IEICE TRANS. INF. & SYST., VOL.E96–D, NO.2 FEBRUARY 2013

PAPER

Register Indirect Jump Target Forwarding

Ryota SHIOYA[†a)], ***Member***, **Naruki KURATA**[††], ***Nonmember***, **Takashi TOYOSHIMA**[††,†††], ***Member***, **Masahiro GOSHIMA**[††], ***Nonmember***, ***and*** **Shuichi SAKAI**[††], ***Fellow***

SUMMARY Object-oriented languages have recently become common, making register indirect jumps more important than ever. In object-oriented languages, virtual functions are heavily used because they improve programming productivity greatly. Virtual function calls usually consist of register indirect jumps, and consequently, programs written in object-oriented languages contain many register indirect jumps. The prediction of the targets of register indirect jumps is more difficult than the prediction of the direction of conditional branches. Many predictors have been proposed for register indirect jumps, but they cannot predict the jump targets with high accuracy or require very complex hardware. We propose a method that resolves jump targets by forwarding execution results. Our proposal dynamically finds the producers of register indirect jumps in virtual function calls. After the execution of the producers, the execution results are forwarded to the processor's front-end. The jump targets can be resolved by the forwarded execution results without requiring prediction. Our proposal improves the performance of programs that include unpredictable register indirect jumps, because it does not rely on prediction but instead uses actual execution results. Our evaluation shows that the IPC improvement using our proposal is as high as 5.4% on average and 9.8% at maximum.

key words: *processor architecture, register indirect jump, object-oriented programming*

1. Introduction

The accuracy of branch predictors is one of the most important factors for out-of-order superscalar processors. Therefore, branch predictors have been researched for a long time, and new predictors have recently been proposed [1]–[3].

Much research on branch predictors has targeted conditional branches, but *register indirect jumps* are now becoming more important than ever. This shift in focus is caused by the widespread use of *object-oriented languages*. In object-oriented languages, *virtual functions* are heavily used for the polymorphism feature, as described in Sect. 2.1. Virtual function calls usually consist of register indirect jumps, thus making them more important than ever.

It is generally more difficult to predict jump targets of register indirect jumps than those of conditional branches, as described in Sect. 3. A branch target buffer (BTB) is widely used for this purpose and it can predict the targets of register indirect jumps. A BTB is relatively effective when the target addresses of register indirect jumps are fixed. However, a BTB is not effective for register indirect jumps with variable jump targets, which often appear in programs written by object-oriented languages. Although many predictors have been proposed for register indirect jumps with variable jump targets, they either do not predict the jump targets with high accuracy or require very complex hardware, as described later in Sect. 8. This is because it is essentially difficult to predict the targets of register indirect jumps in programs written in object-oriented languages. The jump targets are basically predicted on the basis of their histories. However, in object-oriented languages, jump targets often vary depending not on their history but on data contents (Sect. 3).

We propose ***register indirect jump target forwarding***. Our proposal dynamically finds the producers of register indirect jumps in virtual function calls. After the producers are executed, the execution results are forwarded to the processor's front-end. The target addresses of register indirect jumps can be resolved by the forwarded execution results without the need to predict them. Our proposal improves the performance of programs written in object-oriented languages, which include unpredictable register indirect jumps, because our proposal forwards the actual execution results. Furthermore, our proposal is independent of prediction-based methods, and hence it is compatible with them. The evaluation results in Sect. 7 show an IPC improvement as high as 5.4%.

The rest of the paper is organized as follows. Section 2 and 3 introduce virtual function calls and their problems. Section 4 describes the basic ideas of our proposal, following which Sect. 5 gives its detailed design. Section 6 describes static code scheduling for our proposal. Then, Sect. 7 presents evaluation results, and Sect. 8 explains related works.

2. Virtual Function

This section describes virtual function calls. Section 2.1 explains virtual functions in object-oriented languages, and Sect. 2.2 explains the implementation of virtual function calls.

Manuscript received June 18, 2012.
Manuscript revised September 29, 2012.

[†]The author is with Graduate School of Engineering, Nagoya University, Nagoya-shi, 464–8603 Japan.
[††]The authors are with Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, 113–8656 Japan.
[†††]The author is with Google Japan Inc., Tokyo, 106–6126 Japan.
a) E-mail: shioya@nuee.nagoya-u.ac.jp

DOI: 10.1587/transinf.E96.D.278

2.1 Virtual Function Call

A virtual function is a *method* [†] whose behavior can be redefined in a derived class. This redefinition in a derived class is called *overriding*. A virtual function overridden in a derived class is called by the same procedure as that of a base class. This feature is called *polymorphism*, which is one of the most important features in object-oriented programming.

Figure 1 shows an example of a virtual function call and polymorphism. This example source code implements a simple application that draws circles and squares. The Drawable class is a base class of classes that actually draw graphics. This class defines an interface of the Draw() method that draws graphics. Both the Circle and Square classes inherit the Drawable class and override the Draw() method. The list stores objects of these classes and is traversed to draw graphics. In the for loop, the Draw() method is called iteratively for each object in the list. In this case, the Circle::Draw() or Square::Draw() method is called depending on the type of each object.

Polymorphism can effectively reduce program complexity, because programmers can call methods without considering the data types of objects and do not need to write explicit branches for each data type. Consequently, object-oriented languages, which include polymorphism as a feature, are widely used and are indispensable in building large-scale applications.

2.2 Implementation of Virtual Function

In object-oriented languages like C++, virtual functions are typically implemented by using a data structure called a *virtual function table* (VFT). A VFT is a table that stores pointers to virtual functions for each class type.

Figure 2 shows an example of a data structure including VFTs. This figure corresponds to the first three elements of list[N] in Fig. 1. Each object has a pointer variable vtbl. A vtbl stores a pointer to a VFT of its own class type. For example, the objects pointed by list[1] and list[2] belong to the same Square type, so each vtbl of the objects points to the same VFT in the Square type. Each VFT stores function pointers to virtual functions. In this figure, the first entry of each VFT stores a pointer to Draw().

We explain how virtual functions are called using an example in which Circle::Draw() is called from list[0], as shown in Fig. 2. This example would correspond to a statement "list[0]→Draw()" in C++. The steps for calling virtual functions are as follows:

① : A vtbl in an object is read, and then the pointer to a VFT is obtained. In Fig. 2, the vtbl in *Circle Object* is referred from list[0]. Then the pointer to *Circle Virtual Function Table* is obtained.

② : A target function pointer is loaded from the VFT. In Fig. 2, the first entry of the VFT is read. Then, the function pointer to Circle::Draw() is obtained.

```
 0:  // Class definitions.
 1:  class Drawable {
 2:      virtual void Draw() = 0;
 3:  };
 4:
 5:  class Circle : Drawable {
 6:      virtual void Draw() { … };
 7:  };
 8:
 9:  class Square : Drawable {
10:      virtual void Draw() { … };
11:  };
12:
13:  // Virtual function call.
14:  Drawable* list[N];
15:  …
16:  for( i = 0; i < N; i++ ){
17:      list[i]->Draw();
18:  }
```

Fig. 1 Virtual function call with heterogeneous list.

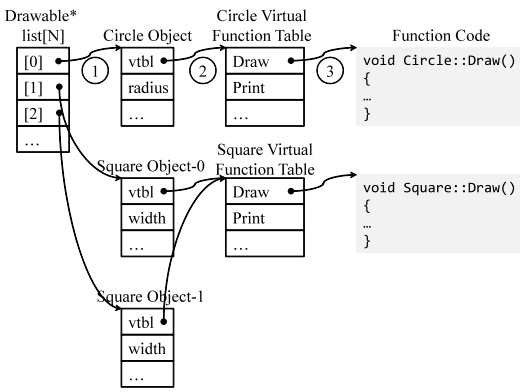

Fig. 2 Data structure of virtual function table.

```
0:  load r2 <= (r1+0)   // Load vtbl from an object
1:  load r3 <= (r2+0)   // Load Draw() addr. from vtbl
2:  jsr  (r3)           // Call Draw
```

Fig. 3 Machine code of virtual function call.

③ : A target function is called with the pointer obtained in the previous step. In Fig. 2, Circle::Draw() is called.

Virtual functions having the same name are called in the same manner among the base and derived classes. This is because the positions of vtbls in objects and layouts of VFTs are common among the base and derived classes. In Fig. 2, both the Circle and Square objects have a vtbl at the first position. Each pointer of the Circle and Square classes to a corresponding Draw() is stored at the same position in their VFTs. Therefore, Circle::Draw() and Square::Draw() are called from list[N] in the same manner.

A virtual function call is usually compiled to two load instructions and one register indirect jump instruction. These three instructions respectively correspond to the above three steps for calling virtual functions. Figure 3 shows a machine code for calling the Draw() functions shown in Fig. 1. The first load loads vtbl from register r1, which stores a pointer to an object, and obtains a pointer to

[†] In object-oriented programming, a function associated with a class is called a *method*.

a VFT. The second load loads an entry in the VFT from the obtained pointer. Then, the address to a function Draw() is obtained. The jsr, which is a register indirect jump instruction, jumps to the obtained address.

3. Problems of Virtual Function Call

Virtual function calls have the following two problems:

1. Difficulty in branch prediction of register indirect jumps.
2. No latency-hiding capability for register indirect jumps by out-of-order superscalar processors.

This section explains these problems.

3.1 Difficulty in Prediction

It is generally more difficult to predict target addresses of register indirect jumps than the directions of conditional branches. This is because of the following reasons:

1. Register indirect jumps require to predict target address values. This is in contrast to conditional branches, which require to predict a direction as either *taken* or *untaken*.
2. In conditional branches, prediction based on control flow histories are usually highly accurate. In contrast, history-based methods cannot predict the target addresses of register indirect jumps for virtual functions with high accuracy. This is because the jump targets are often changed on the basis of the data content. For example, in the heterogeneous list in Fig. 1, jump targets are changed on the basis of the contents of list[N]. Thus, the jump targets cannot be predicted from control flow histories.

A *return* instruction is a type of register indirect jumps. A predictor that is based on *return address stack (RAS)* can predict the target addresses of return instructions with high accuracy [4]–[6]. There has been much research on the other types of register indirect jumps, which will be described in Sect. 8. However, it is generally difficult to predict the jump targets, as described before.

3.2 No Latency-Hiding Capability

It is well known that out-of-order superscalar processors can hide the latency of instructions through dynamic scheduling. However, such dynamic scheduling *cannot* hide the latency of instructions in virtual function calls. This section describes this problem by comparing a case of normal instructions with that of virtual function calls.

(1) Normal Instructions

Processor performance is not directly affected by the latency of load instructions that access a level-1 data cache. This is because out-of-order superscalar processors can hide the latency of those instructions through dynamic scheduling.

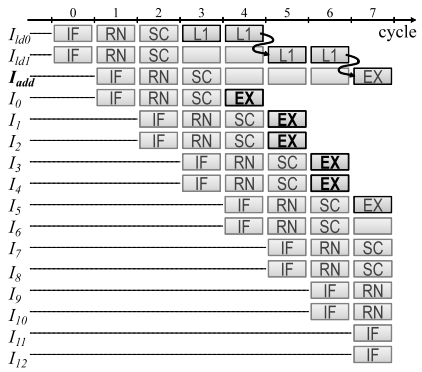

Fig. 4 Latency hiding of normal instructions.

Table 1 Label definition in pipeline charts.

Label	Definition
IF	Instruction Fetch
RN	Renaming
SC	Scheduling
EX	Execution
L1	L1 Cache Access

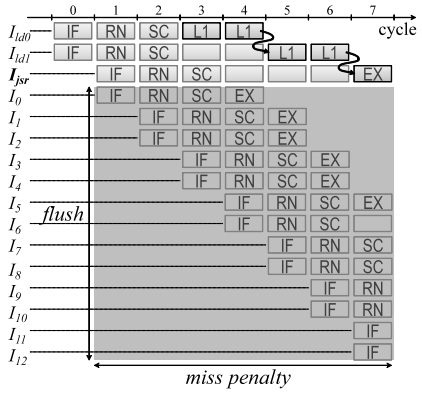

Fig. 5 Branch misprediction in virtual function call.

Figure 4 shows the behavior of a processor pipeline that hides the latency of load instructions. The processor in this figure is a two-issue out-of-order superscalar processor. Table 1 summarizes the labels in this figure. In this figure, I_{add} is an add instruction, and I_{ld0} and I_{ld1} are load instructions. I_{add} is dependent on I_{ld1}, while I_{ld1} is dependent on I_{ld0}. $I_0 \sim I_{12}$ are independent of I_{add}, I_{ld0}, or I_{ld1}.

The execution of I_{add} is delayed while waiting for the execution of I_{ld1} and I_{ld0}. Through dynamic scheduling, the processor can execute $I_0 \sim I_4$ in cycle $4 \sim 6$. This scheduling hides the latency of I_{ld1} and I_{ld0}.

(2) Virtual Function

In contrast to the case of normal instructions, processor performance is decreased directly by the latency of instructions in virtual function calls. This is because dynamic scheduling by out-of-order superscalar processors cannot hide the latency of such instructions.

Figure 5 shows the behavior of a processor pipeline

when a branch misprediction occurs on a virtual function call. In this figure, I_{jsr} is a register indirect jump. I_{jsr} is dependent on I_{ld1}, while I_{ld1} is dependent on I_{ld0}. These instructions correspond to instructions generated from a virtual function call, as described in Sect. 2.2. $I_0 \sim I_{12}$ are independent of I_{jsr}, I_{ld0} or I_{ld1}. The other labels are the same as the ones in Fig. 4. Branch misprediction penalty generally corresponds to the latency from a fetch stage to an execution stage of a branch instruction. In Fig. 5, the latency from the fetch of I_{jsr} to its execution is prolonged by I_{ld1} and I_{ld0}.

In the case of normal instructions, out-of-order superscalar processors hide the latency of load instructions by executing independent successors. However, as shown in Fig. 5, all the successors of I_{jsr} are flushed on branch misprediction. Thus, the processor cannot hide the latency.

4. Register Indirect Jump Target Forwarding

We propose ***register indirect jump target forwarding***. Our proposed scheme dynamically finds the producer instructions of register indirect jumps in virtual function calls. After a producer is executed, the execution result is forwarded to the processor's front-end. The next address of the consumer register indirect jump can be resolved by the forwarded execution results, without the use of prediction. Our proposal consists of static code scheduling of virtual function calls and a forwarding mechanism, which are explained in Sects. 4.1 and 4.2, respectively.

4.1 Static Code Scheduling

Static code scheduling generally reduces branch misprediction penalties of virtual function calls. Figure 6 shows the behavior of a processor pipeline that is executing statically scheduled instructions. Note that the instruction stream in Fig. 6 is a statically scheduled version of that in Fig. 5. In this stream, I_{jsr} is statically scheduled to separate from I_{ld1}, which is the producer of I_{jsr}. This static scheduling makes it possible for a processor to detect a branch misprediction at the earliest after the fetch of I_{jsr}. This is in contrast to the case in Fig. 5.

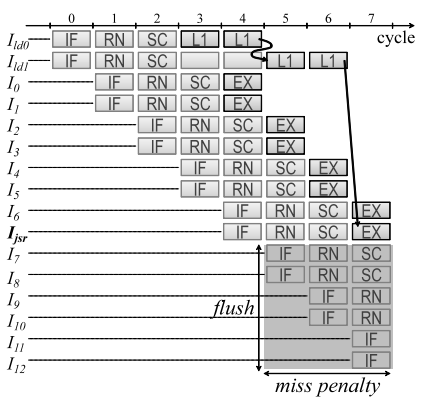

Fig. 6 Pipeline of virtual function call (optimized code).

However, the static scheduling does not improve performance anymore than the case in Fig. 6 in terms of the fact that I_{jsr} is separated further away from its producer. We explain this behavior using Fig. 7. The instructions in Fig. 7 are statically scheduled in the same way as in Fig. 6. In this figure, the processor detects the branch misprediction at the earliest after the fetch of I_{jsr} in the same way as in Fig. 6. The execution of I_{ld1} is already finished at the fetch of I_{jsr}, but I_{jsr} cannot know the result of its producer. Consequently, the branch misprediction penalty is not reduced than that in Fig. 6.

4.2 Target Forwarding

We propose a method that forwards the execution result of a producer to the processor front-end. This method resolves indirect jump targets by the forwarded results on instruction fetch, without needing conventional branch prediction. In Fig. 7, the execution of I_{ld1}, which is the producer of I_{jsr}, is finished at the fetch of I_{jsr}. In this case, our proposal forwards the result of the producer to the processor front-end. Figure 8 shows the behavior of this forwarding for the same instruction stream in Fig. 7. In Fig. 8, I_{jsr} receives the forwarded execution result from I_{ld1}. This makes it possible for the processor front-end to resolve the jump target of I_{jsr} and fetch successive instructions, without requiring any prediction. Consequently, no branch misprediction occurs and our proposal removes the misprediction penalty itself in this case.

4.2.1 Early Recovery Mechanism

If static scheduling does not separate a register indirect jump and its producer sufficiently, our proposed method will not be able to forward the execution result of the producer. This

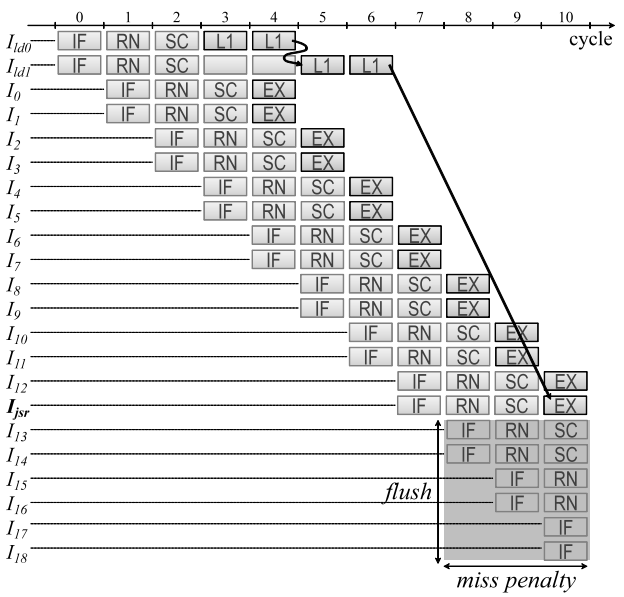

Fig. 7 Pipeline of virtual function call (optimized code, no forwarding).

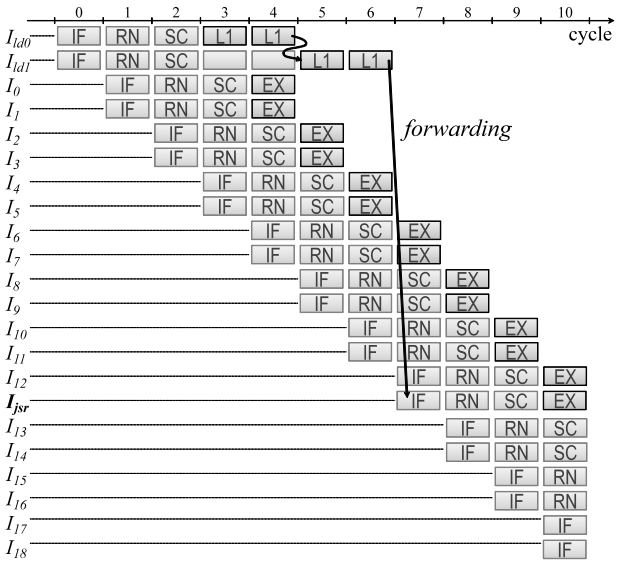

Fig. 8 Pipeline of virtual function call (optimized code, forwarding).

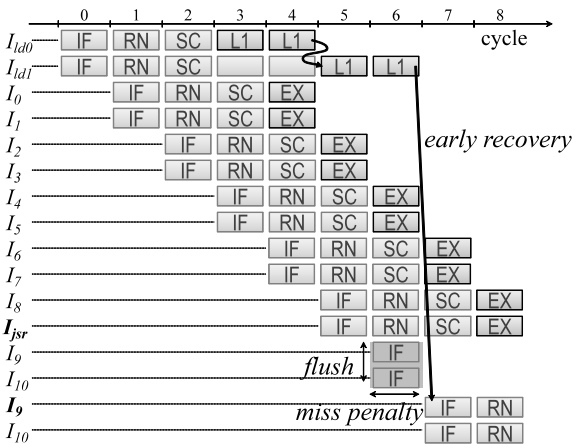

Fig. 9 Early recovery.

is because the execution of the producer occurs too late for the fetch of the register indirect jump.

We propose an *early recovery method* for such a case. Figure 9 shows the behavior of a processor pipeline with our early recovery method. In this figure, the finish of I_{ld1} occurs too late for the fetch of I_{jsr}, which is the consumer of I_{ld1}. This is because I_{jsr} is not sufficiently distant from its producer I_{ld1} statically. Therefore, the successive instructions of I_{jsr} are fetched using branch prediction, as in conventional processors. After the execution of the producer, if its result is different from the predicted target of I_{jsr}, the processor detects branch misprediction. This makes it possible for the processor to recover from branch misprediction without waiting for the execution of I_{jsr}. As shown in Fig. 9, our proposal cannot remove the branch misprediction penalty completely, but the penalty is reduced considerably when compared to that in Fig. 5.

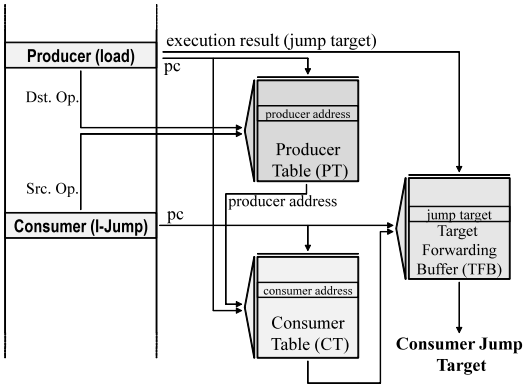

Fig. 10 Block diagram.

5. Implementation

This section describes the implementation and behavior of our proposal. Figure 10 is a block diagram of our system. Our proposal mainly consists of three tables. These tables make it possible for the processor to forward the execution results of producers to consumer register indirect jumps.

5.1 Tables

Our proposal uses the following three tables:

1. Producer Table (PT)
   A PT gives mapping from register numbers to the last instructions that update corresponding the register entries. Specifically, the PT is indexed by a logical register number and returns an instruction address. A consumer instruction looks up the address of its producer from the PT by using the source operands of the consumer.
2. Consumer Table (CT)
   A CT gives mapping from producer instructions to consumer instructions. Specifically, the CT is indexed by the instruction address of a producer and returns that of a consumer.
3. Target Forwarding Buffer (TFB)
   A TFB stores forwarded jump targets. The TFB is indexed by the instruction address of a consumer. It returns a forwarded jump target corresponding to the consumer.

The rest of this section describes the behavior of our proposal according to these three tables.

5.2 Training Tables for Forwarding

First, the processor creates links from producers to consumers. These links make it possible for producers to find their consumers on forwarding their execution results. This link mechanism is implemented by the PT and CT. Figure 11 shows an example of these links. The figure shows a producer instruction I_p (load) at address 0010. Similarly,

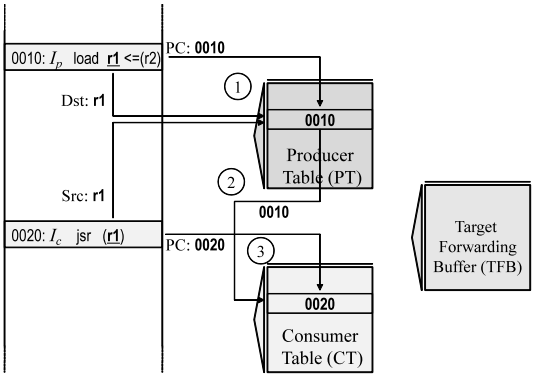

Fig. 11 Linking producer and consumer.

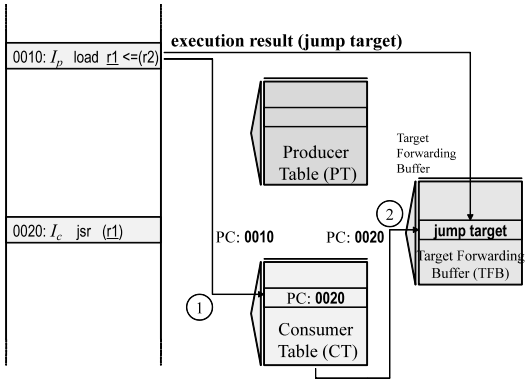

Fig. 12 Updating target forwarding buffer.

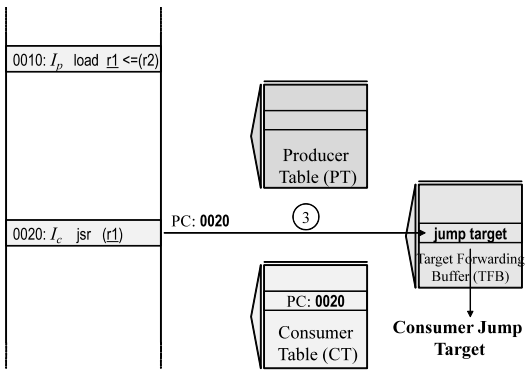

Fig. 13 Forwarding.

the figure shows the consumer instruction I_c (jsr) at address 0020. They are linked in the following manner:

① : After a producer is decoded, it writes its instruction address to an entry in the PT corresponding to its destination operand. In Fig. 11, I_p writes its address (0010) to the entry corresponding to its destination operand (r1).

② : After a consumer is decoded, it looks up an entry in the PT corresponding to its source operand. In Fig. 11, I_c looks up the entry corresponding to its source operand r1, and the PT returns 0010, which is the address of I_p. This is because I_p is the last instruction that updates r1.

③ : The consumer writes its instruction address to an entry in the CT corresponding to the producer address obtained in the phase ②. In Fig. 11, the address of I_c (0020) is written to the entry in the CT corresponding to the address of I_p (0010).

5.3 Forwarding Indirect Jump Target

Link information prepared by the PT and CT is used for forwarding jump targets. The execution result of a producer, which is the jump target of a consumer, is forwarded to the consumer. We explain the behavior of this forwarding using Figs. 12 and 13. The labels in these figures are the same as those in Fig. 11. Forwarding is performed in the following manner:

① : After a producer is executed, it looks up an entry in the CT corresponding to its instruction address. In Fig. 12, I_p looks up the entry in the CT corresponding to its address (0010), and the CT returns the address of I_c (0020).

② : The producer writes its execution result to an entry of the TFB. This execution result is a consumer's jump target. The consumer address obtained in phase ① is used as an index in the TFB when writing the result. In Fig. 12, the execution result of I_p is written to the entry in the TFB corresponding to the address of I_c (0020).

③ : When the consumer is fetched, it looks up an entry in the TFB corresponding to its instruction address. Then, the TFB returns the forwarded jump target written in phase ②. In Fig. 13, I_c looks up the entry in the TFB corresponding to its address (0020), and TFB returns the jump target forwarded by I_p.

This process makes it possible for the processor to forward the execution result of a producer to its consumer. On successful forwarding, the processor overwrites the result of the branch prediction by a forwarded jump target.

5.4 Early Recovery

The early recovery mechanism introduced in Sect. 4.2.1 can be implemented by extending slightly. After a producer is executed, it looks up its consumer in the CT. This process is similar to phase ① in the previous section. If the execution result is different from the predicted jump target of the consumer, the processor detects misprediction and then recovers from a mispredicted path. The processor recovers from misprediction in the same way as a normal recovery case, and thus the early recovery mechanism does not require any additional extensions.

6. Code Scheduling

Our proposal requires static code scheduling as described before. By exploiting properties of virtual function calls, the following heuristics make it possible for compilers to schedule instructions for calling virtual functions.

```
 0:  class Circle : Drawable {
 1:      virtual void Draw();
 2:      Drawable* Next();
 3:      void Process( Drawable* d ){
 4:          ...
 5:          Draw();         // Case 1
 6:          d->Draw();      // Case 2
 7:          Drawable* next = Next();
 8:          ...
 9:          next->Draw();   // Case 3
10:      };
11:  };
```

Fig. 14 Schedulable virtual function calls.

1. From Objects
   The simplest case is that of a virtual function that is called from a function of its object. An example of this case is shown on the line of Case 1 in Fig. 14. On this line, the virtual function Draw() is called in Process(). At the beginning of Process(), the pointer of an object that Process() belongs to is passed from an arbitrary caller, and the pointer of Draw() can be loaded here. Thus, the producer instructions of register indirect jumps, which load function pointers, can be moved to the beginning of Process().
2. From Arguments
   Similar scheduling can be done for a virtual function call of an object that is passed as an argument of functions. The line of Case 2 in Fig. 14 shows this case. At the beginning of Process(), the function pointer of Draw() is determined from the pointer of d. Thus, the producers of register indirect jumps can be moved to the beginning of the Process() by static code scheduling.
3. From Returned Objects
   Virtual function calls of returned objects can also be scheduled. The line of Case 3 in Fig. 14 shows this case. After Next() is called, the function pointer of Draw() is determined from the returned object of Next(). Thus, the producers of register indirect jumps can be moved to the position just after Next() is called.

7. Evaluation

In this section, we evaluate our proposal through simulation. We first describe the evaluation environment, and then discuss the performance improvement.

7.1 Evaluation Environment

We evaluated our proposal using a cycle-accurate processor simulator, *Onikiri* [7]. We used benchmark programs that are listed in Table 2. These benchmark programs belong to the OOCSB A C++ benchmark suite [8]. All the benchmark programs are written in C++ in an object-oriented style. These programs were compiled using gcc 4.2.1 with the "−O3" option.

During the evaluation of deltablue and richards, we skipped the first 1 G instructions and evaluated the next 100 M instructions. During the evaluation of lcom and ixx, we evaluated all the instructions executed in the programs. This is because the number of instructions executed in lcom and ixx is sufficiently small to simulate them in real time. The number of instructions of lcom and ixx that were executed were approximately 260 M and 38 M, respectively.

Table 2 Benchmark applications.

Name	Description	Input Parameter
deltablue	Constraint Solver	3000 constraints
ixx	IDL Parser	Som Plus Fresco.idl
lcom	VHDL Compiler	circuit3.1
richards	OS Simulator	1024 iterations

Table 3 Simulation configurations.

Name	Parameter
ISA	Alpha
fetch width	4 inst.
execution unit	int:4, fp:2, mem:2.
inst. window	int:32, fp:16, mem:16
sched. policy	age based†
physical reg.	int:128, fp:128
ROB	128 entries
LSQ	64 entries
branch pred.	8 KB g-share
br. miss penalty	10 cycles (minimum)
BTB	2 K entries, 4 way
RAS	8 entries
L1IC	32 KB, 4 way, 64 B/line, 2 cycles
L1DC	32 KB, 4 way, 64 B/line, 2 cycles
L2C	4 MB, 8 way, 64 B/line, 10 cycles
main memory	200 cycles
PT	32 entries
CT	32 entries,4 way set assoc.
TFB	32 entries,4 way set assoc.

Table 3 summarizes the configuration of a baseline processor. This basic structure is based on the MIPS R10000 out-of-order superscalar processor [9]. Certain configurations such as the number of integer execution units, the branch predictor and the caches were set to match those of modern processors.

We implemented our proposal on the baseline processor. Table 3 summarizes the configuration of the PT, CT, and TFB. Note that the PT has the same number of entries as the integer logical register file.

7.1.1 Evaluation Models

We evaluated the following models:

1. BASE: This is the baseline model in the evaluation. The benchmark programs are executed on the baseline processor. Note that the baseline processor predicts the jump targets of register indirect jumps by using a BTB.

†Branches have higher priority than the other types of instructions. Within the same type of instructions, older instructions have higher priorities.

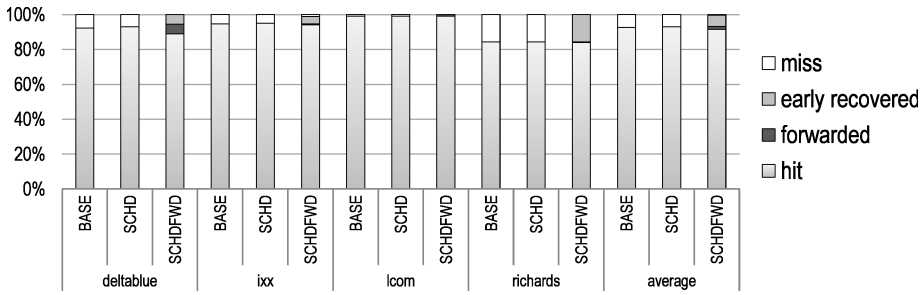

Fig. 15 Breakdown of results of register indirect jump to branch prediction.

2. SCHD: This model executes benchmark programs to which the scheduling method described in Sect. 6 is applied. For simplicity, once the source code of each benchmark program is compiled to assembly code, the code scheduling is manually applied to the assembly code. After this process, the assembly code is finally assembled to a binary file. In this model, the binary files are executed on the baseline processor.
3. FWD: This model executes the same binary files as used in the SCHD. We implemented the forwarding and early recovery methods described in Sect. 4 on the baseline processor, which is used by this model.

7.2 Evaluation Result

(1) IPC

Figure 16 shows the IPCs for each model relative to the baseline model. The average IPC improvement in the SCHD model is 3.3%. As described in Sect. 4.1, static code scheduling reduces the misprediction penalties of virtual function calls, thereby improving the IPC in this model. The IPC improvement in the FWD model is 5.4% on average, and 9.8% at maximum for "richards."

(2) Number of Recoveries

In this section, we show the evaluation result of branch misprediction penalties, because the IPC improvement is a result of the reduction of misprediction penalties achieved by our proposal.

Figure 15 shows the breakdown of the execution results of register indirect jumps. The labels in the figure are summarized as follows:

1. hit/miss: The ratio of the number of register indirect jumps whose predicted jump targets are *correct* or *wrong* to the number of all register indirect jumps. Instructions categorized as hit do not suffer from misprediction penalties while those categorized as miss do.
2. forwarded: The ratio of the number of register indirect jumps whose jump targets are successfully forwarded to the number of all register indirect jumps. This ratio does not include correctly predicted register indirect jumps. These instructions do not suffer from misprediction penalties. This ratio is applicable to only the FWD model.

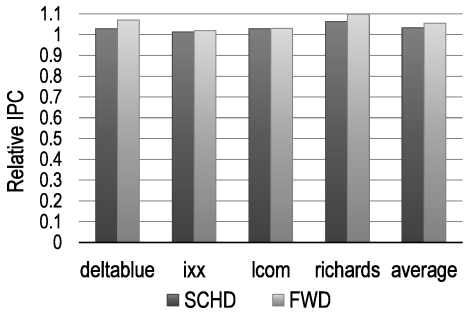

Fig. 16 Relative IPCs.

3. early recovered: The ratio of the number of register indirect jumps that are recovered early to the number of all register indirect jumps. In contrast to the case of forwarded, these instructions cannot avoid all the misprediction penalties. They suffer from misprediction penalties that are proportional to the distance between producer and consumer instructions. This ratio, too, is applicable to only the FWD model.

The average miss ratios of the BASE and SCHD models are 7.4% and 7.1%, respectively. In contrast, the average miss ratio of FWD is significantly reduced, i.e., 0.31%. This is because mispredicted instructions are corrected through the forwarding or early recovery. The average forwarded and early recovered ratios are 1.6% and 6.5%. The average hit ratios of the BASE, SCHD, and FWD models are similar, because our method does not affect the accuracy of a branch predictor.

There are 5.5% register indirect jumps categorized as forwarded in deltablue, but there are almost no register indirect jumps categorized as forwarded in the other benchmark programs. This is because we could not schedule producer or consumer instructions in the three benchmark programs sufficiently. The IPC improvements in these three benchmark programs were achieved by the early recovery mechanism.

It is not possible to determine the degree of IPC improvement using the number of early recovered register indirect jumps, because each misprediction penalties on early recovery are varied for different register indirect jumps. Thus, we show the number of flushed instructions on branch misprediction.

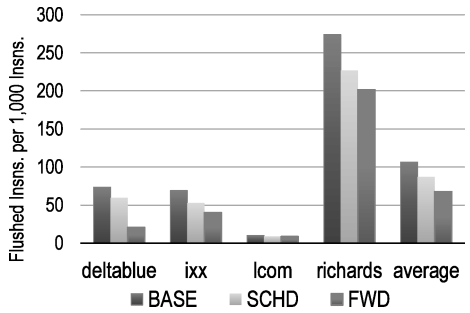

Fig. 17 Flushed instructions per 1,000 instructions.

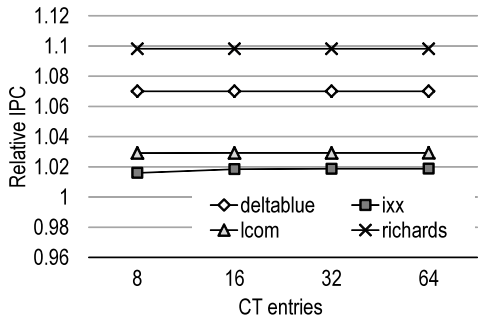

Fig. 18 Sensitivity of IPCs to CT size.

(3) Misprediction Penalty

Figure 17 shows the number of flushed instructions on mispredictions per that of 1,000 committed instructions. Note that these flushed instructions correspond to the instructions in the painted areas labeled as *flush* in Figs. 5 and 6.

On average, the number of flushed instructions in the FWD model is reduced by 19 (18%) from the BASE model. Hence, the IPCs of the SCHD model are improved even though their miss ratios were not reduced, as shown in Fig. 15.

On average, the number of flushed instructions in the FWD model is reduced by 39 (37%) from the BASE model, and hence the FWD achieved IPC improvements. In the FWD model, the number of flushed instructions is considerably larger than that in the SCHD model; thus, to achieve significant IPC improvement, code scheduling alone would not be effective, and forwarding is also necessary.

(4) Sensitivity to Table Size

Figure 18 shows the IPCs for the FWD model relative to the BASE model with 8, 16, 32, and 64 entries in the CT. In ixx, their IPCs are improved when the size of CT is increased to 32 entries; in the other benchmark programs, an 8-entry CT is sufficient.

Similarly, Fig. 19 shows the IPCs for the FWD model relative to the BASE model with 8, 16, 32, and 64 entries in the TFB. The results show a similar trend as those in Fig. 18, and a 32-entry TFB is sufficient. Note that the number of entries in PT is always the same as the number of logical registers, as indicated in Table 3.

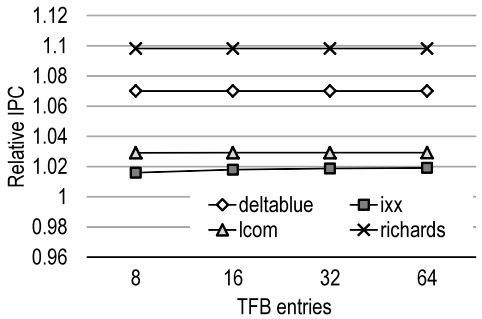

Fig. 19 Sensitivity of IPCs to TFB size.

Table 4 Area overhead.

Table	Capacity
PT	64 bits insn. addr. × 32 entries = 256 B
CT	(64 bits insn._addr. + 54 bits tag) × 32 entries = 472 B
TFB	(64 bits insn. addr. + 54 bits tag) × 32 entries = 472 B
Total	1.17 KB

Our proposal can work well with such small tables, because the static number of register indirect jumps is small. However, the dynamic number of register indirect jumps is large, and the number of instructions flushed by them is also large, as shown in Fig. 17. As a result, our proposal improves IPCs in the benchmark programs.

(5) Area Overhead

The area overhead of our proposal is mainly incurred by the three tables, PT, CT, and TFB. Table 4 shows the capacities of these tables, each with 32 entries. The amount of the capacities of the tables is approximately 1 KB, which is very small. This is considerably smaller than the capacities of BTB or PHT, whose latencies are usually one-cycle, and thus the three tables can be accessed in one-cycle. The capacities of the three tables are also considerably smaller than those of L1/L2 caches, which occupy large areas on a processor chip. Thus, the area and energy consumption overhead incurred on the entire processor using our proposal is negligible.

8. Related Works

There has been much research on predictors for register indirect jumps based on the control flow history [10]–[12]. Such history-based predictors are implemented in commercial processors, such as Intel Pentium-M [13] and IBM Power 7 [14]. These types of predictors exploit the property that indirect jump targets are related to the control flow history. Therefore, they select jump targets based on the global history of branch directions. These predictors differ mainly in terms of the usage of global history and tables that store jump targets. Our proposal can resolve jump targets that are independent of branch history, because it forwards execution results to the processor front-end directly. In addition, our proposal is independent of prediction-based methods, and thus our proposal can cooperate with history-

based methods.

Roth et al. proposed a method based on pre-execution. Their proposal detects virtual function calls and pre-executes them. It dynamically extracts instructions on which the register indirect jumps depend. This extraction is performed via a mechanism based on *Dependence-Based Prefetching* (DBP) [15], and thus their proposal requires additional hardware for DBP. Moreover, their proposal requires special multi-threading hardware for pre-execution and pre-load. In their proposal, it is difficult to load the function pointer of virtual functions early enough by pre-execution. Therefore, their proposal detects loops including virtual function calls and pre-loads objects in arrays. As a result, their proposal requires unrealistic complex hardware. In contrast, our proposal is realistic because it consists of only three small tables with simple logic.

9. Conclusion

We propose register indirect jump target forwarding. After the execution of the producers of register indirect jumps, the execution results are forwarded to the processor's front-end. The target addresses of register indirect jumps are resolved by the forwarded execution results without requiring prediction. Our proposal improves the performance of programs written in object-oriented languages, which includes unpredictable register indirect jumps, because our proposal forwards the actual execution results. The evaluation results showed that the IPC improvement is as high as 5.4% on average and 9.8% at maximum. During the evaluation, we manually scheduled the instructions of virtual function calls, but we now intend to research and develop a specific compiler that automatically schedules instructions for our proposal.

Acknowledgment

This research is partially supported by Grant-in-Aid for Young Scientists A No.24680005 from Ministry of Education, Culture, Sports, Science and Technology Japan, and by Grant-in-Aid for Scientific Research B No.23300013 from Ministry of Education, Culture, Sports, Science and Technology Japan.

References

[1] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," Proc. International Symposium on High-Performance Computer Architecture, pp.197–206, 2001.
[2] A. Seznec, "Analysis of the o-geometric history length branch predictor," Proc. International Symposium on Computer Architecture, pp.394–405, June 2005.
[3] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," The Journal of Instruction Level Parallelism, vol.8, pp.1–23, 2006.
[4] C.F. Webb, "Subroutine call/return stack," IBM Technical Disclosure Bulletin, vol.30, no.11, pp.18–20, 1988.
[5] D.R. Kaeli and P.G. Emma, "Branch history table prediction of moving target branches due to subroutine returns," Proc. International Symposium on Computer Architecture, pp.34–42, 1991.
[6] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark, "Improving prediction for procedure return with return-address-stack repair mechanisms," Proc. International Symposium on Microarchitecture, pp.259–271, 1998.
[7] "Processor simulator onikiri 2." http://www.mtl.t.u-tokyo.ac.jp/~onikiri2/
[8] U. Hölzle, J. Bogda, and S. Dieckmann, "OOCSB: Object-oriented compilers at UCSB - A C++ benchmark suite," http://www.cs.ucsb.edu/~urs/oocsb/
[9] K. Yeager, "The Mips R10000 superscalar microprocessor," IEEE Micro., vol.16, no.2, pp.28–41, 1996.
[10] P.Y. Chang, E. Hao, and Y.N. Patt, "Target prediction for indirect jumps," Proc. International Symposium on Computer Architecture, pp.274–283, 1997.
[11] K. Driesen and U. Hölzle, "Accurate indirect branch prediction," Proc. International Symposium on Computer Architecture, pp.167–178, 1998.
[12] H. Kim, J.A. Joao, O. Mutlu, C.J. Lee, Y.N. Patt, and R. Cohn, "VPC prediction: Reducing the cost of indirect branches via hardware-based dynamic devirtualization," Proc. International Symposium on Computer Architecture, pp.424–435, 2007.
[13] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R.C. Valentine, "The Intel Pentium M Processor: Microarchitecture and performance," Intel Technology Journal, vol.7, no.2, pp.21–36, 2003.
[14] B. Sinharoy, R. Kalla, W.J. Starke, H.Q. Le, R. Cargnoni, J.A. Van Norstrand, B.J. Ronchetti, J. Stuecheli, J. Leenstra, G.L. Guthrie, D.Q. Nguyen, B. Blaner, C.F. Marino, E. Retter, and P. Williams, "IBM POWER7 Multicore Server Processor," IBM J. Res. Dev., vol.55, no.3, pp.191–219, May 2011.
[15] A. Roth, A. Moshovos, and G.S. Sohi, "Dependence based prefetching for linked data structures," ACM SIGPLAN Notices, vol.33, no.11, pp.115–126, 1998.

Ryota Shioya was born in 1981. He received his M.E. and Ph.D. in Information and Communication Engineering from the University of Tokyo in 2008 and 2011, respectively. He was a research fellow of the Japan Society for the Promotion of Science from 2009. Since 2011, he is an assistant professor at the Graduate School of Engineering, Nagoya University. He is a member of IPSJ and IEEE.

Naruki Kurata was born in 1987. He is currently a doctorial student in Information and Communication Engineering in The University of Tokyo. He received his ME degree in Information and Communication Engineering from The University of Tokyo in 2012. He is a member of IPSJ.

Takashi Toyoshima is a software engineer at Google working on the Google Chrome Web browser. He works on WebSocket and SPDY. He was a researcher at Fujitsu Laboratories Limited working on the K supercomputer, and developed the ICC: a Tofu interconnect controller chip for the K supercomputer as a design team lead and a hardware engineer at Fujitsu Limited. His major concerns are computer systems, especially on microprocessors and networks. He received M.S. in information science and technology from the University of Tokyo in 2006, and B.S. in electronic engineering from the University of Tokyo in 2004. He was a board member of IPSJ SIGARC from 2008 to 2011. He is a member of the ACM and IPSJ.

Masahiro Goshima was born in 1968. He received his M.E. in engineering and Ph.D. in informatics from Kyoto University in 1994 and 2004, respectively. He was a research fellow of the Japan Society for the Promotion of Science from 1994. From 1996, he was an assistant professor in the Graduate School of Informatics, Kyoto University. Since 2005, he has been an associate professor in the Graduate School of Information Science and Technology, the University of Tokyo. He has been engaging in the research area of computer architecture. He received IPSJ Yamashita SIG research award and IPSJ best paper award in 2001 and 2002, respectively. He wrote a book titled "Digital Circuits". He is a member of IPSJ and IEEE.

Shuichi Sakai was born in 1958, received B.S., M.S. and D.E. from the University of Tokyo in 1981, 1983 and 1986, respectively. He had been working at Electrotechnical Laboratory (1986–1998), Massachusetts Institute of Technology (MIT, 1991–1992), Real World Computing (RWC, 1993–1996), University of Tsukuba (1996–1998). In 1998, he became an Associate Professor of the University of Tokyo where he has continuously been a Full Professor since 2001. His major concerns are computer systems and their applications, especially computer architectures, interconnection networks, optimizing compilers, low power architectures and dependable systems. He received several awards, including IPSJ Best Paper Award (1991), IBM Science Award (1991), Ichimura Academic Award (1995), IEEE Outstanding Paper Award (1995), Sun Distinguished SpeakerAward (1997). He is a member of IPSJ (fellow since 2010), JSAI, ACM, and IEEE.