Adaptively Speculative Execution for Wide-Issue Superscalar Processors

Chia-Chang Lin and Tien-Fu Chen

Department of Computer Science
National Chung Cheng University
Chiayi, Taiwan 621, ROC

chen@cs.ccu.edu.tw

Abstract

In the past, a scheme of Adaptive Branch Trees (ABT) has been proposed for adaptively keeping track of alternative branch paths and to speculatively execute the code on the most likely path with constrained hardware resources. In this paper, we combine the ABT concept with the instruction prefetch by realizing an ABT table to prefetch the most likely path of execution stream codes so as to reducing instruction cache miss penalty. Then, we focus on the speculative execution with ABT scheme. We will take the advantage of the property of ABT to exploit execution parallelism. We propose a register renaming mechamism to ensure that the values of registers are generated and accessed to keep data consistency and control dependency under dynamic out-of-order execution.

1 Introduction

Several architecture schemes combining various branch prediction mechanisms have been proposed to provide better accurate prediction for superscalar processor[6, 9, 1, 5]. To exploit execution streams as many as possible for parallelism is necessary, two known alternatives to Single Path (SP) execution streams have been proposed for reducing the ill effects of branch, i,e, *Eager Execution* (EE) and *Disjoint Eager Execution* (DEE)[8]. The first executes the code on both paths of a branch, bypassing the branch, giving a branch mispredict penalty of zero. Although, it results in the best performance, EE also has prohibitive cost because of the need of large hardware requirement to satisfy every branch path. Disjoint Eager Execution (DEE) is more promising one because it speculatively executes the code that is most likely to be need. So, the conception of DEE is between EE and SP. On one hand, it takes the advantage of dynamic instruction scheduling to exploit parallelism. On other hand, DEE's cost is also attractive. So, under constrained hardware resource, DEE indicates which code to assign to spare function unit for the best performance.

Chen [2] introduced an concept of Adaptive Branch Trees (ABT). Like DEE, the basic idea of the adaptive branch branch tree is to dynamically keep track of alternative branch paths and to speculatively execute the code on the most likely path. But the ABT is not a static tree heuristic, it can dynamically change its tree shape if a branch is resolved. Unlike branch prediction, ABT concept can provide multiple execution streams and reduce long branch mispredict latency. The concept of the ABT is realized by an adaptive branch tree table (ABTT) supporting multiple execution streams and correcting the update of the machine states.

In this paper, based on the idea of ABT, we extend the study to have two contributions. The first is that we use the ABT mechanism to support aggressive instruction prefetching for the instruction cache. We can reduce cache misses by aggressive prefetching instructions which may be useful in the future. We evaluate the performance of instruction prefetching scheme with ABT design and compare with SP(branch prediction) prefetching scheme. The second is to support speculatively execution of the code on the most likely path provide us with a good solution about multiple paths execution. We evaluate the performance of ABT speculative execution scheme and SP speculative scheme under constrained hardware resources.

The organization of this paper is following: Section 2 introduces the Adaptive Branch Tree concept. Section 3 gives the instruction prefetching via using the ABT scheme. Section 4 proposes the speculative execution which involves optimally adjusting instruction for parallel execution after a branch's two paths by ABT scheme. In Section 5, we present our experiment. Finally, we conclude in Section 6.

2 Superscalar with Adaptive Branch Tree

The basic ideas of adaptive branch Trees (ABT)[2] is to speculatively execute the code on the most likely path. The characteristics of ABT are attractive for speculative execution in that ABT can be adaptively extended and execution path sequence can be easily reset once a branch is resolved. For a given node in the tree, the likelihood fraction of either path is based in the taken probability of the branch, instead of prediction accuracy as in the DEE scheme. The concept of the ABT is realized by an adaptive branch tree table (ABTT) as shown in Figure 1. The function of ABT table is to record the dynamic execution flow from the viewpoint of branch path unit. In the background of execution, the branch

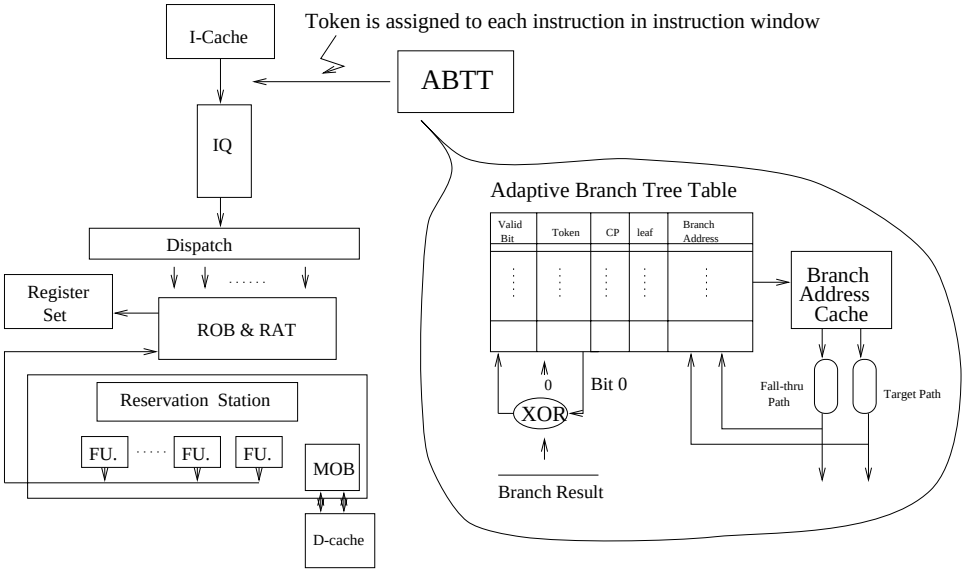


Figure 1. Machine with Adaptive Branch Tree Table

tree can be extended until there is no table entry is available. Each entry is assigned with an unique token, such that the entire tree can be reconfigured by shifting bits and invalidation operation once the root branch is resolved. The cumulative probability gives the cumulative likelihood of execution on this path. The leaf flag indicates whether the corresponding path is pending to be extended or hs been speculatively exploited,

There is a branch address cache (BAC), which cooperates with ABT table, to store branch target address. In the BAC, we also need an additional field to record the taken probability of the corresponding branch. The probability can be obtained at the profile-time or dynamically estimated at runtime. In our experiment, we use a profile to get the global taken probability of each conditional branch. Given a branch address, the BAC gives one fall-through path and one target path for speculative execution. Whenever the execution finds a branch on either path, its new branch addresses will be passed back to the ABT table. Note that the BAC is used for only storing target addresses without prediction capability. This is because the ABT table maintains some of alternative branches with larger likelihood, instead of predicting a single path that is likely to be taken or not.

The novel idea of ABT scheme is in the token assignment and the determination of valid bit. When a branch is resolved, all of the tokens will be shifted right by one bit and one of child nodes of the root will become the new root node. The detailed operations of ABT scheme can be found in [2].

3 Instruction Prefetch using ABT for Instruction Cache

Prefetching scheme is an efficient method to reduce the gap between processor and memory speeds. A prefetching cache generates prefetch requests to bring data in the cache before it is actually needed, thus allowing overlap with premiss computations. The instruction cache exists for supplying requested code to the instruction prefetcher. As shown in Figure 2, the prefetcher issues only read requests to the code cache.

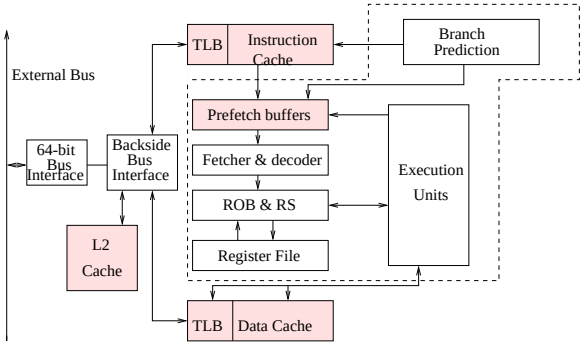


Figure 2. Processor Cache System Overview

Since the target line is usually not ready in time on a taken branch, the prefetched wrong-path line is expected to used only when execution returns to the branch. Another instruction prefetching about speculative execution is to investigate the various fetch policies for speculative execution[3]. Now, we will give an aggressive prefetching scheme, which prefetching instructions unit is branch path instead of cache line to match the requirement of deeper speculative execution.

As mentioned in [3], the processor starts fetching instruction along the wrong execution path when a branch misfetch or misprediction occurs. If an instruction cache is then encounter, two detrimental effects might arise from fetching the missing line: (i) the line on the wrong path may replace useful instructions in the I-cache. (ii) the bus between the IL1 and IL2 cache might be busy while an I-cache miss on the correct paths to be processed. Because the codes of successive branch paths of previous conditional branch might be the same with that of previous branch path, it is no doubt that the program with regular branch behaviors will per-

form well if we use branch prediction to prefetch multiple branch paths. However, the branch prediction mechanism is a history-based prediction method, it needs training time to complete acceptable prediction rate. So, the more deeper depth of the predicted branch is, the less prediction rate it is. When a branch misfetch or a branch misprediction occurs, and the demanded cache lines are not yet ready, the long latency du to cache miss will be detrimental to overall performance, especially under the environment with deeper speculative execution. Another shortcoming of multiple branch paths prefetching is that no execution on wrong path, the previous prefetched code may pollute the I-cache.

Oppositely, if the instructions prefetched on the wrong path will be used sooner than the displaced instructions, it is worth our effort prefetching the instructions of wrong paths. This scheme dynamically keeps track of alternative branch paths so that the code on the most likely path can be prefetched to I-cache for speculative execution. By applying ABT concept, we keep probing the existence of the codes in cache which belong to multiple pending branch paths according to ABT activity. Hence, we can greatly reduce access time and slightly reduce miss rate by ABT prefetching because branch misprediction will be very costly when deeper speculative execution is performed based on branch prediction.

Although aggressive instruction prefetching results in higher replacement rate and cache pollution than little-quantity prefetching, it really reduce the chance of cache miss which may result in long latency. If the prediction rate of branch prediction is higher, and the program behavior is very regular, the selected pending branch paths of branch prediction scheme will be similar to that of ABT scheme. Otherwise, ABT will take the advantage of early fetch the code of alternative branch paths.

4 Aggressively Speculative Execution for Wide-Issue Processors

We exploit more pending branch paths into instruction window for reducing branch misprediction penalty under constrained hardware resources. Therefore, ABT speculative execution scheme can be another choice. As shown in Figure 3, the machine model of ABT speculative execution is slightly modified from typical superscalar model. From a viewpoint of hardware resources, we need not to duplicate multiple instruction queue, to have too many execution units or larger instruction pool in execution core. ABT speculative execution scheme assigns a novel token to every instruction when this instruction enters the instruction window. By the characteristics of shifting operation (see [2]), the branch recovery operation can be simplified without too much system states maintenance or bus traffic (like Eager execution and Disjoint Eager Execution) when branch misprediction occurs. The following will describe the operations when a branch misprediction occurs in ABT speculative execution model:

- all instructions which are currently in the ROB and belong to the successive branch paths of the mispredict branch must be flushed from the ROB. (Maybe there are still pending instructions came after the branch in ROB.)

- all of the instructions which belong to the successive branch paths of the mispredict branch and are currently in the reservation station or are currently being executed must be flushed.

- all instructions in the earlier pipeline stages and belong to the successive branch paths of the mispredict branch must be flushed.

- instructions in the instruction queue and belong to the successive branch paths of the mispredict branch must be flushed.

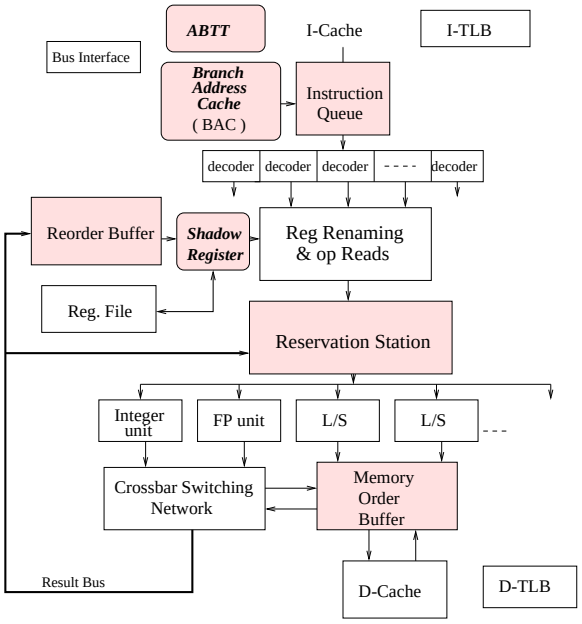


Figure 3. Superscalar Model for ABT scheme

The entries in the gray rectangular parts shown in Figure 3 are with a token as a field. By following the token assignment rule and re-configuration rule of ABT concept, the code flows in execution core are easy to be adjusted when a branch is resolved. However, ABT will make the instruction queue, reorder buffer, memory order buffer generate non-sequential empty slots because of random placement policy. This phenomenon results in the microarchitecture will need some modifications. Originally circular buffer of instruction queue, ROB, and MOB must be altered as a buffer with fully-associative mapping policy. Fortunately, it can be easy implemented by assigning a new tag to each newly allocated entry and a multiport content addressable memory (CAM) supported. When the lookup operation is needed, the result bus provide the result tag for matching when results become available.

4.1 Shadow Register Set

Another hardware issue is that we provide a branch address cache (BAC) to determine branch target address instead of the BTB of traditional branch prediction. ABTT request a lookup operation to BAC and then BAC provides the taken probability and target address to ABTT for fetching suitable target instruction to instruction window.

ABT speculative execution model also provides a shadow register set which is a subset mapping of the alias register in ROB. The primary function of shadow register set is to keep data consistency under the non-sequential branch path execution streams in the execution pool (ROB) and to keep the commit sequence in program order. Based on the regular nature of superscalar machine model, we only need to slightly modify some microarchitecture to match the ABT speculative execution behavior.

4.2 Keep Data Consistency

The major problem is that how the source operand can read form the latest operands to keep the correct program execution under this kind environment with non-sequential ROB entries. One function of reorder buffer is to rename the destination register to a unique tag identifier; i.e. register renaming. The result from a functional unit uses this unique tag to write to an allocated entry in ROB. Originally, ROB behaves like a circular buffer by shifting operation. Although more than one destination CAM cells in ROB can match a source register identifier, the value or tag desired comes from the cell matching the most current destination identifier, which is closest to the top of the reorder buffer. By shifting operation and matching mechanism of register identifier, it is trivial to keep data consistency in traditional single path code stream.

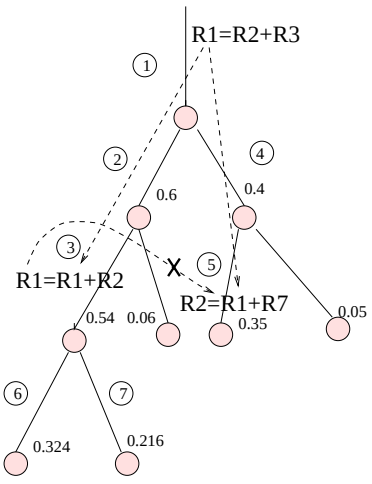


Figure 4. Branch path v.s. data consistency

In ABT speculative execution, however, the fetched instructions in ROB can be fragments from the viewpoint of fetched branch paths. As shown in Figure 4, The sequence of fetched instructions in ROB will be equal to the number of the branch path. (i,e, path 1, path 2, path 3, path 4, path5,....)

How can we ensure that the source operand R1 of path 5 is read from the latest destination operand R1 of path1? Fortunately, the characteristics of ABTT token encoding can solve the problem and help to construct a register renaming mechanism to keep data consistency.

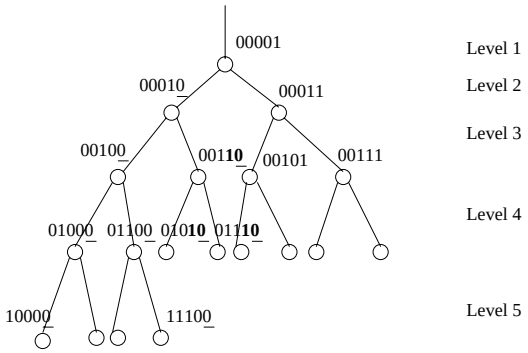


Figure 5. Specific token and its successive token

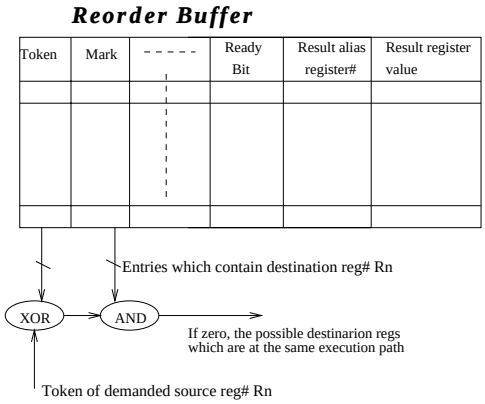


Figure 6. Block diagram of register reference

From the tree shape of pending branch paths as shown in Figure 5, we find that the least ($level_n$ - 1) bits of the token in ($level_n$) is the same with that of its subtree. Then, by judging the tokens and levels of branch paths, we can find out the overall possible source operands to read in the generated destination operand in ($level_n$) by simple exclusive-or operation logic of several bits.

In order to keep data consistency for using ABT speculative execution, the traditional register renaming mechanism must have some changes. According to the characteristics of specific token encoding rule we mentioned in Figure 5, we can develop a feasible register renaming mechanism to find out the latest generated destination register value. We will divide the renaming method into two parts as following:

- **Register assignment**

  Basically, the register identifier can be generated by token encoding. We add two additional fields (i,e, token bit string and mask bit string) to the entries of reorder buffer. The function of mask bit string is to filter the possible demanded registers which may be the

Table 1. Simulation Workload characteristics

Program	Instruction simulated	Static branch instr.#	Dynamic branch instr.#	Uncond Direct (%)	Cond Direct (%)	Call Direct (%)	Uncond Indirect (%)	Call Indirect (%)
Compress	200M	493	39238779	12.83	72.37	7.40	7.40	0.00
Gcc	200M	13573	41672372	6.64	79.31	5.62	8.13	0.30
Ijpeg	200M	1946	26651776	19.56	61.91	9.14	9.27	0.12
Li	184M	4157	14800459	12.54	57.97	12.14	17.07	0.27
Vortex	200M	10027	31944397	1.89	72.26	12.77	13.06	0.02
Applu	200M	1499	6663343	0.04	99.85	0.03	0.06	0.01
Apsi	200M	3025	10295477	4.39	82.87	5.6	6.55	0.59
Hydro2d	200M	1483	4175154	14.69	64.13	6.09	11.73	3.35
Tomcatv	200M	886	40353450	14.59	63.79	7.26	11.68	2.68
Turb3d	200M	1446	9654164	2.81	87.18	5.00	5.01	0.01

destination registers on the same execution path. However, only one value or tag among these filtered register identifiers can match the demand source destination. It means that the destination register identifier closest to the demanded register identifier is only desired. The primary principles of mask bits assignment are:

1. Every entry in ROB must have a mask field follow the token field.

2. For any branch path on the same level will follow the same mask encoding rule.

3. If a token with n bits can be labeled with $a_n a_{n-1} \ldots, a_1$, and this token is at the i^{th} level. Then, we label the mask bits as

$$\underbrace{000\ldots0}_{n-i}\underbrace{111..111}_{i-1}$$

As shown in Figure 6, we can select all possible destination registers for demanded source register by simple XOR and AND logic operations. If register R_n is the demanded source, we must have a lookup operation in ROB to select all entries which contain result register R_n and its ready bit corresponding to its register number. Then, the possible register values at the same execution path will be selected among all entries contain the same register name and its ready bit is set.

- **Keeping data consistency**

  Traditionally, the single path execution model make the instructions in ROB continuous. Although ABT execution model generate several empty slots because of squashing some entries due to branch recovery, it still keeps relative location of program order when instructions are dispatched into ROB. As a result, we can find the latest destination register value by combining the register renaming mechanism mentioned above with a propagation circuit when the lookup operation is done in the entire reorder buffer. If the instructions at the bottom always commit its result to the register file, we can start the lookup from the bottom of ROB. When a possible destination register value, we put the value to the propagation cell corresponding to the ROB entry. Otherwise, the value of previous propagation cell will be propagated to current propagation cell. When the look up operation is over, the latest destination register value will be find.

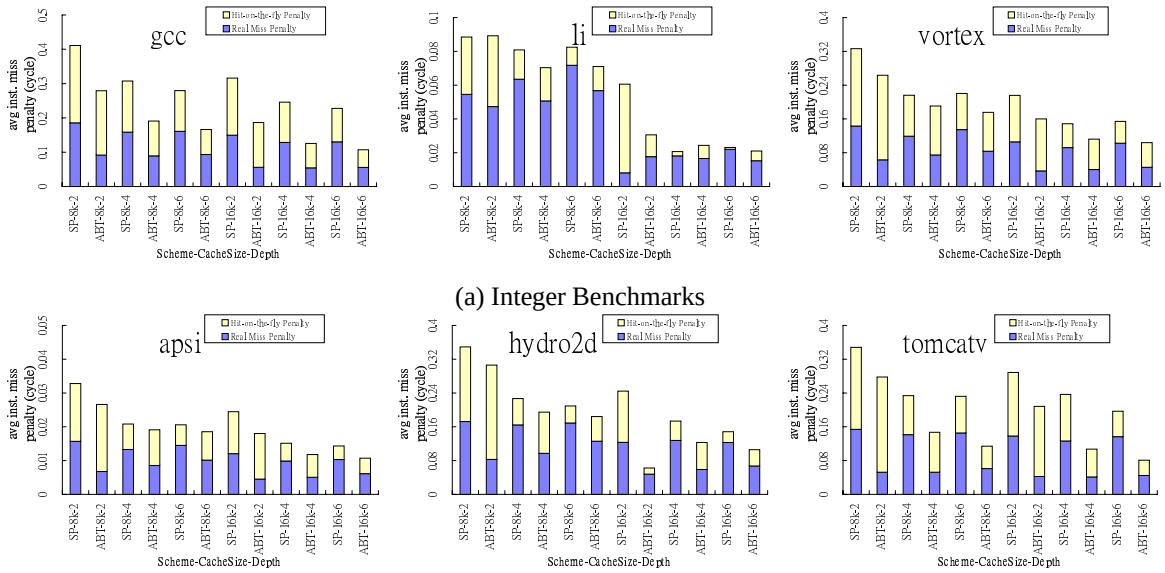
Then, we can combine the token field and the result alias register number in the ROB to find out the possible previous dependent operation, As shown in Figure 6, we retrieve the overall possible source of pending operand into the shadow register set, which is also a CAM cell with tag, token field. Then, we can find out the real source of pending operand by looking up the shadow register set. It's operation is similar to that of ROB.

Table 2. Hardware Configuration of ABT evaluation

Sym.	ROB/MOB /IQ#	Ialu/Imult /FPalu/FPmult#	Mem Port#	Issue Width
R32	32/16/8	6/2/6/2	2	8
R48	48/24/12	9/3/9/3	3	10
R64	64/32/16	12/4/12/4	4	12
R80	80/40/20	15/5/15/5	5	14
R96	96/48/24	18/6/18/6	6	16
R112	112/56/28	21/7/21/7	7	18

5 Performance Evaluation

We used execution-driven simulation environment to evaluate the performance of our proposed architectures and traditional branch prediction mechanisms by using SimpleScalar tool set[4], to evaluate the aggressive instruction prefetching scheme and speculative execution scheme. The out-of-order issue simulator simulates the detailed behaviors of pipeline stages. The out-of-order issue and execution scheme are



(a) Integer Benchmarks

(b) Floating Point Benchmarks

Figure 7. IL1 average instruction miss penalty (assume average L1 miss latency = 10 cycles)

based on the Register Update Unit (RUU)[7]. RUU, a combination of reservation stations and reorder buffer device, serves as a collection of ordered reservations stations and coopertes with reorder buffer. The reservation stations capture register results and await the time when all operands are ready, at which time the instruction is issued to the functional units. This RUU scheme also uses a reorder buffer to automatically rename registers and to hold the results of pending instruction.

The speculative execution was simulated programs from SPEC95 benchmark suite. Table 1 shows the basic statistics for the program we executed. We compare the performance of our proposed scheme with Single Path execution flow model with *PA*s branch prediction scheme. The primary factors to influence cache performance are hit ratio and access time. Hence, the primary metric we used in the experiment are miss ratio and miss penalty of IL1 cache.

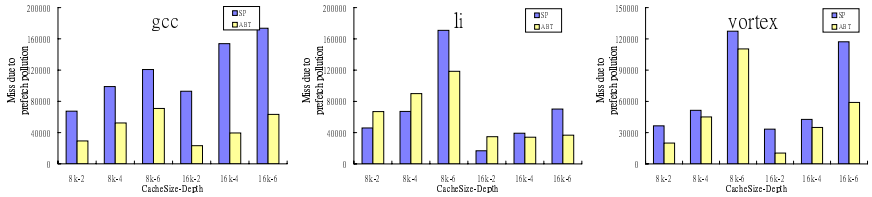
About our cache configuration, we give 2-way associative, LRU replacement policy with IL1 cache size of 8k, 16k bytes, and a 8k/8k bytes unified L2 cache. About the memory latency in the memory hierarchy, L1 ,L2 cache hit latency and memory access latency are 1, 6 ,18 in cycles, respectively. We will evaluate the ABT instruction prefetching scheme compared with SP scheme under varied prefetching path depth of 2, 4, and 6.

We compared the ABT speculative execution scheme with traditional single path speculative execution with branch prediction includes BTB size of 2048 entries. We defined several hardware resource configurations, which include the size or number of the reorder buffer, memory order buffer, instruction queue, integer functional units, float-point functional units, and issue bandwidth, which are as shown in Table 2. First, we give the ABT table size of 16, branch mispredict penalty of 7 cycles, and fixed register set of 32 integer registers, 32 floating-point registers, and several specific registers. Then, we evaluate the following metric terms: IPC(Instructions per Cycle), Average Issue Bandwidth Utilization, Wrong Path Complete Rate, Function Unit Utilization, Reorder Buffer Utilization.

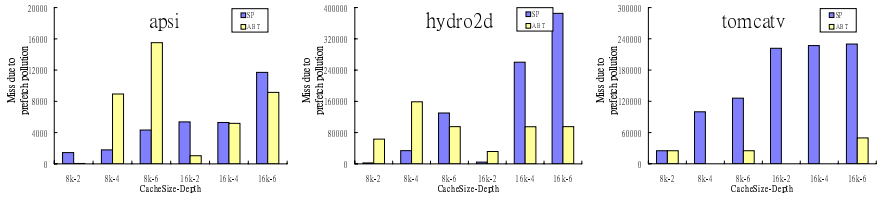
When pipeline is deeper, branch mispredict penalty is larger. We will compare the IPC of SP scheme with that of ABT scheme with an ABT table of 16 entries, varying the branch mispredict penalty from 7 to 15 for knowing the effect of mispredict branch penalty, Lastly, we attempted to exploit more pending branch paths by enlarging the entries of ABT table.

Figure 7 gives the total IL1 cache miss latency by calculating the number of real IL1 cache misses and the total cycle consumption of the IL1 hit-on-the-fly, given the average IL2 access latency of 10 cycles when IL1 real miss occurs. It shows the average latency of ABT and SP scheme with 8k, 16k bytes cache size when the number of prefetching depth varies. It give a breakdown of the two components to contribute to total miss ratio. From Figure 7, we can know that larger IL1 cache size is more feasible when more pending branch depths are prefetched. On the performance issue of total IL1 cache miss penalty, ABT scheme yields an improvement (decrease in total miss penalty) from 20% through 40% than SP scheme in most applications by prefetching the instructions of most-likely execution paths without applying standard branch prediction scheme. No matter what prefetching schemes are applied, the improvement in total miss penalty are 50% from 2 to 6 prefetching branch depths. Therefore, ABT prefetching scheme can be competitive with SP.

Figure 8 illustrates the number of IL1 cache cache misses which are due to the prefetching behavior. It means that the cache line will be hit originally, but the demanded cache line is replaced because prefetching. We named it as polluted block count. This metric gives the availability of cache blocks. We found that ABT scheme still offers an acceptable improvement in most application in our simulation.

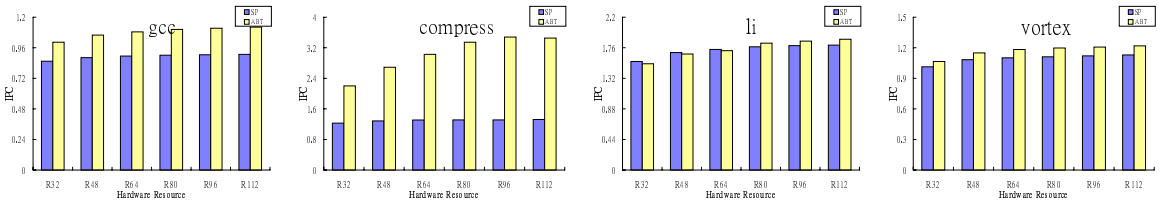


(a) Integer Benchmarks

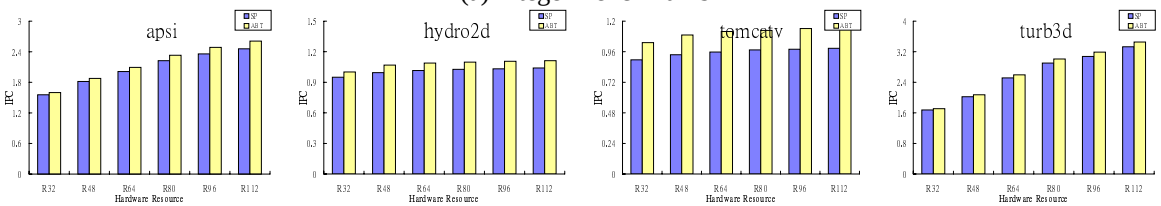


(b) Floating Point Benchmarks

Figure 8. IL1 Cache Miss Rate due to prefetching pollution (direct-mapped cache)



(a) Integer Benchmarks



(b) Floating Point Benchmarks

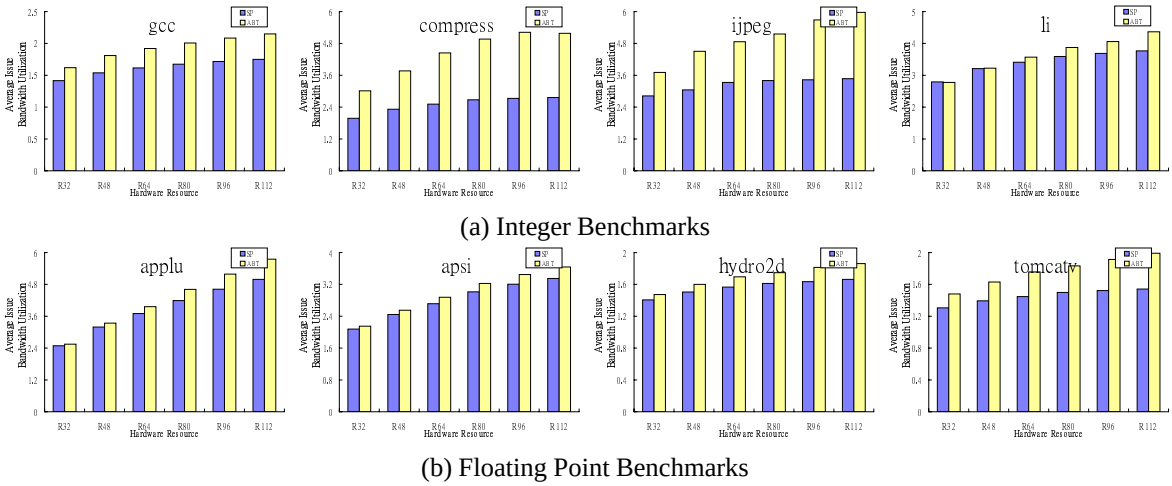
Figure 9. IPC comparison (Branch Mispredict latency = 7 cycles, ABTT entries = 16)

When more branch paths are prefetched, the impact of polluted blocks can be slightly worser when moving prefetching depths from 2 to 6. Except the apsi application, most applications perform better if ABT scheme is provided. We also found that the more regular the branch behavior is, the worser the cache pollution is when ABT prefetching is used. The apsi application demonstrates this condition due to taking in incorrect branch path's instruction into IL1-Cache when using ABT prefetching.

Figure 9 shows the IPC with varied hardware resource, given a fixed branch mispredict penalty of 7 cycles, and a fixed ABTT entries of 16. As shown in Figure 9, we found that ABT speculative execution achieve 7% to 290% improvement relative to SP scheme with hardware resource named R112 (see Table 5.2). The improvement of most applications is slight because the reduction of branch misprediction penalty is small. The program behaviors of some applications are different from that of others. For example, we found that the cycle consumption of some applications(compress, gcc, tomcatv) due to waiting for a previous branch to resolve because of the limited number of outstanding unsolved branches is larger when SP scheme is provides. However, ABT scheme can reduce this kind of waiting penalty and achieve larger IPC speedup because ABT scheme can explore more pending branches. Figure 9 also showed a phenomenon that the difference of IPC of ABT scheme and that of SP scheme is increasing larger when hardware resource is enlarged. The primary reason is because more incorrect prediction may occur in branch prediction(SP scheme) when a processor speculatively executes across a number of branches in a wide-issue or deeper pipeline processor.

As can be seen in Figure 10, the issue bandwidth utilization is proportional to the IPC value. However, it is obvious that the issue bandwidth utilization can not be fully exploited when SP scheme is supported in a wide-issue or deeper pipeline processor. Therefore, it is a good idea to use ABT scheme for increasing issue bandwidth utilization. In fact, executed program in SP scheme is greatly limited in data dependency, control dependency, or resource conflict even large amount of hardware resources are supported.

Since we consider a wide-issue processor, speculative execution may go up to a maximum level of speculative depth for unresolved branches. We limit the number of entries available in the branch tree table (ABTT) to have a constrained executed branch paths in the instruction window. As



(a) Integer Benchmarks

(b) Floating Point Benchmarks

Figure 10. Issue Bandwidth Utilization comparison (Branch Mispred. latency = 7 cycles, ABTT entries = 16)

shown in Figure 11, we assign the number of entries from 8 to 32 to observe the effect of ABTT entries. We found that the the IPC increase slightly when ABTT entries increase from 8 to 16. However, when we enlarge the number of ABTT entries which is larger than 16, we found that the IPC improvement has little or no different. Because of given the constrained hardware resource(which named R32, see table 5.2), the growth of pending instructions of speculative branch paths in the instruction window is also limited. In general, small number of ABTT size is enough. When hardware resources can satisfy more and more instructions executed smoothly, larger number of ABTT entries can be optional.

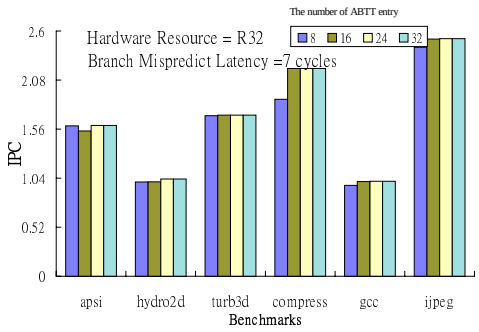


Figure 11. Effects of ABTT's Entry Number

6 Conclusion

In this study, we extend the concept of Adaptive Branch Tree (ABT)[2] in a combination with instruction cache prefetching and speculative execution. We apply the ABT concept for prefetching and speculative execution to reduce the two sources which hurt performance. Our goal is to design an effective and feasible speculative execution mechanism to dynamically keep track of alternative branch paths and to speculatively execute the code on the most likely path.

Our results indicate that ABT prefetching scheme is suitable for aggressive instruction prefetching when using non-blocking, wide memory bandwidth cache. We found that ABT scheme can reduce greatly branch misprediction latency to enhance performance. When larger hardware specification is supported, the speedup of ABT relative to SP scheme in some applications with large IPC value is also larger.

References

[1] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proc. of 21st Annual Symposium on Computer Architecture.*, pages 2–11, 1994.

[2] T.-F. Chen. Support highly speculative execution via adaptive branch trees. In *Proc. of the 4th Intl. Symposium on High-Performance Computer Architecture*, 1998.

[3] Brad Calder Dennis Lee, Jean-Loup Baer and Dirk Grunwald. Instruction cache fetch policies for speculative execution. In *Proc. of the 22nd Annual Intl. Symposium on Computer Architecture*, pages 357–367, 1995.

[4] Todd M. Austin Doug Burger and Steve Bennett. Evaluating future microprocessor: the simplescalar tool set. Technical Report TR-1308, Computer Sciences Department, University of Wisconsin-Madison.

[5] Ravi Nair. Dynamic path-based branch correlation. In *Proc. of the 28th International Symposium on Microarchitecture.*, pages 15–23, 1995.

[6] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, 1992.

[7] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computer. In *Proc. of IEEE Transactions on Computers*, pages 39(3):349–359, 1990.

[8] A. K. Uht. and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *IEEE Proceeding of MICRO-28*, pages 313–325, 1995.

[9] T. Yeh. and Y. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. of 20th Annual Intl. Symp. on Computer Architecture.*, 1993.