ELSEVIER

The Journal of Systems and Software 52 (2000) 67–78

The Journal of Systems and Software

www.elsevier.com/locate/jss

Applying stack simulation for branch target buffers ☆

R-Ming Shiu *, Neng-Pin Lu, Chung-Ping Chung

Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan 30050, ROC

Received 20 August 1998; received in revised form 8 January 1999; accepted 15 February 1999

Abstract

Branch target buffer (BTB) is widely used in modern microprocessor designs to reduce the penalties caused by branches. To evaluate the performance of a BTB, trace-driven simulation is often used. However, as the trace of a typical program is very large, the simulation time is often too long. To reduce the simulation time, we developed a stack simulation technique for BTB to evaluate many sets of design parameters in one simulation pass. Due to the fact that the prediction information in the BTB does not have the inclusion property – a property which makes the stack simulation work, we propose a state vector method to enumerate the prediction information for different sets of BTB design parameters to mimic the inclusion property. Simulation results show that the state vector method greatly reduces the simulation time. The speedup of the stack simulation for BTB proposed in this paper over the traditional BTB simulation is 4.68 in terms of simulation time when 13 sets of BTB design parameters are simulated in one simulation pass. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Branch target buffer; Branch prediction; Trace-driven simulation; Single-pass simulation; Stack simulation

1. Introduction

Contemporary microprocessor designs achieve higher performance by using superpipelining and superscalar processing. In contrast, the performance degradation caused by branch instructions becomes more vital. To reduce the penalties caused by branches, many branch prediction schemes have been proposed in the literature (Lee and Smith, 1984; Dubey and Flynn, 1991). Among these schemes, the branch target buffer (BTB) has been used in many modern microprocessors.

The BTB is a storage associated with the instruction fetch stage of the instruction pipeline. As shown in Fig. 1, each entry of a BTB contains three fields to record the behavior of the previously executed branches: the *branch address*, the *target address*, and the *prediction state*. In the instruction fetch stage, the address of an instruction is compared with all the valid branch address fields. If a match is found in the comparison, the associated prediction state will be used to predict the branch direction. If the branch is predicted to be taken, the content of the target address field will be used as the next instruction address. To gain higher performance, the BTB must be designed carefully. The design parameters of a BTB include the size, placement, associativity degree, replacement, and the mechanism that maintains the prediction state. As the performance loss due to prediction miss becomes more vital, careful selections of these parameters become more crucial.

Trace-driven simulation has been the most popular method to evaluate the BTB designs (Dubey and Flynn, 1991). It can give more accurate and detailed performance data than the analytical modeling, and is more efficient than the software-based emulation. The time required in a trace-driven simulation is in proportion to both the length of the simulated trace and the number of alternative design parameters. Though the computing power of recent computers has been greatly increased, the trace lengths of generally accepted benchmark suits have also grown indefinitely. For example, BTB trace-simulation for some of the SPEC95 benchmarks may run several days with only one set of design parameters (Case, 1995). Worse yet, the speed of reading trace files from secondary storage has not been improved as expected. Several techniques, such as *trace sampling* (Laha et al., 1988; Chame and Dubois, 1993) and *trace reduction* (Wang and Baer,

☆ This paper presents partial result of a long-term research project financed by both the NSC of ROC under contract no. NSC 87-2622-E-009-009 and the industry.

* Corresponding aurhor. Tel.: +886-3-5712121-54740; fax: +886-3-5724176.

E-mail address: rmshiu@csie.nctu.edu.tw (R.-M. Shiu).

0164-1212/00/$ - see front matter © 2000 Elsevier Science Inc. All rights reserved.
PII: S 0 1 6 4 - 1 2 1 2 (9 9) 0 0 1 3 3 - 8

branch address	target address	prediction state
⋮	⋮	⋮

Fig. 1. The organization of a branch target buffer.

1991), have been developed to reduce the trace length without sacrificing accuracy. However, these techniques still require that the simulation be run once to trim the original trace down. The trimmed trace can then be used in the simulation, once for each set of design parameters. To reduce the time required in the simulation process, some approach, called the *single-pass simulation*, has been proposed to evaluate many sets of design parameters in one simulation pass for systems with certain particular properties.

The single-pass simulation techniques for storage systems of different kinds, such as memory hierarchies, buffers and caches, have been thoroughly studied in the last decade. Mattson and his colleagues (Mattson et al., 1970, 1971) explored the single-pass simulation technique to evaluate memories with different sizes, known as the *stack simulation*. They observed that when the data stored in the memories of different specifications hold the *inclusion property*, then all the stored data can be presented as in a stack and evaluated in a single trace pass. This technique has been applied to designs on different storage hierarchies (Gecsei, 1974; Tompson and Smith, 1989; Hill and Smith, 1989; Wang and Baer, 1991; Silberman, 1983; Sugumar and Abraham, 1995; Wu and Muntz, 1995).

In this research, we intend to apply the stack simulation technique to evaluate the performance of different BTB designs. Several difficulties remain to be conquered. The BTB is a storage to help predict branch directions by keeping the past branch information. The branch instruction addresses and target addresses in the BTB adhere to the inclusion property, thus we can simply apply the stack simulation on these two fields. However, since the prediction state in the BTB usually does not adhere to the inclusion property, modification must be made to the stack simulation to remedy this situation.

To mimic the including property of the prediction state, we propose a *state vector method*. The behavior of the prediction state can be modeled with a finite state automata. By enumerating the prediction states of BTBs with different sizes, this method can manipulate the finite state automata on every prediction state. As a result, the algorithm proposed in this paper can reduce the simulation time of the trace-driven simulation for the BTB designs significantly.

The remainder of this paper is organized as follows. Section 2 reviews the stack simulation for the general memories. Section 3 presents the general BTB simulation algorithm, studies the behavior of the BTBs, and creates a simulation model suitable for stack simulation. We develop the stack simulation procedure for BTBs in Section 4. Section 5 shows the experimental results and in Section 6, we conclude this paper.

2. Stack simulation

This section describes the stack simulation for the general memory as proposed in Mattson et al. (1970). Mattson et al. have showed that if the replacement algorithm used in the memories of different sizes obeys the inclusion property, these memories can be represented and maintained as a stack, and the replacement algorithm is called the *stack algorithm*. The popular stack algorithms include the least recently used (LRU) algorithm, the least frequently used (LFU) algorithm, and the optimum (OPT) algorithm. In this paper, we focus our attention on the LRU algorithm due to its popularity. However, it is easy to extend the technique to deal with other stack algorithms.

Let $X = [x_1, x_2, \ldots, x_T]$ be the *memory access trace* used in our simulation, where x_t is the access address used at time t and T is the length of the trace. Assume that the purpose of the simulation is to measure the hit ratios of memories with sizes ranged from 1 to C blocks. The inclusion property holds if the data contained in the smaller buffer are also contained in the larger buffer at any given point of simulated time. Let $M_t(c)$ be the set of blocks contained in the memory of size c at time t. The *inclusion property* can be described as follows:

$$M_t(c-1) \subseteq M_t(c) \quad \text{for all } 1 \leqslant t \leqslant T \quad \text{and} \quad 1 \leqslant c \leqslant C.$$

By this property, one can see that the sets of blocks contained in the memories of different sizes at the same time form a total ordering. And the blocks contained in the memories of different sizes can be represented with a stack $S_t = [s_t(1), s_t(2), \ldots, s_t(C)]$, where $s_t(c)$ is the cth block of the stack at time t. We call the cth block as at the *level* c of the stack. As shown in Fig. 2, $s_t(0) = \phi, s_t(c) = M_t(c) - M_t(c-1)$ for $1 \leqslant c \leqslant C$, and

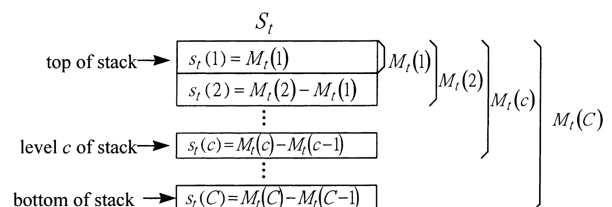

Fig. 2. The stack formed by the blocks contained in the memories of different sizes.

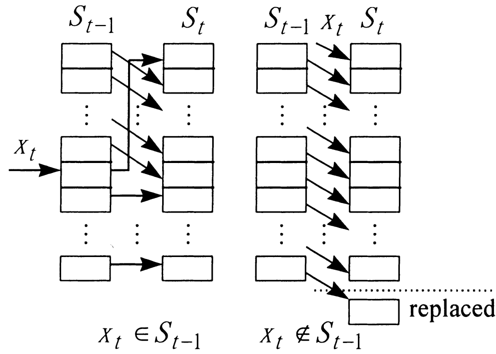

Fig. 3. Stack update with the LRU algorithm.

the blocks contained in the memory of size c are those from the stack top to level c.

The stack portrays the history of the memory access trace X. When an access x_t is to take place in the simulation, we identify the level of the stack S_{t-1} such that $s_{t-1}(\Delta_t) = x_t$, where Δ_t is the *stack distance* of x_t from the top of the stack. If x_t is not in S_{t-1}, then $\Delta_t = \infty$. The access to x_t is a hit when the memory size is larger than or equal to Δ_t; otherwise it is a miss. If the access to the stack is a hit, the stack should be updated as shown in Fig. 3. Under the LRU algorithm, the stack level position of a memory block in the stack represents the priority of a block to be replaced. The deeper the stack level of a memory block, the lower priority the memory block to stay in the memory. Since x_t has the highest priority, it is moved to the top of the stack and becomes $s_t(1)$. The other blocks originally on top of x_t in the stack are pushed down one place and kept in the original order.

The stack simulation proposed in Mattson et al. (1970) evaluates only fully associative memories. Hill and Smith (1989) extended the stack simulation to simulate memories with different associativities, called the *all-associativity simulation*. In this paper, we focus our attention on stack simulation of the fully associative BTBs. To extend the stack simulation to become the all-associativity simulation using the same technique proposed in Hill and Smith (1989), our technique can easily be applied to evaluate the BTBs with different associativities.

3. BTB simulation model

To develop the stack simulation technique for the BTB, a BTB simulation model is first built to model the behavior of a BTB simulation. We refined the BTB model in Dubey and Flynn (1991). The *BTB simulation model* contains the following four components: the *branch trace*, the *prioritized BTB entries*, the *branch prediction automata*, and the *general BTB simulation algorithm*. These components are presented in the following sections.

3.1. Branch trace

Each entry of the input trace to the BTB simulation model represents a branch encountered in the program execution trace, and must have three fields: the branch address, the branch target address, and an indicator showing whether the branch is taken or not. Preprocessing may be needed to generate this trace from the program execution trace. We define the *branch trace* as follows:

Definition 1. Branch trace – A *branch trace* $B = [(ba_1, bta_1, bt_1), (ba_2, bta_2, bt_2), \ldots, (ba_T, bta_T, bt_T)]$ is the input trace to the BTB simulator, where (ba_t, bta_t, bt_t) represents the branch encountered at time t, whose instruction address is the *branch address* ba_t, and the target address is the branch target address bta_t. If the branch is taken, the *branch taken* bt_t is *true*; otherwise, bt_t is *false*.

3.2. Prioritized BTB entries

Each entry of a typical BTB consists of three fields: the address tag, the target address, and the prediction state. Under the LRU replacement algorithm, the priority for replacement of a BTB entry can be determined by the position of the entry in the BTB. (Under other stack algorithms, a priority field may be needed to hold the priority level.) By sorting the entries in the replacement order, a total ordering set, called the *prioritized BTB Entries*, is obtained. The prioritized BTB entries are defined as follows:

Definition 2. Prioritized BTB entries – The *prioritized BTB entries*, abbreviated the *BTB entries*, is a total ordering set:

$$E_t(c) = [e_t(1,c), e_t(2,c), \ldots, e_t(c,c)],$$

where $e_t(i,c) = (ea_t(i,c), eta_t(i,c), est_t(i,c))$ is the ith entry in the BTB of size c at time t, for $1 \leqslant i \leqslant c$, whose branch address is $ea_t(i,c)$, target address is $eta_t(i,c)$, and prediction state is $est_t(i,c)$. We say that $e_t(i,c)$ is at the level i of the BTB entries. If $i > j$, the entry $e_t(i,c)$ is likely to be replaced before $e_t(j,c)$. We assume that $E_t(0)$ is empty.

3.3. Branch prediction automata

The maintenance of the prediction states of a BTB is regular and can be represented with a Moore machine with some modifications. We define a *Branch Prediction Automata* (BPA) to specify the maintenance procedure. An n-bit prediction pattern can be used to represent 2^n states of the BPA. When an entry is created in the BTB,

the prediction state associated with the entry is initialized by a *state initialization function* v in the BPA. According to the prediction state, one can predict the branch behavior using a *branch prediction function* ρ in the BPA. The input to the BPA is the branch taken field bt_t of the branch trace. If the branch at time t is taken, the input is *true*; otherwise, the input is *false*. According to the input, the state is translated by a *state transition function* δ in the BPA. That is, given a state st_{t-1} at time $t-1$, one can predict the branch behavior by the value of $\rho(st_{t-1})$ and get the state at time t by $st_t = \delta(st_{t-1}, bt_t)$. The BPA is formally defined in the following:

Definition 3. Branch prediction automata – A *Branch Prediction Automata* (BPA) is a quintuple $(Q, \Sigma, \delta, \rho, v)$, where

(1) Q is the set of states, denoted as $Q = \{q_0, q_1, \ldots, q_{2^n-1}\}$, where n is the number of bits used in the prediction state field.
(2) $\Sigma = \{false, true\}$, is the branch outcome.
(3) $\delta : Q \times \Sigma \rightarrow Q$, is a *state transition function*.
(4) $\rho : Q \rightarrow \{false, true\}$, is a *branch prediction function*.
(5) $v : \Sigma \rightarrow Q$, is a *state initialization function*.

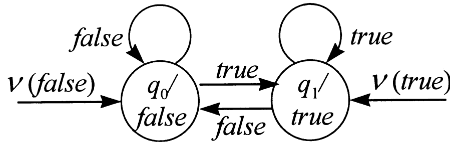

Fig. 4. The transition diagram for the BPA of a 1-bit predictor.

An example of a 1-bit predictor is given below. Here $Q = \{q_0, q_1\}, \Sigma = \{false, true\}$, and the functions δ, ρ, and v are given as follows:

$$\delta : \ \delta(q_0, false) = q_0, \ \delta(q_0, true) = q_1, \ \delta(q_1, false) = q_0, \ \delta(q_1, true) = q_1,$$

$$\rho : \rho(q_0) = false, \ \rho(q_1) = true, \quad v : v(false) = q_0, \ v(true) = q_1.$$

Fig. 4 shows the transition diagram of the BPA. The initialization arrows are determined by v, and the transition arrows are determined by δ. The inputs to the BPA are marked by the side of the transition arrows, and the outputs associated with the states are determined by the branch prediction function ρ.

3.4. General simulation for BTBs

The following presents the general algorithm to simulate the BTBs with different sizes, called the *general simulation for BTBs*. This algorithm, shown in Fig. 5, is then used as the basis to develop our stack simulation for BTBs in Section 4. The input to the algorithm is the branch trace as defined in Section 3.1. The most important metric to evaluate the BTB performance is the *prediction accuracy ratio*. The BTB hit ratio may also be inspected to see if the performance is limited by the BTB size. The outputs of this algorithm are the two counts defined as follows:

```
Algorithm General_BTB (B) ;
Input: B (branch trace) ;
Output: N_hit (hit counts), N_correct (correct counts) ;
begin
 1. for c := 1 to C do                                              //For all sizes.
 2.    E_0(c) := φ ;                                                //Initialize BTB.
 3.    N_hit(c):= 0 ;  N_correct(c):= 0 ;                           //Initialize counts.
 4.    for t := 1 to T do                                           //For all branches.
 5.       if ∃ e_t-1(d, c) ∈ E_t-1(c) such that ea_t-1(d, c) = ba_t then   //If BTB hits.
 6.          N_hit(c) := N_hit(c) + 1 ;                             //Count the hit.
 7.          pred := ρ(est_t-1(d, c)) ;                             //Predict.
 8.          e_t(1, c) := (ba_t, bta_t, δ(est_t-1(d, c), bt_t)) ;   //Build e_t(1, c).
 9.          for i := 2 to d do e_t(i, c) := e_t-1(i - 1, c) ;
10.          for i := d + 1 to c do e_t(i, c) := e_t-1(i, c) ;      //Build other entries.
11.          if Predict_Verify (pred, bt_t, bta_t, eta_t-1(d, c)) then   //If predict correct.
12.             N_correct(c) := N_correct(c) + 1 ;                  //Count the correct.
13.       else                                                      //If BTB misses.
14.          e_t(1, c) := (ba_t, bta_t, v(bt_t)) ;                  // Build e_t(1, c).
15.          for i := 2 to c do e_t(i, c) := e_t-1(i - 1, c) ;      // Push other entries.
16.          if bt_t = false then                                   //If branch not taken.
17.             N_correct(c) := N_correct(c) + 1 ;                  // Count the correct.
```

Fig. 5. The general simulation algorithm for BTBs.

Definition 4. BTB hit count and prediction correct count–For a BTB with size c, where $1 \leqslant c \leqslant C$, the *BTB hit count*, denoted as $N_{\text{hit}}(c)$, is the number of BTB hits, and the *prediction correct count*, denoted as $N_{\text{correct}}(c)$, is the number of predictions which are correct.

Thus the prediction accuracy ratio for a BTB with size c is equal to $N_{\text{correct}}(c)/T$, and the hit ratio is $N_{\text{hit}}(c)/T$. The average penalty caused by mispredictions can be represented by $p \times (1 - N_{\text{correct}}(c)/T)$, where p is the penalty caused by a misprediction. In this paper, we assume that all the penalties caused by mispredictions in different conditions are the same for simplicity and clarity. It is an easy task to modify the algorithm to simulate with different penalties in different conditions if necessary.

This algorithm simulates the behavior of BTBs with sizes from 1 to C. For an arbitrary size c, this algorithm works as follows: Each time a branch (ba_t, bta_t, bt_t) is encountered, $E_{t-1}(c)$ is searched to see if there is an $e_{t-1}(d,c)$ in it such that $ea_{t-1}(d,c) = ba_t$. If the entry is found, the BTB hit count will be increased and the branch will be predicted using the value of $\rho(est_{t-1}(d,c))$. This entry is then given the lowest priority to be replaced, and becomes the first entry of $E_t(c)$. Its prediction state is also updated to $\delta(est_{t-1}(d,c), bt_t)$. The order of the other entries of $E_t(c)$ keeps unchanged.

On the other hand, if the search of $E_{t-1}(c)$ for $e_{t-1}(d,c)$ is a miss, a new entry is created at the top of the BTB with its prediction state initialized to $v(bt_t)$. Other entries in the $E_t(c)$ still keep the original order, and the previously last entry is replaced. In case of a BTB miss, the prediction is still correct if the branch happens to be not taken.

In this algorithm, we use a Predict_Verify function shown below to check if the prediction is correct.

function Predict_Verify (*pred*, *bt*: **boolean**, *bta*, *eta*: **address**): **boolean**;
begin
Predict_Verify := $(pred = bt)$ **and** $(bt = false$ **or** $(bt = true$ **and** $bta = eta))$;
end

This function also appears in the next algorithm presented in this paper. The inputs to this function are the prediction result *pred*, the branch taken field *bt*, the branch target address *bta* of the branch trace and the target address field *eta* of the BTB entries. If the branch is not taken, a prediction is said to be correct only if it also predicts that the branch will not be taken. However, if the branch is taken, a prediction is said to be correct only when it predicts a taken branch and the predicted target address agrees with the real target address.

This algorithm scans the branch trace once for every BTB size evaluated. Since the trace is either read from a file or obtained during program execution, the time needed to scan the trace is critical. We develop a stack simulation technique for the BTB in the following section to overcome this defect.

4. Stack simulation for the BTBs

In this section, we develop a stack simulation technique to evaluate the performance of BTBs. We first show that the BTBs have the inclusion property if their entries do not contain the prediction state field, or if the prediction state field contains a 1-bit predictor. Then we point out that the BTBs have not the inclusion property if their prediction state field contains more than 1 bit. Finally, we propose a *stack vector* method to mimic the inclusion property for multi-bit predictors, and develop the stack simulation for the BTBs.

4.1. Inclusion property of the BTBs with no prediction state field

To apply the stack simulation technique to evaluate the BTBs, the inclusion property of the BTBs must first be guaranteed. A typical BTB entry contains the following three fields: the branch address field ea_t, the target address field eta_t, and the prediction state est_t. The behaviors of the first two fields are similar to that of an ordinary memory, thus their inclusion property can easily be proved. If the BTB entries contain no prediction state field, conventional stack simulation algorithm can easily be applied to evaluate such BTBs.

To show that these two fields, ea_t and eta_t, have the inclusion property, we must show that each entry contained in any BTB will also be contained in a BTB of larger size. Moreover, since the entries in a BTB are ordered, thus the first i entries of any arbitrary BTB must all be the same. We show this fact in Theorem 1.

Theorem 1. *Inclusion property of the BTBs with no prediction state field – Under the LRU replacement algorithm, for any time t and any BTB with size c ranged from 1 to C,*

$$ea_t(i,c) = ea_t(i,C) \text{ and } eta_t(i,c) = eta_t(i,C) \quad \text{for all } 1 \leqslant i \leqslant c.$$

Proof. We prove this theorem by induction. We first choose an arbitrary C. The equalities must hold at $t = 1$, since there is only one entry in the BTB. Assume that at time $t-1$,

$$\text{for all } 1 \leqslant c \leqslant C \text{ and } 1 \leqslant i \leqslant c, \quad ea_{t-1}(i,c) = ea_{t-1}(i,C).$$

Beginning with this induction hypothesis, the proof must satisfy the three conditions shown below, accord-

ing to the assignment of ea_t in the general simulation for BTBs shown in Fig. 5:

(1) For all $1 \leqslant c \leqslant C$, $ea_t t(1,c) = ba_t$ by lines 8 and 14 of the general simulation algorithm for BTBs.

(2) If we can find an $e_{t-1}(d,C)$ in $E_{t-1}(C)$ such that $ea_{t-1}(d,C) = ba_t$, then

(i) For all $1 \leqslant c < d$, $ea_t(i,c) = ea_{t-1}(i-1,c) = ea_{t-1}(i-1,C)$, for all $2 \leqslant i \leqslant c$, by line 15; besides, $ea_t(i,C) = ea_{t-1}(i-1,C)$ for all $2 \leqslant i \leqslant c$, by line 9.

(ii) For all $d \leqslant c \leqslant C$,

$$ea_t(i,c) = ea_{t-1}(i-1,c) = ea_{t-1}(i-1,C)$$
$$\text{for all } 2 \leqslant i \leqslant d, \text{ by line 9, and}$$
$$ea_t(i,c) = ea_{t-1}(i,c) = ea_{t-1}(i,C)$$
$$\text{for all } d+1 \leqslant i \leqslant c, \text{ by line 10.}$$

(3) If we cannot find an $e_{t-1}(d,C)$ in $E_{t-1}(C)$ such that $ea_{t-1}(d,C) = ba_t$, then for all $1 \leqslant c \leqslant C$ and $2 \leqslant i \leqslant c, ea_t(i,c) = ea_{t-1}(i-1,c) = ea_{t-1}(i-1,C)$, by line 15.

From the above conditions (1)–(3), we conclude that

for all $1 \leqslant c \leqslant C$ and $1 \leqslant i \leqslant c$, $\quad ea_t(i,c) = ea_t(i,C)$.

In the similar way, we can get that

for all $1 \leqslant c \leqslant C$ and $1 \leqslant i \leqslant c$, $\quad eta_t(i,c) = eta_t(i,C)$. □

By Theorem 1, we know that for the two fields ea_t and eta_t, the first i entries of any arbitrary BTB are the same. Thus the contents of these two fields in the largest BTB can be used to represent the entries of BTBs with different sizes. Since the contents of these two fields do not change as a function of BTB size, we rename these fields in the largest BTB and remove the index of size as follows:

Definition 5. Branch address field and target address field – The content of the *branch address field* at level i and time t is denoted as $a_t(i)$ and is equal to $ea_t(i,C)$. The content of the *target address field* at level i and time t is denoted as $ta_t(i)$ and is equal to $eta_t(i,C)$.

4.2. Inclusion property of BTB with 1-bit prediction state field

If the prediction state field of the BTB works as the 1-bit predictor shown in Fig. 4, then the BTB has the inclusion property, and the general stack simulation algorithm can be applied to evaluate such BTBs. This fact is formally stated in Theorem 2.

Theorem 2. *Inclusion property of the* 1*-bit prediction state field – Under the LRU replacement algorithm, for any time t and any BTB size c ranged from* 1 *to C, if the est field is* 1*-bit wide and works as the* 1*-bit predictor, then*

$est_t(i,c) = est_t(i,C)$ *for all* $1 \leqslant i \leqslant c$.

Proof. The proof is very similar to the proof of Theorem 1, except that all the *ea* fields are changed to become the *est* fields, and condition (1) of Theorem 1 should be changed to

$(1)'$ (i) If we can find an $e_{t-1}(d,C)$ in $E_{t-1}(C)$ such that $ea_{t-1}(d,C) = ba_t$, then $est_t(1,c) = v(bt_t)$ for all $1 \leqslant c \leqslant d$, by line 14 of the general simulation algorithm for BTBs, and $est_t(1,c) = \delta(est_{t-1}(d,c), bt_t)$ for all $d+1 \leqslant c \leqslant C$, by line 8. Since the transition function of the 1-bit predictor always sets $est_t(1,c)$ to $v(bt_t)$, regardless of the previous $est_{t-1}(d,c)$ state, $est_t(1,c) = est_t(1,C)$.

(ii) If we cannot find an $e_{t-1}(d,C)$ in $E_{t-1}(C)$ such that $ea_t(d,C) = ba_t$, then for all $1 \leqslant c \leqslant C, est_t(1,c) = v(bt_t)$ by line 14. So $est_t(1,c) = est_t(1,C)$. □

4.3. Finite automata problem

Here we show that the prediction state field est_t may violate the inclusion property by giving an example. Since the prediction state is maintained as a finite automata – the BPA, we call such a violation the *finite automata problem*.

Consider the instance shown in Fig. 6(a). The BTB entries of BTB sizes ranging from 1 to 3 are shown by concatenating the branch address and the states of the three sizes in sequence. The states are maintained by a 2-bit shifter whose BPA is shown in Fig. 6(b). From $t = 1$ to 6, the given sequence of branches are encountered and their addresses ba_t and taken fields bt_t were shown on the following rows of Fig. 6(a). The entries in E are modified according to the general simulation algorithm for BTBs, and the three branch address fields are modified separately. At $t = 6$, the three branch address fields of the first entry are in different states. These fields are shadowed in the figure. So, the prediction state field does not show the inclusion property.

4.4. State vector

To solve the finite automata problem, we enumerate the prediction states of all BTBs of different sizes, and manipulate the behavior of them using the BPA. By concatenating the states of all sizes to be evaluated, we define a *state vector* as follows. The cth state of the vector at level i is equal to the state of the ith entry of the BTB of size c. When the size is smaller than the level, the state does not exist in the BTB.

Definition 6. State vector – A *state vector* at level i and time t is $stv_t(i) = [st_t(i,1), st_t(i,2), \ldots, st_t(i,C)]$, where $1 \leqslant i \leqslant C$, and $st_t(i,c) = est_t(i,c)$, if $i \leqslant c$; otherwise, $st_t(i,c)$ can be discarded.

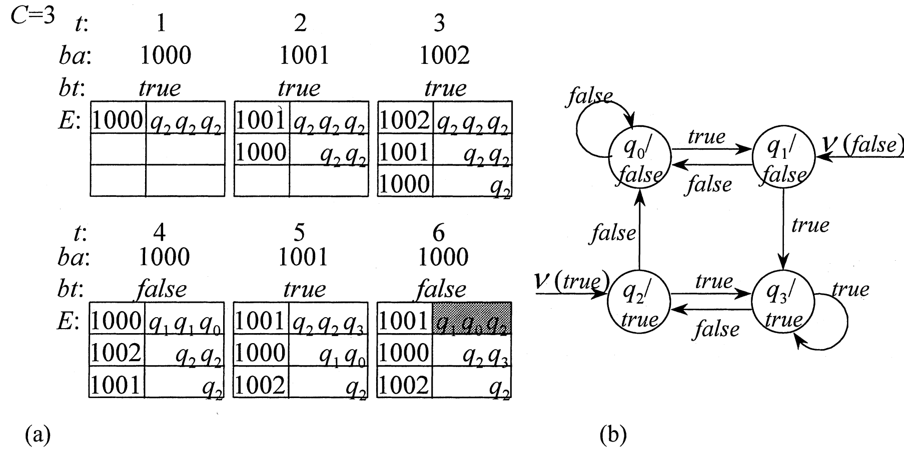

Fig. 6. An example of the finite automata problem. (a) Comparing states in BTBs with different capacities. (b) The BPA of a 2-bit shifter.

We modify the BTB entries to become the concatenation of the branch address field, the target address field, and the state vector. Since these three fields are all size independent, we can remove the index of the BTB size from the representations of the entries. We call the modified BTB entries the *BTB entries with state vector*, and define them in the following.

Definition 7. BTB entries with state vector – The *BTB entries with state vector*, abbreviated as the *extended BTB entries*, is a total ordering set

$E'_t(c) = [e'_t(1), e'_t(2), \dots, e'_t(c)],$

where $e'_t(i) = (a_t(i), ta_t(i), stv_t(i))$ is the ith entry in a BTB of size c at time t for all $1 \leqslant i \leqslant c$, whose branch address is $a_t(i)$, target address is $ta_t(i)$, and the associated state vector is $stv_t(i)$. The order of these entries is the same as that of the BTB entries. We assume that $E'_t(0)$ is empty.

The way to turn the BTB entries into the extended BTB entries is shown in Fig. 7. In Fig. 7(a), the extended BTB entries with sizes ranging from 1 to C are listed. Since the first two fields $ea_t(i,c)$ and $eta_t(i,c)$ of a BTB entry are fixed independent of the BTB size, these two fields can be renamed to $a_t(i)$ and $ta_t(i)$. The state vector stv_t enumerates the states of different BTB sizes. The extended BTB entries consist of the three fields: a_t, ta_t, and stv_t, as shown in Fig. 7(b). Note that the lower left triangular portion of the stv_t matrix which is shadowed in Fig. 7(b) does not actually exist.

To develop the stack simulation algorithm for BTB, it must be shown that the extended BTB entries have the inclusion property. Since the contents of the fields of the extended BTB entries are size independent, the first i entries of any arbitrary BTB are the same. Thus the proof of the inclusion property of the extended BTB entries is trivial.

Theorem 3. *Inclusion property of the extended BTB entries*:

$E'_t(c-1) \subseteq E'_t(c)$ *for all* $1 \leqslant t \leqslant T$ *and* $1 \leqslant c \leqslant C$.

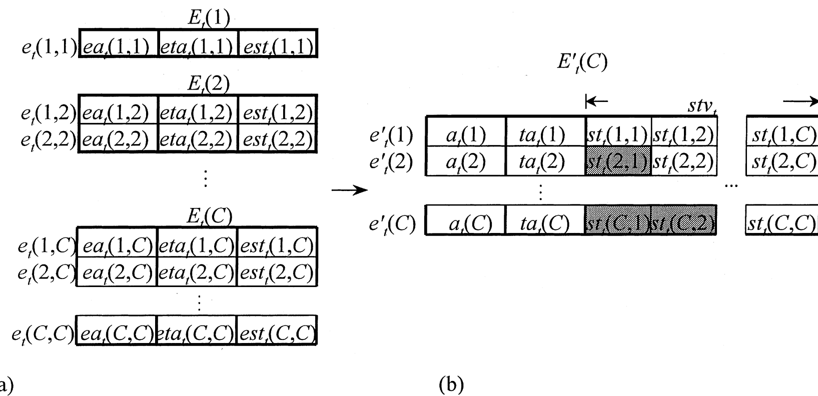

Fig. 7. Turning the BTB entries into the extended BTB entries. (a) The BTB entries of different BTB sizes. (b) The BTB entries with state vectors.

Proof.

Since $[e'_t(1), e'_t(2), \ldots, e'_t(c-1)] \subseteq [e'_t(1), e'_t(2), \ldots, e'_t(c)]$

for all $1 \leqslant t \leqslant T$ and $1 \leqslant c \leqslant C$,

$E'_t(c-1) \subseteq E'_t(c)$ for all $1 \leqslant t \leqslant T$ and $1 \leqslant c \leqslant C$. □

4.5. Stack simulation for BTBs

With the extended BTB entries, we can combine the features of conventional stack simulation and the general simulation for BTBs, and propose a *stack simulation for BTBs* to evaluate the BTBs of different sizes in a single simulation pass. The stack simulation algorithm is shown in Fig. 8 and works as follows: Each time a branch (ba_t, bta_t, bt_t) is encountered, $E'_{t-1}(C)$ is searched for an $e'_{t-1}(\Delta)$ such that $a_{t-1}(\Delta) = ba_t$, where Δ is the stack distance. If the search is a miss, then Δ will be set to be $C+1$. For the BTBs of sizes smaller than Δ, the branch results in a BTB miss, and the prediction is correct if the branch is not taken. For the BTBs of sizes larger then or equal to Δ, the BTB search is a hit and the prediction is determined by the states enumerated in the vector $stv_{t-1}(\Delta)$. The accuracy of these predictions is checked by the Predict_Verify function. Since the branch just encountered has the lowest priority to be replaced, it is pushed onto the top of the stack, and its state vector can be obtained by the following way: For the BTB sizes smaller than Δ, the BTB search is a miss, and the states are initialized by the state initialization function ν. For the BTB sizes larger than or equal to Δ, the BTB search is a hit and the states are obtained by the state transition function δ, according to the previous states obtained from the vector $stv_{t-1}(\Delta)$.

To reduce the time needed for simulation statistics, we count the number of BTB search hits in a set of *partial BTB hit counts*, denoted as $N_{p_hit}(c)$. Each time a branch is encountered and its Δ is known, for all the BTBs whose sizes are larger than or equal to Δ, we increase the $N_{p_hit}(\Delta)$ to record this BTB hit. At the end of the simulation, the BTB hit counts can be obtained by summing up these partial BTB hit counts.

This algorithm scans the trace only once and evaluates the BTBs of all sizes considered. Since the time to scan the trace is always critical, the improvement due to the stack simulation is enormous.

To prove the correctness of this algorithm, we must show that the hit and correct counts of this algorithm are the same as those of the general simulation algorithm for BTBs. Since these counts are summed up in the same way according to $\{E_t(c) | 1 \leqslant c \leqslant C\}$ (the BTB entries of different sizes) and $E'_t(C)$ (the extended BTB entries), we can prove the correctness of the stack algorithm by showing that at any time t, the information in $\{E_t(c) | 1 \leqslant c \leqslant C\}$ and $E'_t(C)$ is the same. In the following proof, we denote the nth-line statement of the general simulation algorithm for BTBs in Fig. 5 with Gn, and the stack simulation algorithm for BTBs in Fig. 8 with Sn, respectively.

Algorithm Stack_BTB_Vector(B) ;
Input: B (branch trace) ;
Output: N_{hit} (hit counts), $N_{correct}$ (correct counts) ;
begin

1. $E'_0(C) := \phi$; //Initialize BTB stack.
2. **for** $c := 1$ **to** C **do** $N_{p_hit}(c) := 0$; $N_{correct}(c) := 0$; //Initialize counts.
3. **for** $t := 1$ **to** T **do** //For all branches.
4.   **if** $\exists\, e'_{t-1}(\Delta) \in E'_{t-1}(C)$ such that $a_{t-1}(\Delta) = ba_t$ **then** //If BTB hit.
5. $N_{p_hit}(\Delta) := N_{p_hit}(\Delta) + 1$; //Count the hit.
6.   **else** $\Delta := C + 1$; //Miss even in the largest BTB.
7.   **if** $bt_t = false$ **then** //If branch not taken.
8.   **for** $i := 1$ **to** $\Delta - 1$ **do** $N_{correct}(i) := N_{correct}(i) + 1$; //Prediction correct.
9.   **for** $i := \Delta$ **to** C **do** //For all sizes.
10. $pred := \rho(st_{t-1}(\Delta, i))$; //Predict.
11.   **if** Predict_Verify ($pred, bt_t, bta_t, ta_{t-1}(\Delta)$) **then** //If prediction correct.
12. $N_{correct}(i) := N_{correct}(i) + 1$; //Predict correct.
13. $a_t(1) := ba_t$; $ta_t(1) := bta_t$;
14.   **for** $i := 1$ **to** $\Delta - 1$ **do** $st_t(1, i) := \nu(bt_t)$;
15.   **for** $i := \Delta$ **to** C **do** $st_t(1, i) := \delta(st_{t-1}(\Delta, i), bt_t)$; //Build $e'_t(1)$.
16.   **for** $i := 2$ **to** Δ **do** $e'_t(i) := e'_{t-1}(i-1)$;
17.   **for** $i := \Delta + 1$ **to** C **do** $e'_t(i) := e'_{t-1}(i)$; //Build other entries.
18. **for** $c := 1$ **to** C **do** $N_{hit}(c) := \sum_{i=c}^{C} pN_{hit}(i)$; //Sum up the hit counts.

Fig. 8. The stack simulation algorithm for BTBs.

Theorem 4. *The correctness of the stack simulation algorithm for BTBs – In the stack simulation for BTBs, for any time t and any BTB size c ranged from* 1 *to* C,

$$a_t(i) = ea_t(i,c), ta_t(i) = eta_t(i,c), \text{ and } st_t(i,c) = est_t(i,c) \text{ for all } 1 \leqslant i \leqslant c.$$

Proof. We prove this theorem by induction. We first choose an arbitrary C. The equalities must hold at $t = 1$, since there is only one entry in the BTB and for all $1 \leqslant c \leqslant C$, $e_t(1,c) := (ba_t, bta_t, v(bt_t))$ (by G14); while

$$a_t(1) := ba_t; ta_t(1) := bta_t(\text{by S13}), st_t(1,c) := v(bt_t) \text{ (by S14)}.$$

Assume that at time $t-1$, for all $1 \leqslant c \leqslant C$ and $1 \leqslant i \leqslant c$,

$$a_{t-1}(i) = ea_{t-1}(i,c), ta_{t-1}(i) = eta_{t-1}(i,c) \text{ and } st_{t-1}(i,c) = est_{t-1}(i,c).$$

Beginning with this induction hypothesis, the proof must satisfy the three conditions shown below, according to the assignment of ea_t in the stack simulation algorithm for BTBs shown in Fig. 5:

(1) For all $1 \leqslant c \leqslant C$, $ea_t(1,c) = ba_t, eta_t(1,c) = bta_t$ (by G8 and G14); while $a_t(1) = ba_t, ta_t(1,c) = bta_t$ (by S13).

(2) If we can find an $e'_{t-1}(d)$ in $E'_{t-1}(C)$ such that $a_{t-1}(d) = ba_t$, then

(i) For all $1 \leqslant c < d$, by the induction hypothesis, we cannot find an $e_{t-1}(d,c)$ in $E_{t-1}(c)$ such that $ea_{t-1}(d,c) = ba_t$. Hence

(1) For all $1 \leqslant i \leqslant c$, $ets_t(1,t) = v(bt_t)$ (by G14); while $st_t(1,i) = v(bt_t)$ (by S14).

(2) For all $2 \leqslant i \leqslant c, e_t(i,c) = e_{t-1}(i-1,c)$ (by G15); while $e'_t(i) = e'_{t-1}(i1)$ (by S16).

(ii) For all $d \leqslant c \leqslant C$, by the induction hypothesis, we can find an $e_{t-1}(d,c)$ in $E_{t-1}(c)$ such that $ea_{t-1}(d,c) = ba_t$. Hence

(1) For all $c \leqslant i \leqslant C, est_t(1,i) = \delta(est_{t-1}(d,i), bt_t)$ (by G8); while $st_t(1,i) = \delta(est_{t-1}(d,i), bt_t)$ (by S15).

(2) For all $2 \leqslant i \leqslant d, e_t(i,c) = e_{t-1}(i-1,c)$ (by G9); while $e'_t(i,C) = e'_{t-1}(i-1,C)$ for all $2 \leqslant i \leqslant d$ (by S16).

(3) For all $d+1 \leqslant i \leqslant c, et(i,c) = e_{t-1}(i,c)$ (by G10); while $e'_t(i) = e'_{t-1}(i)$ for all $d+1 \leqslant i \leqslant c$ (by S17).

(3) If we cannot find an $e'_{t-1}(d)$ in $E'_{t-1}(C)$ such that $a_{t-1}(d) = ba_t$, then by the induction hypothesis, for all $1 \leqslant c \leqslant C$, we cannot find an $e_{t-1}(d,c)$ in $E_{t-1}(c)$ such that $ea_{t-1}(d,c) = ba_t$. Hence

(i) For all $1 \leqslant c \leqslant C, est_t t(1,c) = v(bt_t)$ (by G14); while $st_t(1,c) = v(bt_t)$ (by S14).

(ii) For all $2 \leqslant i \leqslant c, e_t(i.c) = e_{t-1}(i-1,c)$ (by G15); while $e'_t(i) = e'_{t-1}(i)$ for all $d+1 \leqslant i \leqslant c$ (by S17).

From the above conditions (1)–(3), we conclude that for all $1 \leqslant c \leqslant C$ and $1 \leqslant i \leqslant c$, $a_t(i) = ea_t(i,c)$, $ta_t(i) = eta_t(i,c)$ and $st_t(i,c) = est_t(i,c)$. □

5. Experiments

To show the speedup of the stack simulation over the general simulation for BTBs, we build a trace-driven simulator to obtain their simulation times. In this section, we first describe the overview of the simulator built. Then the benefits of our approach are demonstrated by comparing the simulation times of the different methods.

5.1. Simulator overview

We build a trace-driven simulator to measure the simulation times of the general simulation and the stack simulation for BTBs. An input file is used to specify the simulation options, including the BPA, the algorithm mode, and the range of the BTB sizes to be simulated. To simplify the process of simulation without losing generality, we assume that the sizes of BTBs to be simulated are powers of 2. And the only BPA used in this simulator is the 2-bit saturation counter.

Six benchmark programs, as listed in Table 1, are selected from the SPEC CINT95 benchmark suite (Reilly, 1995) and compiled to the MIPS code for experiments. The original benchmark traces are obtained by the *pixie* (Smith, 1991) utility which runs on an MIPS machine. The branch traces are then filtered from the original traces.

The simulation times of the two algorithms are measured by the UNIX *time* command. This simulator is run on a lightly loaded DECstation 5133. In the simulation, we assume the fully associative BTBs with LRU replacement algorithm. There are two algorithmic modes in the simulator, the *general mode* and the *stack mode*. In the general mode, we run the general simulation algorithm shown in Fig. 5 in only one iteration with $C = 2^{12}$. In the stack mode, we run the stack simulation algorithm shown in Fig. 8 to measure the BTBs whose sizes range from 2^0 to 2^{12}. Since only the BTB sizes of

Table 1
Benchmark programs selected from SPEC CINT95

Benchmark	Description
Go	Application for artificial intelligence
Gcc	GNU C compiler
Compress	File compression
Li	LISP interpreter
Ijpeg	Graphic compression and decompress
Vortex	A database program

powers of 2 are considered, the number of sets of parameters that the stack mode evaluates in one pass is 13.

5.2. Simulation time

We list the number of branches simulated for the six benchmarks in Fig. 9.

In Fig. 10, we show the simulation time of the two algorithmic modes together with the *trace read time* of each benchmark, where the *trace read time* is the time needed to read the trace files while the simulator does nothing else. The simulation time of all benchmarks are proportional to the branch counts. On average, the simulation time of the stack mode is only 1.17 times that of the general mode. Nevertheless, with only this slight amount of increase in simulation time, this proposed stack simulation algorithm can simulate the behaviors of 13 various-sized BTBs, as opposed to only 1.

Note that the trace read time is dependent on the environment of the simulator. We calculate the *net* simulation times of the two algorithmic modes by subtracting the trace read time from the original total simulation times. In our environment, the trace read time is 8.16 times the net simulation time on average in the general mode. We list the net simulation times of the two algorithmic modes in Fig. 11. The net simulation time of the stack mode is 2.58 times that of the general mode.

For each set of parameters, the speedup of the stack mode over the general mode in terms of the net simulation time is shown in Fig. 12. On average, this speedup of the stack mode over the general BTB algorithm is 4.68. The speedups are obtained assuming that the trace read time is zero, and that 13 sets of parameters are evaluated in one simulation pass in the stack mode. These speedups would have been higher if the ratio of the trace read time is much higher, or if more sets of BTB sizes are to be evaluated in one simulation pass. This justifies the benefits of the stack simulation technique for the BTBs that we have proposed.

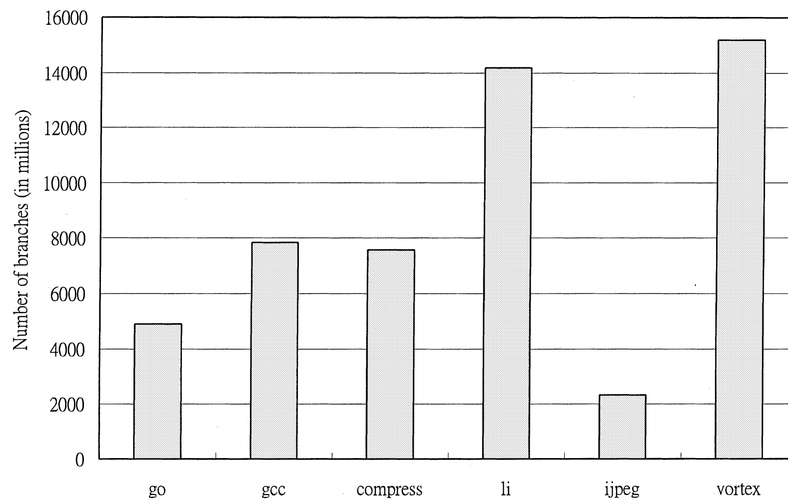

Fig. 9. Dynamic branch counts of the benchmarks.

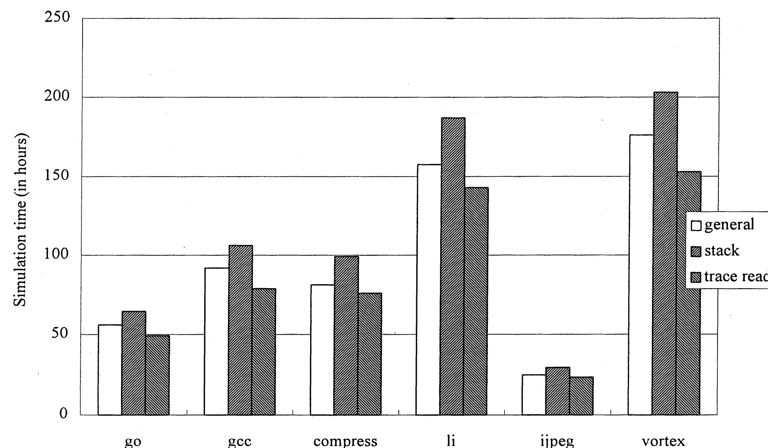

Fig. 10. Simulation times for the two algorithmic modes and the trace read time.

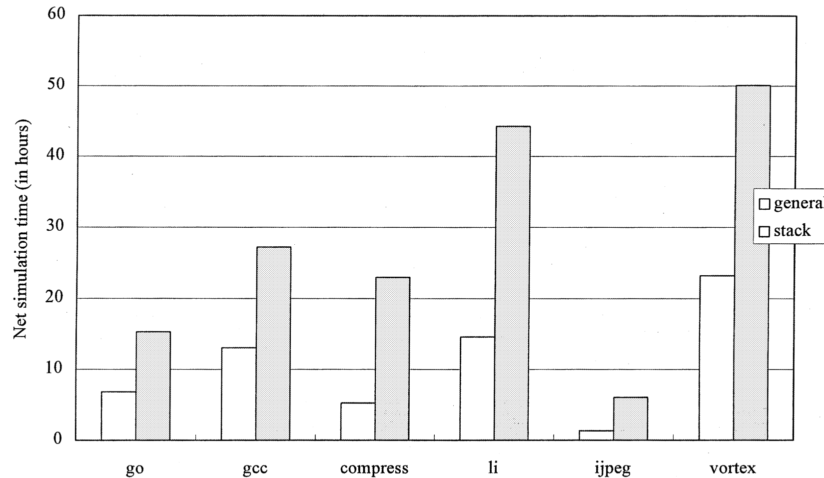

Fig. 11. Net simulation times for the two algorithms.

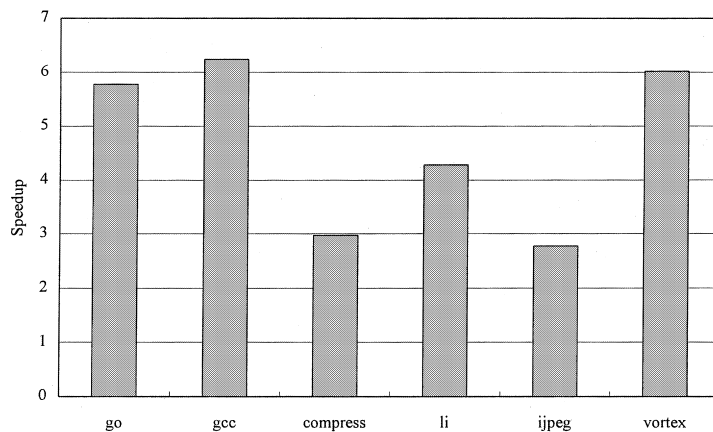

Fig. 12. Speedups of the net simulation time that stack BTB algorithm over the general BTB algorithm.

6. Conclusions

In this paper, we have developed a stack simulation technique for evaluating the performance of BTBs of different sizes. The stack simulation dramatically reduces the simulation time of multi-pass simulation. Simulation results show that the speedup provided by the stack simulation over the traditional method is 4.68 when simulate 13 different BTB sizes in one simulation pass. In this paper, we focused our attention on fully associative BTBs. With some modifications, our technique can be extended to evaluate the BTBs of different associativities.

As the design of BTB has increasing performance impact on modern microprocessors with ever higher instruction level parallelism, its design parameters need to be more carefully tuned in the early design phase. Furthermore, as the application program trace lengths grow, the trace read time will become a increasingly critical issue in BTB simulation. The single-pass BTB simulation technique proposed in this paper can dramatically speedup the simulation of BTB evaluation by amortizing the trace read time.

The state vector method can be used to facilitate the single-pass simulation for other applications whose behavior does not abide by the inclusion property, but can be maintained in a regular way. These applications include the memory consistency protocols and communication protocols. In the future, we plan to extend our one-pass simulation techniques to cover the applications of these applications and other more sophisticated BTBs.

References

Case, B., 1995. SPEC95 retires SPEC92, MicroProcessor Report, 21 August, pp. 11–14.

Chame, J., Dubois, M., 1993. Cache inclusion and processor sampling in multiprocessor simulations. *Proceedings of the* 1993 *ACM SIGMETRICS*, May 1993, pp. 36–47.

Dubey, P.K., Flynn, M.J., 1991. Branch strategies: modeling and optimization. IEEE Trans. Comput. 40 (10), 1159–1167.

Gecsei, J., 1974. Determining hit ratios for multilevel hierarchies. IBM Syst. J. 8 (4), 316–327.

Hill, M.D., Smith, A.W., 1989. Evaluating associativity in CPU caches. IEEE Trans. Comput. 38 (12), 1612–1630.

Laha, S., Patel, J.H., Iyer, R.K., 1988. Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Trans. Comput. 37 (11), 1325–1335.

Lee, J.K., Smith, A.J., 1984. Branch prediction strategies and branch target buffer design. Computer 7 (1), 6–22.

Mattson, R.L., 1971. Evaluation of multilevel memories. IEEE Trans. Magnet. Mag-7 (4), 814–819.

Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L., 1970. Evaluation techniques for storage hierarchies. IBM Syst. J. 9 (2), 78–117.

Reilly, J., 1995. SPEC discusses the history and reasoning behind SPEC95. SPEC Newsletter 7 (3), 1–3.

Silberman, G.M., 1983. Stack processing techniques in delayed-staging storage hierarchies. Commun. ACM 26, 999–1007.

Smith, M.D., 1991. Tracing with pixie. Technique Report of Stanford CA 94305-4070, April.

Sugumar, R.A., Abraham, S.G., 1995. Set-associative cache simulation using generalized binomial trees. ACM Trans. Comput. Syst. 13 (1), 32–56.

Tompson, J.G., Smith, A.J., 1989. Efficient. algorithms for write-back and sector memories. ACM Trans. Comput. Syst. 7 (1), 78–115.

Wang, W.H., Baer, J.L., 1991. Efficient trace-driven simulation methods for cache performance analysis. ACM Trans. Comput. Syst. 9 (3), 222–241.

Wu, Y.W., Muntz, R., 1995. Stack evaluation of arbitrary set-associative multi-processor caches. IEEE Trans. Parallel Distributed Syst. 6 (9), 930–942.

R-Ming Shiu received the B.E. degree in computer science and information engineering from the Fu Jen Catholic University, Taiwan, Republic of China in 1993. Currently he is pursuing the Ph.D. degree in computer science and information engineering at the National Chiao Tung University. His research interests include computer architecture and parallel processing.

Neng-Pin Lu received the B.E. and M.E. degrees from the National Chiao Tung University, Taiwan, Republic of China in 1989 and 1991, respectively, all in computer science and information engineering. Currently he is pursuing the Ph.D. degree in computer science and information engineering at the National Chiao Tung University. His research interests include computer architecture, interconnection network, and parallel processing.

Chung-Ping Chung received the B.E. degree from the National Cheng-Kung University, Taiwan, Republic of China in 1976, and the M.E. and Ph.D. degrees from the Texas A&M University in 1981 and 1986, respectively, all in electrical engineering. He was a lecturer in electrical engineering at the Texas A&M University while working towards the Ph.D. degree. Since 1986 he has been with computer science and information engineering at the National Chiao Tung University, Taiwan, Republic of China, where he is a professor. From 1991 to 1992, he was a visiting associate professor of computer science at Michigan State University. Currently, he is on leave and served as the director of Advanced Technology Center, Computer and Communications Research Laboratories, Industrial Technology Research Institute (CCL, ITRI), ROC, and then the consultant of CCL, ITRI. His research interests include computer architecture, parallel processing, and parallel compiler design.