tree

the term *"trace"* we implicitly mean *"probable execution trace"*.

3. Speculative Trace Scheduling

In this scheduling scheme, the application code is divided into a number of traces i.e. probable execution traces of the application are formed. 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 generate efficient schedules. There are two phases in this scheduling scheme. 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 [2]. Decision trees are split into traces in the manner shown in figure 2. Path ABCF forms one trace as shown in the figure 2. Similarly all the possible paths in the decision tree are split into corresponding traces. The operations in a trace do not face ordering constraints during scheduling because of the removal of decision points from the body of the trace. 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. Since we factor in the branch direction as predicted by the branch predictor [1] no delay slots are allocated for branches. Gain in schedule lengths is achieved and code density is increased in the schedules generated by this scheme. As the check for correct execution of trace is done at the end of the trace, penalty paid on a trace misprediction is the length of the trace. The processor has to roll back to the previous checkpoint sate in the event of a misprediction with the help of additional hardware support [4]. 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 working registers are discarded and the state of the processor is retrieved from the shadow registers and execution of the next trace starts. 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.

```
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

3.1. Exploiting ILP

Schedules generated by our scheme have higher code density as compared to the schedules generated using decision trees. We explain this with the help of an example. Figure 4(a) corresponds to the decision tree generated by the compiler for the *"for loop"* in the pseudo code shown in figure 3. Figure 4(b) gives the corresponding scheduled code. Number in the parenthesis of *"if"* condition (figure 4(a)) shows the probability with which that condition is taken. As seen in figure 4(b), 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 taken into consideration or discarded. 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. Total number of processor cycles to execute this schedule is 6. As evident from the figure, last 3 cycles do not issue any operation but *"nops"* have to be included because of the branch operations, which have a

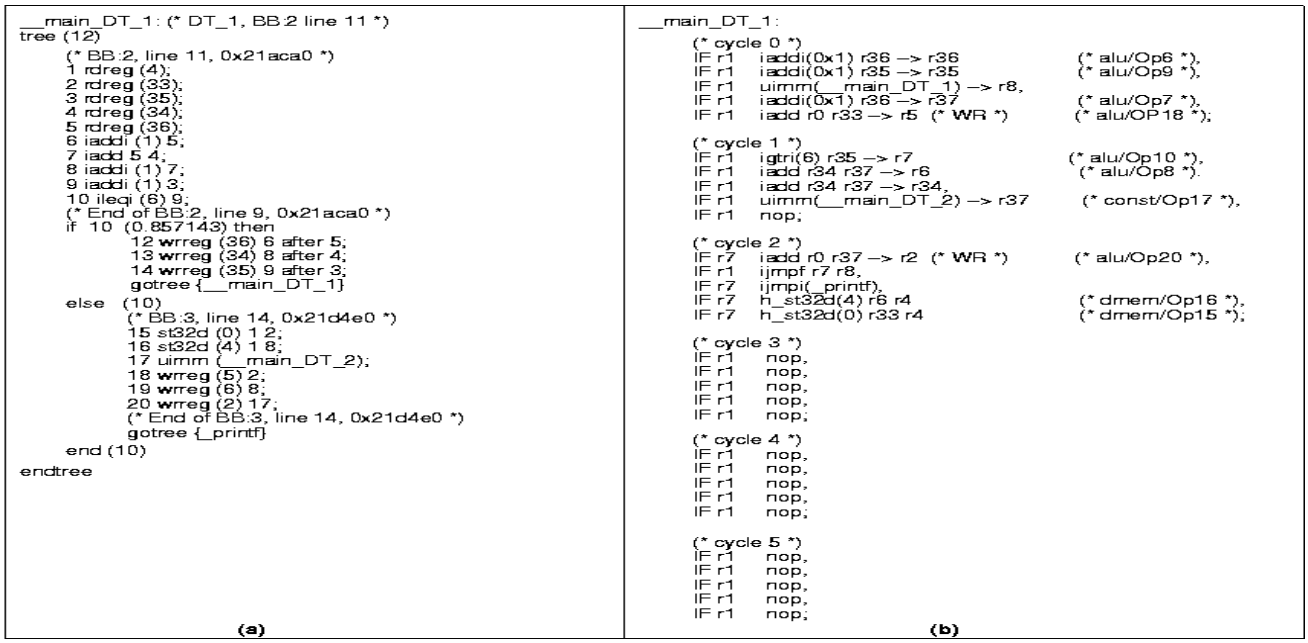

Figure 4. (a) Decision Tree as generated by the TriMedia compiler. (b) Schedule of the Decision Tree in (a) generated by the TriMedia Scheduler

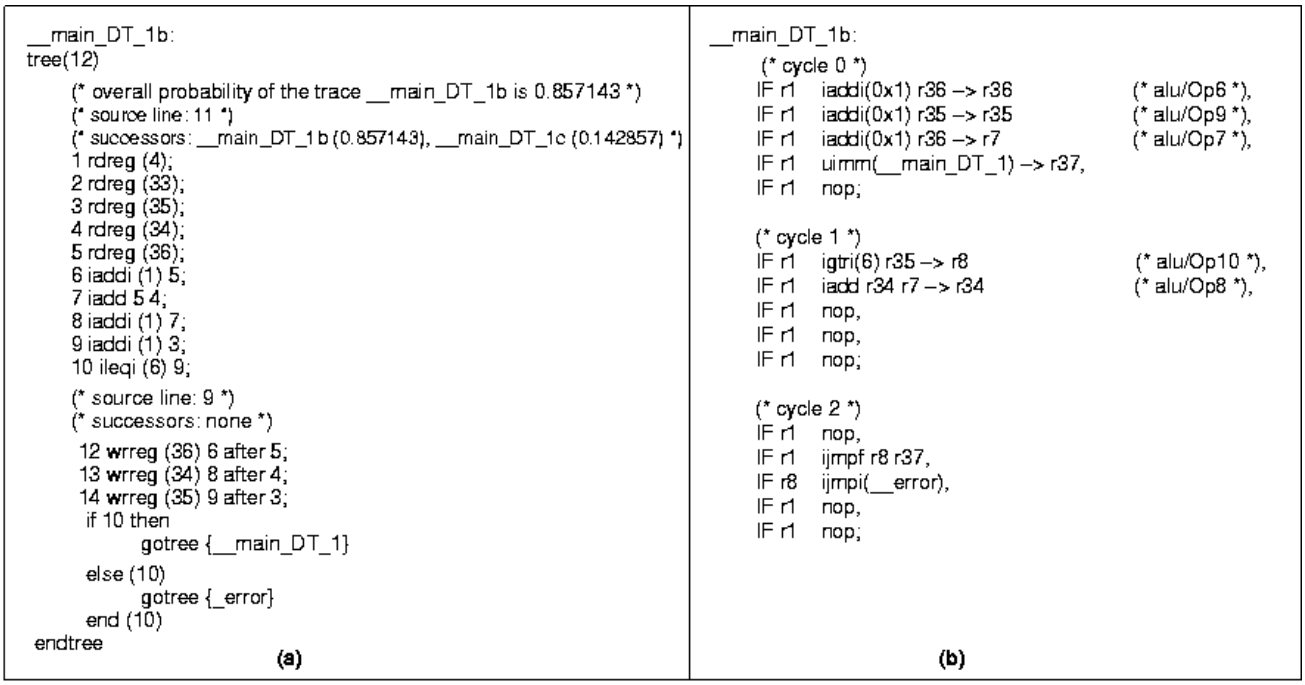

Figure 5. (a) Trace of the most probable path of the decision tree shown in figure 4(a) generated by *SpliTree*. (b) Schedule of the trace shown in (a) generated by the TriMedia scheduler using branch prediciton

delay of 3 cycles. 4 issue slots are seen wasted in cycle 2 of the schedule, which have register r7 as their predicate register. Figure 5(a) shows the trace of the most probable path of the tree shown in figure 4(a). Head of trace contains overall probability of the trace and does not have control operations in the body. Scheduled code of the trace in figure 5(a) is shown in figure 5(b). If 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 are no issue slots wasted in scheduling the second path of the decision tree (shown in figure 4(a)). As is apparent from the figures schedule length has shrunk by 3 cycles by using speculative trace scheduling.

	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

3.2. Simulation and Results

Simulation environment used for the project is Philips TriMedia SDE version 2.0, which is a Philips proprietary software tool. 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 execution of the trace based on the profile information (if available) obtained from the previous runs of the application. (For our simulation purposes we have used Philips proprietary input data to gather profile information.) 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). Most probable trace of the same tree is shown in figure 5(a) with its schedule in figure 5(b). Number of branch delay slots is 0 cycles in our schedule because dynamic branch prediction is employed to predict branch at the end of the trace when the trace is executed.

In order to bring out the efficacy 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}"* in Case 0 is given by:

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

where, *"L_{tree}"* is the schedule length of a tree and can be expressed by $L_{tree} = \sum_{path=1}^{n} L_{path} * p_{path}$. *"$L_{path}$"* 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}"* in Case 1 is given by:

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

where, *"MP_{tree}"* is the effective penalty for a 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. Execution time, *"ET_{trace}"* of the application in Case 2 is given by:

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

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 evaluated as:

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

where, *"MP_{trace}"* is the effective misprediction penalty of the trace and can be expressed as $MP_{trace} = R * MispredictionPenalty$. *"R"* is the next PC misprediction rate of the branch predictor and the 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, sufficient hardware [4](which is not present in TriMedia) is assumed 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 [1]. We have used the results of branch prediction from the paper of Jan [1], where the branch predictor predicts the direction of the branch along with the issue-slot that contains the taken branch. 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 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 processors. 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. Jan Hoogerbrugge has reported Case 1 results in [1] and we have reproduced them in this paper for the sake of comparison with our speculative tracescheduling scheme. Performance gain in the case of

Benchmark	Predicted Trees (Case 1)	Unpredicted Traces (Case 2)	Predicted Traces (Case 3)
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

Table 1. Performance Improvement relative to delayed branches in TriMedia for three cases: predicted branches in trees (branch delay is 0 cycles), split traces (no branch prediction i.e. branch delay is 3 cycles) and branch prediction in traces (branch delay is 0 cycles).

branch prediction is obvious considering the fact that branch delay slots are reduced to zero. Gain in performance is also achieved by splitting trees into traces without using branch prediction (branch delay slot = 3 cycles) as is evident from column 2 results in Table 1. This is 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 effectively. Column 2 results give the theoretical gain of trace scheduling over decision tree scheduling and have been produced here to show theoretical comparison of trace scheduling with decision tree scheduling. Decision trees with branch prediction 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 (column 1) and traces without branch prediction (column 2). 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 in the schedules. The performance achieved by our scheduling scheme is approximately 1.41 times the original TriMedia scheduling scheme, which is based on decision tree scheduling. 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. Although 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 multiple traces and using dynamic branch prediction for scheduling. By speculatively scheduling traces based on their probability of execution, the performance obtained by us is approximately 1.41 times the original TriMedia performance. 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] Jan Hoogerbrooge, "Dynamic Branch Prediction for a VLIW processor", In *Proceedings of the 2000 International Conference on Parallel Architecture and Compiler Techniques (PACT'00)*, pp. 207-216, Philadelphia, PA, Oct. 2000.

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

[3] John R. Ellis, *"BULLDOG: A Compiler for VLIW Architectures",* ACM Doctoral Dissertation Awards, MIT Press, Cambridge, Massachusettes, 1986.

[4] Manvi Agarwal *et.al.*, "Multithreaded Architectural Support for Soeculative Trace Scheduling in VLIW Processors", Accepted for *SBCCI 2002, 15th Symposium on Integrated Circuits and System Design*, Porte Alegre, RS, Brazil, September 9-14, 2002.

[5] Peter Y. T. Hsu *et al.,* "Highly Concurrent Scalar Processing", In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA-13)*, 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 (MICRO-25)*, 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.