NCB: A Mechanism for Improving Instruction Fetching Efficiency

Toshinori Sato
Microelectronics Engineering Laboratory, Toshiba Corporation
toshinori.sato@toshiba.co.jp

A new instruction fetching mechanism for exploiting the instruction level parallelism (ILP) is presented. As a consequence of the trend toward increasing the number of functional units integrated in superscalar processors, the instruction fetching mechanism is being a bottleneck limiting the performance of the ILP architectures. This paper proposes a new technique named the *non-consecutive basic block buffer* (NCB). The NCB is an extension of the branch target buffer, and enlarges the effective instruction fetch bandwidth. From the experimental evaluation, it is found that the effective instruction fetch bandwidth is improved by approximately 5% and hence the ILP is increased by approximately 4% for integer workloads.

分岐先バッファを応用した命令フェッチバンド幅の向上

佐藤寿倫
株式会社東芝 マイクロエレクトロニクス技術研究所

スーパースカラープロセッサの命令供給効率を改善するための NCB を提案する。NCB は分岐先バッファを改良したもので、プロセッサのサイクルタイムに影響を与えないで、命令供給効率を改善する。NCB の同一ラインには複数の基本ブロックが登録されており、このため分岐命令を越えた命令を効率良く供給できる。SPEC92 ベンチマークを用いた評価では、整数系プログラムで平均約 5% の効率改善が得られた。その結果、命令レベル並列度が平均約 4% 向上した。

1 Introduction

As a consequence of the trend toward increasing the number of functional units integrated in superscalar processors, the instruction fetch mechanism is being a bottleneck limiting the processor performance. The increasing instruction issue width is no longer filled up because of the shortage of effective fetch bandwidth. For example, potentially 6 of the instruction level parallelism (ILP) is included in 8-issue out-of-order superscalar processors with a perfect instruction fetch mechanism[7], but in practice it can not be exploited. There are two obstacles reducing the effective fetch bandwidth. One is the instruction cache miss which has been studied in many papers. The other is the branch instructions disturbing the flow of control. This paper considers the latter one. Even though the branch outcome is perfectly predicted, the misalignment of the instruction run wastes the fetch bandwidth. As the instruction fetch width becomes longer, this undesirable effect becomes serious because there are at most 6 instructions in a basic block.

Several techniques attacking this problem[2, 3, 4, 17, 20] have been proposed. These techniques concurrently fetch the non-consecutive instruction runs across branch instructions using branch predictors, and carry out the aligning and merging of those instruction runs. Such a mechanism needs a hardware support which consists of the multiple branch predictor, the interleaved instruction cache, and the aligning and merging unit. The hardware is so complicated that it can easily increase the cycle time or must have the additional pipeline stages. The additional pipeline stages increase the branch misprediction penalty, and hence degrade the processor performance.

This paper proposes a new technique which is free from the drawback explained above. The technique named the *non-consecutive basic block buffer* (NCB) is an extension of the branch target buffer (BTB)[9] and utilizes the fill unit technique[11]. The NCB and instruction cache are accessed concurrently. If a group of non-consecutive basic blocks, the head of which instruction is indicated by the program counter (PC) is cached in the NCB and the branch outcome is predicted taken, the multiple basic blocks are fetched from NCB. The hardware of the NCB is as simple as the instruction cache, and therefore any additional pipeline stages are not needed.

The organization of the rest of this paper is as follows. Section 2 explains the proposed fetch mechanism using the NCB. In Section 3, the evaluation methodology is presented and the effect of the NCB is evaluated in Section 4. Section 5 surveys previously proposed related works. Finally, our conclusions are presented in Section 6.

2 Non-consecutive Basic Block Buffer Technique

The NCB is an extension of the BTB and utilizes the fill unit technique. Firstly in this section, we explain the BTB and next the fill unit mechanism. And finally, we present the NCB.

2.1 Branch Target Buffer

The BTB is a small cache memory associated with the instruction fetch stage of the pipeline[9]. Four types of information may be stored in an entry of the BTB: a branch instruction address tag, prediction information, a branch target address, and a target instruction[14]. The address tag is used for identifying

individual branch instructions. The prediction information is used for the decision of the branch outcome. The target address is an instruction address when the outcome is taken. And, the target instruction can reduce the cycles needed to fetch the instruction indicated by the target address.

In practical implementation, the BTB stores less information than four information described above. For example, MIPS R1000[21] has a 512-entry BTB, an entry of which consists of a 2-bit saturating counter. Therefore, the predictor can predict only the outcome of branch instructions. In addition, the tag-less BTB arises aliasing which decreases the prediction accuracy[23, 16]. The problem caused by the aliasing is beyond the scope of this paper, and our interest focuses on the BTB provided with tag.

One of the main ideas of the NCB is caching a number of instructions which follows the target instruction. If all instruction included in the target basic block is stored in the BTB, it becomes possible to fetch the target basic block simultaneously with the basic block indicated by the current PC. This improves the instruction fetching efficiency, because the instruction block which is fetched per cycle can be enlarged. The scheme is shown in Figure 1. An entry of the BTB shown in Figure 1 consists of the address tag, and a 2-bit saturating counter, and n target instructions.

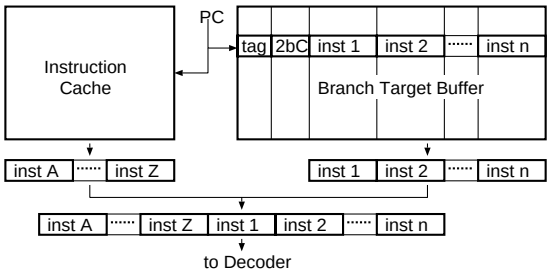

Figure 1: BTB Storing Target Instructions

However, this is not enough, because it is necessary to align and to merge the instructions consisting of two basic blocks, i.e. the fetch mechanism has to know the end of the basic block indicated by the current PC, and to combine two instruction runs. See Figure 1. If the basic block indicated by the current PC consists of instructions `A` to `Z`, there is not a alignment problem. Otherwise, the alignment problem arises. For example, let us imagine the basic block consisting of instructions `I` to `M`. In this case, it is necessary to quarry instructions between `I` and `M`, to merge them with the branch target instructions fetched from the BTB, and to shift the merged instructions in order to be aligned with the decoder slots. This process is shown in Figure 2. It is so complicated that it may increase the cycle time or must have the additional pipeline stages, and hence the processor performance is reduced.

2.2 Fill Unit

Originally, Melvin et al.[11] proposed the fill unit in order to combine a number of micro-operations into a larger atomic unit stored in the decoded instruction cache (DINC). The atomic units are dynamically scheduled and passed way to execution units. That is, the purpose of the fill unit is to enlarge the micro-operation level parallelism. Later, Franklin et al.[5] extended the fill unit approach to merge multiple RISC-style instruction into a VLIW-style instruction, which are stored in the DINC[1]. Figure 3 shows the diagram of the instruction issuing mechanism using the fill unit. The decoded RISC-style instructions are provided for the fill unit one by one, grouped into a VLIW-style instruction, and cached in the DINC. If the instructions indicated by the current PC are in the DINC, there is no need to decode and to schedule the instructions fetched from the DINC, and hence the issue complexity is reduced.

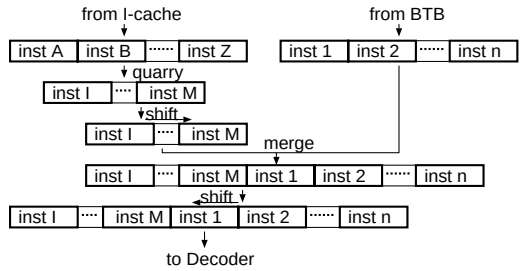

Figure 2: Basic Block Merging

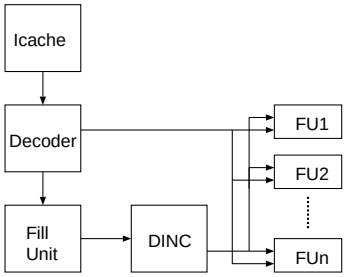

Figure 3: Instruction Issuing Mechanism with Fill Unit

In proposal by Franklin et al., the instruction merging is finished when a branch instruction is fetched. Figure 4 shows an entry of the decoded instruction cache line, each of which consists of instruction fields for individual functional units, the next instruction address field, and the branch target address field. The next instruction address field holds the address of instruction which immediately follows the instruction stored lastly in the DINC entry. The branch target filed holds the taken-branch address, when a conditional branch instruction is included in the entry.

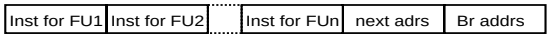

Figure 4: Format of Decoded Instruction Cache

Because of the merging criteria, instructions can not be grouped beyond the basic block boundaries,

[1] Franklin et al. referred it as shadow cache

and thus this limits exploiting the ILP. Another idea lying behind the NCB is storing instructions in the DINC beyond branch instructions. If the direction of the branch instruction stored in the DINC is much tendencious, it become possible to issue as many instructions as allowed by the number of functional units. In practice, however, the branch outcome is irregular and hence the correcting mechanism for instruction misissuing is needed. The ordinary correcting mechanism for branch misprediction is not applicable, because the instruction run is reconstructed into VLIW-style instructions and the order of instruction sequence is lost.

2.3 Non-consecutive Basic Block Buffer

The previous two sections briefly explain the scheme of the BTB and the fill unit, and introduce the main ideas regarding the NCB. Those are

1. Storing instructions following a branch target instruction in the BTB.
2. Merging instructions in the DINC beyond branch instructions.

Each of those ideas has a problem for performing effectively. The problems are

1. The mechanism for aligning and merging instructions consisting of multiple basic block is so complicated that it may degrade the processor performance.
2. The correcting mechanism for branch misprediction is inapplicable, because the instructions stored in the DINC are reconstructed into VLIW-style instructions.

This section presents the scheme of the NCB, which can solve the problems and improve the instruction fetching efficiency.

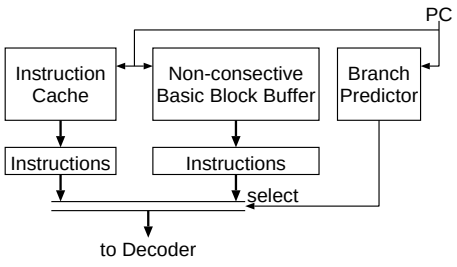

Figure 5: Instruction Fetching Mechanism with NCB

The NCB is an extension of the BTB discussed in Section 2.1, and caches not only the target basic block but also the basic block including the branch instruction in question, i.e. the NCB stores multiple non-consecutive basic blocks. This feature solves the first problem explained above. Figure 5 indicates the instruction fetching mechanism using the NCB. The instruction cache and the NCB are indexed by the instruction address indicated by the current PC. The branch predictor is used for selecting an instruction run from the instruction runs fed by the instruction cache and the NCB. Therefore, the predictor only predicts the branch outcome. It needs not hold the information about the target address. If the branch outcome is predicted taken, the instruction run from the NCB passes way to the decoder. Otherwise, that from the instruction cache is selected. In the case that the instruction runs starting with basic block indicated by the current PC is not cached in the NCB, the instruction run from the instruction cache is used. This process resemble that of selecting the next instruction address when the BTB is attached. In case of this, the next instruction address is selected between the PC incremented by an unit address offset and the target address from the BTB. If a branch is predicted taken, the target address from the BTB is selected. Otherwise, the incremented PC is used. In this mean, the NCB is regarded as the extension of the BTB.

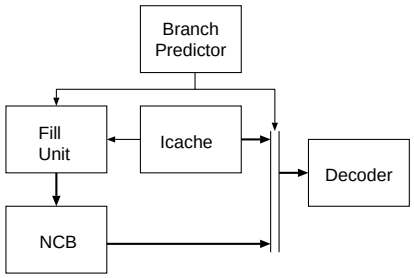

Figure 6: NCB with Fill Unit

Figure 6 shows the instruction storing scheme into the NCB. Different from the previously studied fill unit, the proposed mechanism utilizes the fill unit before decoding process, i.e. the NCB does not cache the decoded instructions but the original instructions. In addition, the instructions are cached in sequential order. This feature is a solution for the second problem described above, because the PC can be processed normally even if the instruction runs from the NCB is selected. In other words, the decoder knows the address of the branch instruction located in the basic block boundary, and thus the branch misprediction can be corrected. Figure 7 shows the instruction merging process executed in the fill unit. The fill unit temporary buffers instructions from the instruction cache (Figure 7a). If there is a branch instruction in the instruction run and the branch outcome is predicted taken, the instruction run reaching the branch instruction is stored in the line buffer (Figure 7b). When next instruction run from the instruction cache is fetched, the former instruction run stored in the line buffer and the latter instruction runs are combined into an enlarged instruction run, i.e. two basic block are merged (Figure 7c). The instruction merging process is finished when two basic blocks are stored in the line buffer or when the line buffer is full. The combined instruction run is forwarded to the NCB which stores it with indexing by the starting basic block address. Figure 8 shows the format of the NCB line. It consists of the tag address field, a number of instructions field, and the next instruction address field. The tag address field stores the starting basic block address. The instructions field holds the merged instructions. The last filed indicates the next instruction address which is similar to the

branch target address in the BTB.

From the above explanation, it can be seen that the problems explained at top of this section are solved by

1. caching multiple non-consecutive basic blocks in the NCB

   and

2. caching original instructions in the sequential order through the dynamic execution

and the instruction fetching efficiency is improved by the basic block merging.

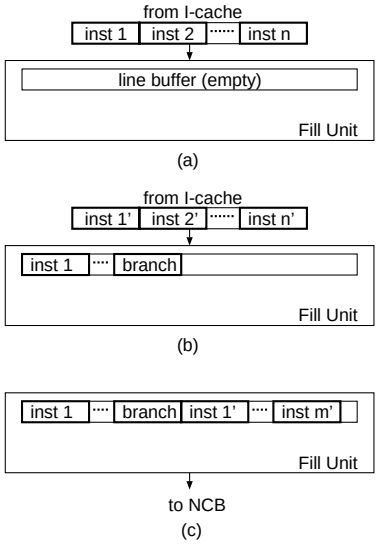

Figure 7: Instruction Merging Mechanism

3 Evaluation Methodology

In this section, we describe the evaluation methodology by explaining a processor model and benchmark programs.

3.1 Processor Model

We evaluated the effect of the proposed mechanism by using the SimpleScalar tool set (version 1.0.2)[1]. The SimpleScalar architecture is based on the MIPS architecture[8], and the cycle-by-cycle simulator is executed on a SPARCstation.

The baseline model is an out-of-order issue superscalar processor based on the register update unit (RUU)[19]. Following the discussion explained in [7], we decide the configuration of the baseline processor summarized in Table 1.

The interleaved sequential scheme[2] is used. The instruction cache is interleaved into two banks and prefetched one sequential block in advance. Therefore, the consecutive eight instructions can always be fetched simultaneously. The instruction fetching beyond branch instructions is prohibited. When the decoded instructions can not be queued in the RUU because of its capacity, the pipeline stalls instruction fetching.

We choose a two level adaptive branch predictor using the gshare scheme[10]. The predictor performs speculatively, i.e. the predictions may be based on the former branch predictions whose outcomes have not be resolved. Our scheme updates the branch history register (BHR) only, because the implementation is easy not only on simulation platform but also on processors in practice. Our preliminary simulation resulted in that there was not significant difference between the accuracy of the predictor speculatively updating both the BHR and the pattern history table (PHT) and that of the predictor speculatively updating the BHR only. When a prediction fails, the BHR is corrected. The index scheme for the BTB uses the starting address of the basic block in which the branch instruction in question is included[22].

Table 1: Baseline Processor Configuration

Fetch Width	8 instructions, interleaved sequential
Branch Predictor	512 set, 2way set-associative BTB, gshare scheme, 12-bit BHR, 4096 entry PHT, 3 cycle miss penalty
Issue Width	8 instructions, 128 entry RUU
Functional Units	5 iALU's, 1 iMUL/DIV, 2 ld/st units, 2 fALU's, 2 fMULs, 2 fDIV/SQRT's
FU Latency (total/issue)	iALU 1/1, iMUL 3/1, iDIV 35/35, Ld/St 2/1, fADD 2/1, fMUL 3/1, fDIV 6/6, fSQRT 6/6
Register Files	32 32-bit fixed point registers, 32 32-bit floating point registers
I-Cache	64K 4way set-associative, 32 byte blocks, 2-port, 6 cycle miss penalty
D-Cache	64K 4way set-associative, 32 byte blocks, 2-port, write-back, non-blocking load, hit under miss, 6 cycle miss penalty
L2 Cache	ideal

The evaluated model is as follows. The NCB is constructed by a 512 set, 2way set-associative cache, because it is replaced with the BTB. Its line size is chosen to be equal to the issue width, i.e. 8 words. This means the NCB is as large as 32K byte storage.

3.2 Workload

The SPEC92 benchmark suite is used for this study. The reference input files which are provided by SPEC are used with slight modifications. Table 2 shows the summary. The Fortran programs were converted to C programs using AT&T F2C (version 1994.11.03), and then all programs were compiled by GNU GCC (version 2.6.3) with the optimization option, `-O3`. Each program was executed to completion or for the first 1 billion instructions.

Table 2 also indicates the characteristics of each benchmark program. Each column shows the dynamic instruction count completed, the dynamic branch instruction count, and the average size of basic blocks, respectively. Note that the statistics in the table do not include the cycles executed by the operating system.

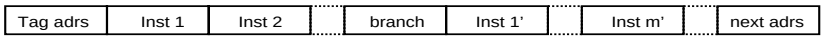

Figure 8: Format of NCB

Table 2: Benchmark Programs

Benchmark	Input	Modification	inst(mil.)	br(mil.)	BB size
008.espresso	bca.in		450.2	81.6	5.52
022.li	li-input.lsp	short input	1000.0	219.3	4.56
023.eqntott	int_pri_3.eqn		1000.0	335.9	2.98
026.compress	in		664.4	14.7	4.52
072.sc	loada3		363.4	78.5	4.63
085.gcc	cexp.i		19.9	4.0	4.96
013.spice2g6	greycode.in	short input	1000.0	94.5	10.58
015.doduc	doducin		1000.0	87.6	11.42
034.mdljdp2	mdlj2.dat	MAX_STEPS=250	1000.0	64.2	15.57
039.wave5			1000.0	130.4	7.67
047.tomcatv		N=129	480.8	18.4	26.20
048.ora	params	ITER=15200	116.8	9.1	12.85
052.alvinn		NUM_EPOCHS=50	1000.0	98.2	10.19
056.ear	args.short		420.5	39.3	10.67
077.mdljsp2	mdlj2.dat	MAX_STEPS=250	1000.0	68.2	14.67
078.swm256	swm256.in	ITMAX=120	1000.0	22.5	44.46
089.su2cor	su2cor.in	short input	766.2	28.1	27.30
090.hydro2d	hydro2d.in	short input	11.2	1.3	8.62
093.nasa7			1000.0	27.5	36.39
094.fpppp	natoms	short input	1000.0	13.5	74.14

4 Experimental Results

This section discusses the experimental results. Firstly, we evaluate the improvement of the instruction fetch bandwidth. And next, we present the performance improvement gained by the NCB.

4.1 Fetch Width

Figure 9 shows the instruction fetch bandwidth of the baseline model and the model attached with the NCB. For each group of four bars, the first bar (see from left to right) indicates the effective fetch bandwidth of the baseline model. Only committed instructions are considered for counting the effective bandwidth. Next bar indicates the total fetch bandwidth of the baseline model. The total bandwidth means that the fetched instructions includes squashed instructions which are issued during the mispredicted path. The remaining two bars present the effective and total instruction fetching bandwidth of the model with the NCB.

For integer programs, the effective bandwidth is increased by 4.8% on average and a maximum of 9.4%. The total bandwidth is increased by 17.0% on average and as much as 29.8%. The reason why there is a big difference between the effective and total bandwidths is poor branch prediction accuracy. Tables 3 and 4 present the statistics for the simulation results, consisting of the total executed instructions including the squashed instructions, the execution cycle time, and the percentage of correct branch prediction. Two results for the branch prediction accuracy is presented, one is for the prediction of branch outcome and the other is for that of the target instruction address. Note that the statistics for the total executed instructions and the execution cycle time are divided by one million. As can be seen at Tables 3 and 4, the branch prediction accuracy is at most 90%, and then the enlarged fetch bandwidth is wasted the squashed instructions. Therefore, the improvement of the effective bandwidth is small, in spite of that the total bandwidth is increased much. This results encourage us, however, because it can be possible to increase the effective bandwidth as much as the total bandwidth when the branch prediction accuracy is improved.

Compared with the integer program cases, for floating point programs the improvements of both effective and total bandwidth are much smaller. The effective bandwidth is increased by 2.0% on average and a maximum of 7.8%, and the total bandwidth is increased by 4.0% on average and a maximum of 12.2%. This is because the average sizes of basic block of the floating point programs are quite larger than those of the integer programs. From Table 2, it is larger than eight of the fetch width of the processor model, and is as much as 74 instructions in case of fpppp. Thus, almost the time, the instruction fetch bandwidth is effectively utilized for the floating point programs. This is confirmed by the result that wave5 whose average basic block size is smallest among the floating point programs marks the highest improvement of both effective and total fetch band width. From the above investigation, we expect that the NCB is useful for processors whose fetch width is at least sixteen.

4.2 Performance Improvement

Figure 10 shows the instruction per cycle (IPC) for two models. It also shows the IPC of the processor with a perfect instruction fetching mechanism consists of the

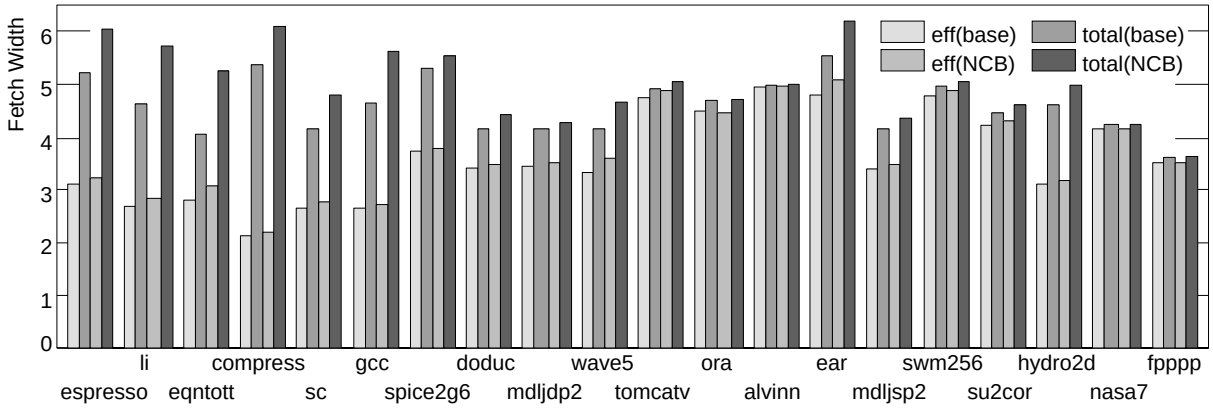

Figure 9: Simulation Result (Fetch Width)

Table 3: Simulation Results for Baseline Model

Benchmark	total inst	exec. cycle	br pred. correct outcome	br pred. correct target
espresso	757.1	164.6	96.4	92.1
li	1727.3	463.1	93.2	86.3
eqntott	1440.1	461.1	95.6	89.5
compress	167.7	37.4	90.4	85.8
sc	572.5	162.0	95.6	89.7
gcc	35.0	9.6	90.7	84.1
spice2g6	1418.7	287.5	96.4	92.7
doduc	1220.4	327.6	95.9	89.6
mdljdp2	1200.8	298.5	95.9	95.0
wave5	1240.8	373.7	97.3	80.8
tomcatv	495.0	102.1	99.0	98.1
ora	122.6	26.8	98.8	97.1
alvinn	1007.0	204.8	99.7	99.5
ear	484.5	95.0	95.7	93.5
mdljsp2	1220.7	303.7	95.7	94.8
swm256	1031.7	216.4	98.7	88.3
su2cor	810.2	190.2	96.8	90.1
hydro2d	16.5	4.0	93.3	88.4
nasa7	1015.1	240.4	99.2	98.8
fpppp	1034.3	291.0	94.9	90.3

Table 4: Simulation Results for Model with NCB

Benchmark	total inst	exec. cycle	br pred. correct outcome	br pred. correct target
espresso	845.8	159.4	96.4	92.1
li	2015.9	443.0	93.2	86.3
eqntott	1709.0	430.5	95.6	89.5
compress	185.0	36.7	90.4	85.8
sc	629.9	155.3	95.6	89.7
gcc	41.1	9.2	90.7	84.1
spice2g6	1464.4	284.4	96.4	92.7
doduc	1261.9	313.1	95.9	89.6
mdljdp2	1214.2	294.8	95.9	95.0
wave5	1291.6	352.0	97.3	80.8
tomcatv	498.8	99.9	99.0	98.1
ora	123.4	26.9	98.8	97.1
alvinn	1008.2	203.5	99.7	99.5
ear	512.3	90.3	95.7	93.5
mdljsp2	1241.9	296.0	95.7	94.8
swm256	1035.8	213.3	98.7	88.3
su2cor	824.8	186.7	96.8	90.1
hydro2d	17.4	4.0	93.3	88.4
nasa7	1016.6	240.5	99.2	98.8
fpppp	1037.3	289.1	94.9	90.3

perfect instruction cache, the perfect branch predictor, and the perfect aligning and merging mechanism. The perfect aligning and merging mechanism can fetch instructions until the fetch width is full.

As can be seen at Figure 10, the IPC for the integer programs is increased by 4.3% on average and a maximum of 7.1%. Comparing Figures 9 and 10, it is found that the more the improvement of the fetch bandwidth is, the more the IPC is increased. Therefore, it is confirmed that the NCB is useful to exploit the ILP by improving the instruction fetching efficiency. In addition, it is important to note there is a big difference between the performance of the model which have the practical instruction fetching mechanism and that of the model having ideal instruction fetching mechanism. One of the strict obstacles which limit the performance is the branch prediction accuracy. As discussed in previous section, the instruction fetch bandwidth is not utilized effectively because of branch misprediction. It becomes possible to exploit four of the ILP, if the branch prediction accuracy is improved.

Next, let us discuss the simulation results for the floating point programs. The IPC is increased by 2.1% on average and a maximum of 6.2%. This is coincident with the investigation above that the more the improvement of the fetch bandwidth is, the more the IPC is increased. As we have already discussed, there is little room for improving the instruction fetching efficiency because the average basic block size of the floating point programs are quite large. It is interesting that there is not significant difference between the ideal performance for the integer programs and that for the floating point programs. As we have seen at Table 2, the average basic block size of the floating point programs is much larger than that of the integer programs. From this results, we expected that the ILP for the floating point programs was more exploited than that for the integer programs. However, the fact is not. Furthermore, the difference between

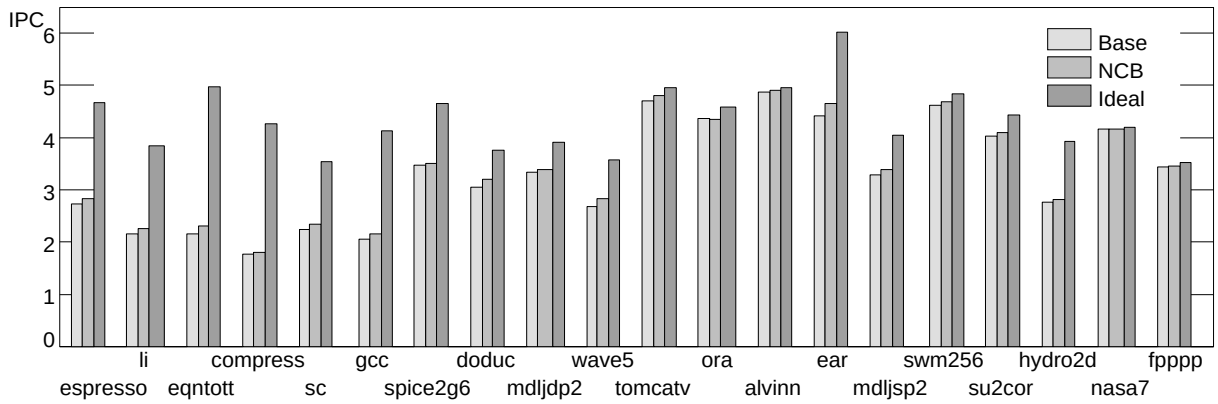

Figure 10: Simulation Results (IPC)

performance of the practical models and that of ideal model is also small. These are due to the resource shortage. The floating point unit has longer latency and less throughput that the integer unit, and thus the floating point instructions often can not be issued.

5 Related Work

There are several proposals for improving the fetch bandwidth. In this section, we survey those works and compare them with the NCB.

The collapsing buffer[2] aligns and merges two basic blocks in a same instruction cache line with the help of multiple branch predictors. Therefore, the effective fetch bandwidth is at most the instruction line size. Furthermore, because it relies on the interleaved branch predictor, there may be additional pipeline stages for instruction fetching, which causes the performance degradation.

In the two-block ahead branch predictor proposed by Seznec et al.[17], the current PC is not used for predicting the address of next basic block, but for predicting the address of the block following the next basic block. The predicted address and the current PC are used on the next cycle to fetch two non-consecutive basic blocks. In practical implementation, the two-ahead branch predictor includes the dual-ported BTB, which may slow the cycle time of the pipeline.

Wallace et al.[20] proposed the dual branch target buffer (DBTB) consisting of two BTBs. The former BTB is indexed by the current PC, and the latter BTB is indexed the branch target address given from the former one. Hence, two non-consecutive basic blocks in different cache lines are fetched simultaneously. The shortcoming of the DBTB is a chain of two branch predictors, which has a serious impact on the cycle time.

Dutta et al. proposed the block-level prediction[3], and extended it to the tree-like subgraph-level prediction[4]. Instead of predicting the outcome of each individual branch, this scheme performs the multiple branch predictions per cycle, which are hidden in a single prediction. Therefore, it becomes possible to fetch multiple non-consecutive basic blocks explained as a tree-like subgraph. The drawback of this scheme is its high hardware cost for presenting the control flow graph.

These mechanisms described above need complicated hardware support, which easily increase the cycle time or must have the additional pipeline stages. The additional pipeline stages increase the branch misprediction penalty, and hence degrade the processor performance. On the other hand, the NCB dose not have any impact on the pipeline cycle time.

The fill unit mechanism was first proposed by Melvin et al.[11], and extended by Franklin and Smotherman[5] to reconstruct RISC instructions into a VLIW-style instruction stored in the shadow cache. The purpose of the shadow cache is to eliminate the complex dependency checking logic. Smotherman and Franklin[18] also investigate to utilize the fill unit for the purpose of improving CISC instruction decoding performance.

Intrater et al.[6] proposed the DINC for variable instruction length processors, and investigated its structure. Each entry in the DINC contains a decoded instructions, a branch condition, and a branch target address, and is similar to the entry of the NCB. However, the purpose of the DINC is to reduce control hazards in pipeline, and the improvement of fetch bandwidth is not considered.

Hyperscalar processors[12, 13] have an additional architecture visible register file named the decoded instruction register file (DIRF). The DIRF are provided for every functional unit. The instructions dispatched to the combined functional unit are loaded to the DIRF by a special load instruction and reconstruct a VLIW-style instruction. Therefore, hyperscalar processors realize high dispatch width with relatively low fetch bandwidth. However, the DIRF is useful only for small instruction runs which are iterated heavily, such as `DO` loop. The drawback of hyperscalar architecture is the overhead of loading instructions to the DIRF.

The main difference between the NCB and the shadow cache, the DINC, and the DIRF is that the goal of these structures. Any of them except the NCB are not interested in enlarging the instruction fetch bandwidth.

The trace cache[15] is similar to the NCB. It caches instruction runs each of which consist of multiple basic blocks. As a result, a number of basic blocks are combined one large block, and the effective fetch band-

width may be enlarged. In order to fetch instructions, the trace cache needs the help of multiple branch prediction. They evaluated the trace cache under the impractical model, including the infinite functional unit resources, the instruction window with 2048 entries, and 16-way interleaved BTB.

6 Concluding Remarks

In this paper, we have proposed the NCB for improving the instruction fetching efficiency. The NCB is an extension of the BTB and utilizes the fill unit. It caches multiple non-consecutive basic blocks, and thus the effective fetch bandwidth is enlarged.

From the simulation results, we have found that for the integer programs the effective fetch bandwidth is increased by 4.8% on average and a maximum of 9.4%. It has also been found that the total fetch bandwidth is increased by 17.0% on average and a maximum of 29.8%. This implies the possibility that the effective fetch bandwidth might be improved further, if more accurate branch predictor is studied. This gave us the confidence that the NCB is useful for improving instruction fetching efficiency. The IPC is increased by 4.3% on average and a maximum of 7.1%. We have found that the more the improvement of the fetch bandwidth is, the more the IPC is increased. This means the branch prediction accuracy is a key factor also for exploiting the ILP.

In the case of the floating point programs, the NCB is less useful than the case of the integer programs. The effective bandwidth is increased by 2.0% on average and a maximum of 7.8%, and the total bandwidth is increased by 4.0% on average and a maximum of 12.2%. This is because the average basic block size of the floating point program is quite large, and hence there is little room for increasing the effective fetch bandwidth when the instruction fetch width is eight. The IPC is improved by 2.1% on average and a maximum of 6.2%. The obstacles limiting the ILP is the resource shortage. For the next generation processors which might have sixteen of the instruction fetch width and the plenty of floating point units, we expect that the NCB is effective for improving the instruction fetching efficiency and for exploiting the ILP.

Future study dealing with the NCB will investigate the timing analysis with the gate-level implementation for the NCB mechanism, especially the complicated fill unit. Evaluating the trade-off could appear that the NCB were practical.

References

[1] D.Burger, T.M.Austin, S.Bennett, "Evaluating Future Microprocessors: the SimpleScalar Tool Set", Technical Report 1308, Computer Science Department, University of Wisconsin Madison, WI, 1996.

[2] T.M.Conte, K.N.Menezes, P.M.Mills, B.A.Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", Proc. of 22nd Ann. Int'l Symp. on Computer Architecture, pp.333-344, 1995.

[3] S.Dutta, M.Franklin, "Block-Level Prediction for Wide-Issue Superscalar Processors", Proc. of 1st Int'l Conf. on Algorithms and Architectures for Parallel Processing, pp.143-152, 1995.

[4] S.Dutta, M.Franklin, "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors", Proc. of 28th Ann. Int'l Symp. on Microarchitecture, pp.258-263, 1995.

[5] M.Franklin, M.Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue", Proc. of 27th Ann. Int'l Symp. on Microarchitecture, pp.162-171, 1994.

[6] G.D.Intrater, I.Y.Spillinger, "Performance Evaluation of a Decoded Instruction Cache for Variable Instruction Length Computers", IEEE Trans. on Computers, vol.43, no.10, pp.1140-1150, Oct. 1994.

[7] S.Jourdan, P.Sainrat, D.Litaize, "Exploring Configuration of Functional Units in an Out-of-Order Superscalar Processor", Proc. of 22nd Ann. Int'l Symp. on Computer Architecture, pp.117-125, 1995.

[8] G.Kane, J.Heinrich, "MIPS RISC Architecture", Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[9] J.K.F.Lee, A.J.Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer pp.6-22, Jan. 1984.

[10] S.McFarling, "Combining Branch Predictors", WRL Technical Note TN-36, Digital Western Research Laboratory, June 1993.

[11] S.W.Melvin, M.C.Shebanow, Y.N.Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines", Proc. of 21st Ann. Int'l Symp. on Microarchitecture, pp.60-63, 1988.

[12] H.Miyajima, T.Hironaka, Y.Saitoh, K.Murakami, "Hyperscalar Processor Architecture", Trans. of IPSJ, vol.36, no.8, pp.1964-1975, Aug. 1995.

[13] K.Murakami, "Design and Evaluation of Advanced Processor Architectures", Ph.D. Dissertation, Kyoto University, Nov. 1993.

[14] C.H.Perleberg, A.J.Smith, "Branch Target Buffer Design and Optimization", IEEE Trans. on Computers, vol.42, no.4, pp.396-412, April 1993.

[15] E.Rotenberg, S.Bennet, J.Smith, "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching", Proc. of 29th Ann. Int'l Symp. on Microarchitecture, pp.24-34, 1996.

[16] S.Sechrest, C-C.Lee, T.Mudge, "Correlation and Aliasing in Dynamic Branch Predictors", Proc. of 23rd Int'l Symp . on Computer Architecture, pp.22-32, 1996.

[17] A.Seznec, S.Jourdan, P.Sainrat, P.Michaud, "Multiple-Block Ahead Branch Predictors", Proc. of Architectural Support for Programming Languages and Operating Systems VII, pp.116-127, 1996.

[18] M.Smotherman, M.Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit", Proc. of 28th Ann. Int'l Symp. on Microarchitecture, pp.219-229, 1995.

[19] G.S.Sohi, "Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", IEEE Trans. on Computers, vol.39, no.3, pp.349-359, Mar. 1990.

[20] S.Wallace, N.Bagherzadeh, "Instruction Fetching Mechanisms for Superscalar Microprocessors", Proc. of Euro-Par'96, 1996.

[21] K.C.Yeager, "The MIPS R10000 Superscalar Microprocessor", IEEE Micro, pp.28-40, April 1996.

[22] T-Y.Yeh, Y.N.Patt, "Branch History Table Indexing to Prevent Pipeline Bubbles in Wide-Issue Superscalar Processors", Proc. of 26th Ann. Int'l Symp. on Microarchitecture, pp.164-175, 1993.

[23] C.Young, N.Gloy, M.D.Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction", Proc. of 22nd Ann. Int'l Symp. on Computer Architecture, pp.276-286, 1995.