Analysis of branch handling strategies under hierarchical memory system

Hung-Chang Lee
Feipei Lai

Indexing terms: Computers, Modelling

Abstract: Branch instructions form a significant fraction of executed instructions in a computer program, and handling them is thus a crucial design issue for any architecture. In a hierarchical memory system, branch handling not only causes the pipeline drain but also results in more instructions being executed, which can result in a higher miss ratio. These phenomena are relevant to the resolution of the branch condition stage, the generation of branch target stage, and how the branch is handled. A new branch handling model to quantify the overall effects is developed. Eight kinds of branch handling strategies currently used are examined by this model. Tradeoffs among them can be made on a statistical and theoretical basis. Also, the results of prediction brought forth by this model among some architectures are compared with those of evaluation to justify this model.

1 Introduction

Pipeline and memory hierarchy are two major factors in trying to improve computer performance. The memory hierarchy attempts to match the speed of the processor with the rate of information transfer or the bandwidth of the memory at the lowest level at a reasonable cost [1]. Pipeline processing offers an economical way to realise temporal parallelism by overlapping the execution of several different instructions [2]. If no interactions exist among instructions in the pipeline all the time, then the memory system can use a simple sequential prefetching technique to match the bandwidth required and let several instructions be executed simultaneously.

However, branch instructions interrupt this smoothly executing pipeline arrangement by requiring instructions from a new target for correct execution. As a result, the processor must flush out the instructions currently in the pipeline after a successful branch and wait for the fetching of the new target instructions from the memory system.

Techniques to reduce the delay which a branch causes include static and dynamic branch prediction, the branch target buffer, the delayed branch, branch bypassing and multiple prefetching, branch folding, and the prepare-to-branch instruction. Lilja [3] gave a probabilistic model to quantify the performance effects that a branch causes in a typical pipeline. However, because Lilja's model used only a branch penalty parameter to model the various branch handling strategies, it is not clear how this parameter is related to the design of a pipeline stage. Still less clear is that Lilja's model did not take into consideration the different branch handling strategies which would require more memory bandwidth from the memory system. The reason is that branch handling strategies often operate by a specific instruction fetch mechanism and this will demand more instructions from the memory system. In a hierarchical memory organisation, this quasi-request may cause a miss in this level and then propagate the request to a higher level, as shown in Fig. 1.

This article develops a new model of the branch penalty in terms of pipeline drain and the miss ratio caused by the branch handling disciplines. In this model, one can get a clearer view of the tradeoffs: how the decreased branch penalty and the increased miss ratio interact with each other. Comparisons among various branch handling disciplines are made using a trace simulation method. Also, the results of prediction and evaluation are compared to prove the accuracy of this model.

2 Global branch delay model and definition

A typical pipeline stage might consist of instruction fetch, instruction decode, generate effective operand address, operand fetch, execute, and result write back. To analyse the effects of branch instructions on the performance of a pipelined processor, the pipeline stages are abstracted into two parameters, namely, the distance between the resolution of branch condition and the instruction fetch stages (called d_c) and the distance between the generation of the target address of the branch instruction and the instruction fetch stages (called d_t). Let c denote the number of stages executed after the resolution of branch condition plus 1, therefore, the number of pipeline stages, as shown in Fig. 2, is $d_c + c$ (for most pipeline systems, c is less than 3).

Data dependence between instructions and resource conflicts is not considered in our model because instruction reorganisation [4] and more sophisticated hardware designs [5–6] have been proposed. To see better how the branch handling disciplines influence the memory system, we exclude the effect of data reference. This means that we separate the instruction and data storage. Therefore, the reference of data will not influence the instruction storage. This assumption simplifies but not weakens the model because some pipeline processors do separate the instruction and data storage (for example, mips R2000 [7], CLIPPER [8]).

Paper 8263E (C2, C3), first received 11th September 1989 and in revised form 13th May 1991

The authors are with the Department of Electrical Engineering and the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

Under the assumption mentioned above, branch delay can be divided into two parts, one is the pipeline drain because of the control transfer, the other is the branch handling strategies that cause a miss in the lowest hierarchical memory level (i.e. cache). A branch instruction divides the execution into two possible paths; the pipeline drain delay, d_c, if the resolution of branch condition is taken and, 0, if the resolution of branch condition is not taken. Assume the following parameters:

P_b: the probability that a particular instruction is a branch

P_t: the probability that a branch is taken

T_{cyc}: the average number of cycles required per instruction

After several delays to start up the pipeline, one non-branch instruction with a probability of $(1 - P_b)$ completes every cycle. The average number of cycles per instruction is then

$$T_{cyc} = (1 - P_b) + P_b[P_t(1 + d_c) + (1 - P_t)] \quad (1)$$

which reduces to

$$T_{cyc} = 1 + d_c P_b P_t \quad (2)$$

Eqn. 2 shows the average number of cycles required to execute an instruction if the lowest hierarchical memory level is infinite. However, when the limited storage in the lowest level is considered, eqn. 2 must be modified to reflect the delay due to the memory hierarchy. Assume an n-level memory hierarchy and the following parameters:

f_i: the missing-item fault rate at level i

B_i: the access time ratio between memory level $i + 1$ and i (include the waiting time due to memory conflict and delay in switching network)

In fact, the missing-item rate depends on diverse factors (such as capacity, fetch and replace algorithms, etc. [12]). In the model, we fix these factors and focus on how a branch influences this ratio. So f_i is a function of branch handling strategy and is expressed as f_i(BS). The effective access time (T_{eff}) in the ratio of the lowest hierarchical memory level is

$$T_{eff} = \sum_{i=1}^{n} \left[\prod_{k=1}^{i-1} f_k(BS)B_k \right](1 - f_i(BS)) \quad (3)$$

So the average number of cycles required per instruction under hierarchical memory organisation is

$$T_{eff} = (1 + d_c P_b P_t)^* \times \left(\sum_{i=1}^{n} \left[\prod_{k=1}^{i-1} f_k(BS)B_k \right](1 - f_k(BS)) \right) \quad (4)$$

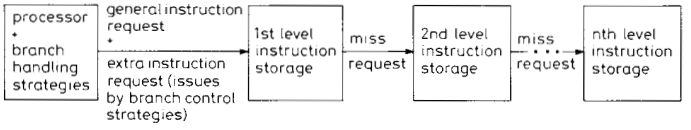

Fig. 1 *Branch handling strategy model in hierarchical memory system*

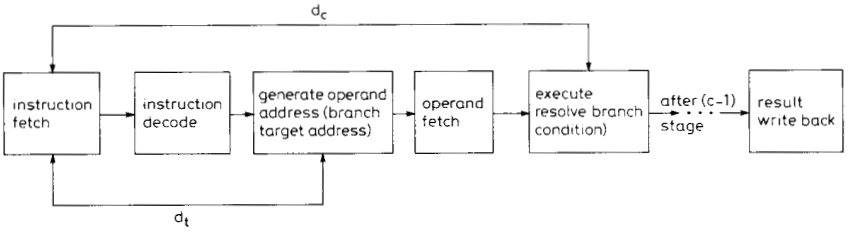

Fig. 2 *Abstracted processor pipeline model*

Because each program executed has its own P_b, P_t, and f_i, we define $P_b(j)$, $P_t(j)$, $f_i(j)$ for the specific jth program, the expected average cycle when running on a specific processor can be obtained by summing the m individual average times and then divide by m. The average performance (i.e. the inverse of expected average cycles on this processor) is

$$P_{ave} = \frac{1}{|\text{one stage period}|} * \frac{m}{\sum_{j=1}^{m} \left\{ (1 + d_c P_b(j) P_t(j))^* \times \left(\sum_{i=1}^{n} \left[\prod_{k=1}^{i-1} f_k(BS, j)B_k \right](1 - f_i(BS, j)) \right) \right\}} \quad (5)$$

For different processors, because of the variety of design philosophy, the comparisons among them are chaotic. Some processors are designed according to the philosophy that the more an instruction can do, the less instruction bandwidth it needs. Therefore, the execution time will be shortened. The processors of this kind form a category called CISC (complex instruction set computer) [9, 10]. For others, the concept is that the simpler the instruction is, the less instruction it needs. Arguments between these categories are beyond the scope of this article. However, both categories can be modelled as follows:

$$P_{ave} = \frac{d_c + c}{T} * \frac{m}{\sum_{j=1}^{m} \left\{ (1 + d_c P_b(j) P_t(j))^* \times \left(\sum_{i=1}^{n} \left[\prod_{k=1}^{i-1} f_k(BS, j)B_k \right](1 - f_i(BS, j)) \right) \right\}} \quad (6)$$

where T is the average execution time per instruction needed in either category. Once the major design philosophy is determined (CISC or RISC), one can go on to other design factors, including the number of pipeline stages, the branch handling discipline, the cache organisation, etc. Eqn. 6 shows that if we try to increase the number of pipeline stages, the pipeline drain delay increases (normally the value of c is less than 3, and cannot grow with the increase of the pipeline stage number). Therefore, the overall performance will not be satisfactorily improved unless the branch handling discipline is effective.

In summary, from the preceding discussion, when coming to the design of a processor, several issues should be addressed:

(i) the degree of matching between high level language and the underlying machine, and how complex the instructions should be [9–11]

(ii) the number of pipeline stages, where to put the calculation of branch target address stage, and the resolution branch condition

(iii) the branch handling strategy

(iv) the cache organisation to minimise the miss ratio

3 Modelling of branch handle strategies

This Section provides a theoretical interpretation of the branch strategies. Some general results are then outlined and developed. One of these has been used in the design of a new branch handling strategy used in MARS [21]. A virtual branch handling strategy is included for comparison and reflects the design in an extreme case. Before we go on, several definitions must be made:

Ib: the address where branch instructions are located

It: the branch target address for the branch instructions

Ib_i: the address for the ith branch instructions

It_i: the branch target address for the ith branch instructions

These definitions offer access to the extra memory demand so as to evaluate a specific branch handling discipline.

3.1 Virtual branch handle (VB)

A virtual branch handling strategy is modelled here to offer a base for comparison. It functions as follows (illustrated in Fig. 3): when branch instructions execute, they do nothing but wait for the resolution of the branch condition and then continue the execution following the resolved path. The pipeline drain due to branching is always d_c cycle. Therefore the average number of cycles required per instruction is

$$T_{cyc} = 1 + d_c P_b \tag{7}$$

The steps of the virtual branch handling mechanism are as follows:

(a) fetch the instruction; if the type of the instruction is branch, wait for the determination of branch condition

(b) i not the end of instruction fetch go to step a

The memory request is exactly the same as the instruction execution. Therefore, the cache miss ratio caused by this branch handling strategy will be minimal. Some cache organisation studies were based on this branch handling stragegy [12, 13].

3.2 Branch prediction

A branch control with branch prediction uses some additional logic to guess the outcome of a branch before it is determined. The processor may predict the branch path either statically or dynamically. In the static case, once the branch instruction is decoded, a decision is made to prefetch from the predicted path. As for the dynamical case, branch handling is the same except that the predicted path is chosen by the history of the instruction, rather than by the instruction itself [14].

A simplified diagram, shown in Fig. 4, indicates how the branch is in action. Drain slot with length d_t is needed because of the generation of the target address. If the prediction is correct, no further delay is required. On the other hand, if the prediction is not right, $(d_c - d_t)$ delays must be added to the drain delay. The average number of cycles required per instruction due to pipeline drain is

$$T_{cyc} = 1 + P_b[(1 - P_w)d_t + P_w d_c] \tag{8}$$

where P_w is the probability of a wrong prediction. The above equation can be reduced to

$$T_{cyc} = 1 + P_b d_t + P_b P_w(d_c - d_t) \tag{9}$$

If the parameters d_t, d_c, and P_w are given, the steps of this branch handling are as follows:

(i) fetch the instruction; if the type of the instruction is branch:

(a) wait for the determination of branch condition (d_t cycles waste)

(b) predict the branch path

(c) for $i = 1$ to $(d_c - d_t)$ fetch instruction $(Ib + i)$ if the prediction is not taken, and fetch instruction $(It + i)$ if the prediction is taken

(ii) if not the end of instruction go to step (i)

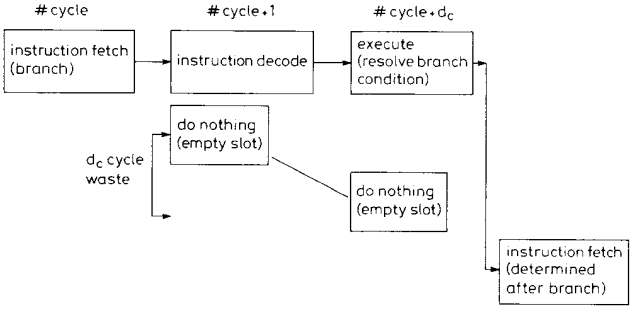

Fig. 3 *Virtual branch pipeline drain delay*

The extra memory request of this method, compared with that of the virtual branch method, is the summation of the instruction fetches, and each wrong predicted guess will cause a length of $(d_c - d_t)$ instruction fetches. These overhead memory accesses can be expressed as follows:

$\{It_i, It_i + 1, \ldots, It_i + (d_c - d_t)$: Assume the ith branch prediction taken but wrong,

$Ib_i, Ib_i + 1, \ldots, Ib_i + (d_c - d_t)$: Assume the ith branch prediction not taken but wrong$\}$.

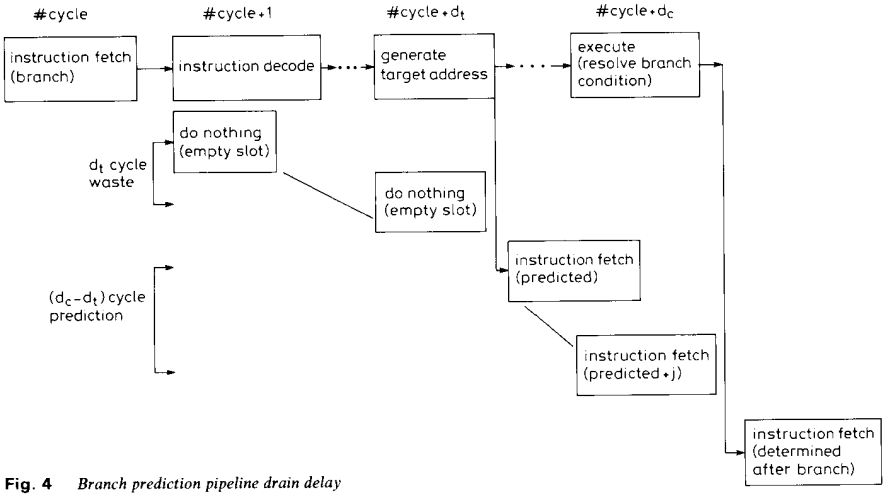

Fig. 4 *Branch prediction pipeline drain delay*

3.3 Delayed branch

With a processor using a delayed branch discipline, instructions after the branch (called delay slot) still execute whether the branch is taken or not. The compiler detects dependencies in the instruction stream and allows a few instructions preceding the branch to be moved into the delay slots [15, 16].

As before, we can model the effects of using a delayed branch architecture (shown in Fig. 5). Because the actual path is not known until the condition resolution stage, a delayed slot of length d_c is needed. The average number of cycles required to execute an instruction due to pipeline drain is

$$T_{cyc} = 1 + d_c P_b P_{nop} \tag{10}$$

This equation uses the probability that the delayed slot must be filled with no-operation. This value can be calculated by summing all the delayed slots filled with no-operations divided by the total delayed slots. Let $S_j(i)$ equal 1 if the jth delay slot of the ith branch instruction is filled with a no-operation instruction, and equal 0, otherwise. Assume there are N branch instructions in the program. Then,

$$P_{nop} = \left(\sum_{i=1}^{N} \sum_{j=1}^{d_c} S_j(i) \right) \Big/ d_c * N \tag{11}$$

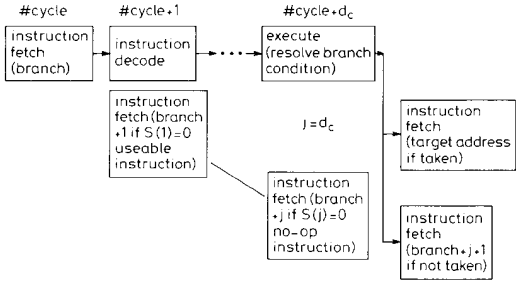

Fig. 5 *Delayed branch pipeline drain delay*

Because the compiler takes the responsibility of providing the no-operation instruction, the steps of the delayed branch mechanism are the same as virtual branch strategies. When compared with the VB strategy, the additional memory request is the reference to the storage filled with no-operations. And these extra memory accesses can be grouped as the following:

$$\{Ib_i + j\text{: if the } S_j(i) \text{ equal } 1\} \tag{12}$$

A debatable feature of the delayed branch is the delayed slot with nullification, which is proposed in the Hewlett-Packard precision architecture [17]. This feature allows a branch instruction to dynamically force the delay slot after the branch to be a no-operation. Architectures with this feature will waste d_c cycles when nullification acts. If the procedure resolves the branch condition and the nullification does not happen, the pipeline drain delay is $d_c^* P_{nop}$. Consequently, the average number of cycles required to execute an instruction due to pipeline drain is

$$T_{cyc} = 1 + P_b d_c (P_{null} + (1 - P_{null}) P_{nop}) \tag{13}$$

The term P_{null} is the probability that the delay slots are nullified. This probability equals the one that the branch is taken, P_t, if the instruction is of the nullify-on-branch-taken type. Otherwise, P_{null} equals $(1 - P_t)$.

The memory request not used is the delay slot if nullification is true and the instruction filled with no-operation, otherwise. Therefore, these extra memory accesses can be grouped as the following:

$\{Ib_i, Ib_i + 1, \ldots, Ib_i + d_c$: if the ith branch instruction takes nullification,

$Ib_i + j$: if the ith branch instruction does not take nullification and $S_j(i)$ equals $1\}$ (14)

3.4 Branch target buffer (BTB)

The BTB is a special, selective cache to supply the predicted target instruction directly. Each entry in the BTB

contains three fields, one to keep the address of the previously executed instruction, another to keep the history information which is used to make a prediction, and the other to keep the target address for the branch instruction [18]. The BTB functions as follows: when the pipeline decodes a branch instruction, it searches the BTB for the address of that instruction. If there is a match, then the prediction field is combined with the predicting function to make a prediction. If the prediction is that the branch will be taken, the target instruction field is supplied to the next instruction. If the instruction is not in the BTB, the processor waits for the fetch of the target instruction. After the branch is resolved, the BTB is updated with the corrected history information and the target instruction.

We can extend the pipeline drain model to simulate the effects of adding a BTB. Assume that instruction decode is the next stage after instruction fetch. Because the BTB supplies the target instruction, a pipeline drain will not happen if the BTB hit occurs and the prediction is right. Otherwise, a delay of d_c length is required if BTB hit occurs and the prediction is wrong. Under the BTB miss condition, d_t delay is needed to calculate the target address, and another $(d_c - d_t)$ delay is wasted if the prediction is wrong. Then, the average number of cycles required to execute an instruction is

$$T_{cyc} = 1 + P_b[(1 - f_b)(1 - P_w)*0 + (1 - f_b)P_w d_c + f_b(1 - P_w)d_t + f_b P_w d_c] \quad (15)$$

where f_b is the probability that a BTB miss occurs. The above equation is reduced to

$$T_{cyc} = 1 + P_b[f_b d_t(1 - P_w) + d_c P_w] \quad (16)$$

The steps of branch target buffer mechanism are as follows:

(i) fetch the instruction; if the type of instruction is branch:

(*a*) predict the branch path

(*b*) access the branch target buffer; if there is a hit then: for $i = 1$ to d_c fetch instruction $(Ib + i)$ if the prediction is not taken, fetch instruction $(It + i)$ if the prediction is taken; else wait for the calculation and access the branch target for $i = 1$ to $(d_c - d_t)$ fetch instruction $(Ib + i)$ if the prediction is not taken, fetch instruction $(It + i)$ if the prediction is taken

(ii) if not the end of instruction fetch go to step (i)

When compared with those of the VB method, the additional memory request happens when the prediction is wrong. For the branch target miss case, if the prediction is that the branch will be taken and the prediction is not correct, then the wasted instruction fetch ranges from the target address to the target address plus $(d_c - d_t)$. Otherwise, the wasted instruction fetch ranges from the address of the branch instruction to the value plus $(d_c - d_t)$. In the case of the branch target buffer hit, the situation is the same as the above one except d_c is used instead of $(d_c - d_t)$. These extra memory accesses can be represented as the following:

$\{It_i, It_i + 1, \ldots, It_i + (d_c - j)$: if the ith branch is assumed to be taken but wrong, and $j = 0$ for branch target buffer fit and $j = d_t$, otherwise,
$Ib_i, Ib_i + 1, \ldots, Ib_i + (d_c - j)$: if the ith branch prediction is assumed to be not taken but wrong, and $j = 0$ for the branch target buffer fit and $j = d_t$, otherwise.$\}$ (17)

3.5 Branch bypass and multiple prefetch

A pipeline suffers a branch penalty because it must wait for the determination of a condition branch. A brute force approach to this problem is to replicate the stages so that both the sequential and target instruction can be in the pipeline at the same time. Therefore, a pipeline which contains one branch instruction must fetch two execution paths. If another branch instruction in both paths appears before the resolution of the previous branch instruction, four paths must be fetched. It seems that under this approach, the pipeline drain is zero because it takes all possible paths. However, a delay of length d_t is still required because the branch target can not be fetched until the address of the target address is calculated. The average number of cycles required to execute an instruction is

$$T_{cyc} = 1 + d_t P_b \quad (18)$$

How the branch bypass mechanism works is stated as follows:

(i) fetch the instruction; if the type of instruction is branch:

(*a*) wait for the calculation of branch target address

(*b*) for $i = 1$ to $(d_c - d_t)$ fetch instruction $(Ib + i)$ and $(It + i)$

(ii) if not the end of instruction go to step (i)

This multiple prefetch technique always fetches the two possible paths when it executes the branch instruction. Therefore once the condition is determined, one of the two paths must be discarded. This discarded memory causes the additional memory request compared with the VB strategy. These extra memory accesses can be represented as the following:

$\{It_i, It_i + 1, \ldots, It_i + (d_c - d_t)$: if the ith branch is not taken,

$Ib_i, Ib_i + 1, \ldots, Ib_i + (d_c - d_t)$: if the ith branch is taken$\}$ (19)

3.6 Branch folding

A technique called branch folding which is used in the AT&T CRISP processor can reduce the branch penalty to zero [19]. CRISP consists of two logically separated machines, a prefetch and decode unit and an execution unit, connected by a decoded instruction cache. The former unit fetches the encoded instruction and places it into the decoded instruction cache. Each decoded instruction is associated with the next address in the next-address field. The prefetch and decode unit recognises an unconditional branch instruction following a nonbranch instruction and folds the target address of the branch instruction into the next-address field of the nonbranch instruction. Therefore, the unconditional branch can be eliminated effectively. When a folded conditioned instruction executes, the execution unit selects one of the two next-address paths. When the pipeline resolves the branch, it either throws away the unselected path address or flushes itself and restarts execution at the unselected path.

Because the execution unit uses the decoded instruction to execute, the pipeline drain is minus 1 for unconditional branches. The performance of conditional branches is the same as that in the model of branch prediction with P_S (the probability that selection is not correct) instead of P_w. Furthermore, a delay slot to calculate the target address is also eliminated because of the branch folding. The average number of cycles required

per instruction is

$$T_{cyc} = 1 - P_{b\_u} + P_{b\_c} P_s d_c \quad (20)$$

Where the term P_b is further divided into two parameters: one is the probability of unconditional branch (P_{b_u}); the other is the probability of conditional branch (P_{b_u}).

The additional memory request is almost the same as that of branch prediction except that the extra fetch length is d_c rather than $(d_c - d_i)$. These memory accesses can be represented as the following:

$\{It_i, It_i + 1, \ldots, It_i + d_c$: Assume the ith branch selection to be taken but wrong,
$Ib_i, Ib_i + 1, \ldots, Ib_i + d_c$: Assume the ith branch selection not to be taken but wrong$\}$ (21)

3.7 Prepare-to-branch instruction

The Texas Instruments ASC computer uses the prepare-to-branch instruction to forewarn of the execution of a branch instruction [20]. The request of the target address is then passed to the control unit so that the unit can begin instruction prefetch from the address. Inserting prepare-to-branch instruction will increase the number of instructions executed, and thus decrease the performance of the processor. The compiler or programmer can help to join another instruction with the prepare-to-branch instruction, assuming that the execution of these two instructions do not use the same hardware resource. This capabilities of the compiler can be modelled as a factor called the joint probability (P_j), namely, the probability of a compiler finding the instruction to join with the prepare-to-branch instruction. However, cycles of length d_c must be wasted until the determination of the branch condition. Therefore, the average number of cycles required per instruction, considering the compiler effect, is

$$T_{cyc} = 1 + (1 - P_j)P_b + P_b d_c \quad (22)$$

In this model, the branch target address is always fetched. If the condition decides to take branching, this fetch is not additional. Otherwise, it is an extra request to the memory system. Consequently, these memory accesses can be expressed as the following:

$\{It_i$: if the ith branch is not taken$\}$ (23)

3.8 Branch peephole

The techniques discussed so far imply that pipeline slots before the generation of the target address waste most of the time. The MARS processor board design [21, 22] uses a technique called branch peephole which determines the target address in the same stage as instruction fetch. It then fetches both the target instruction and the sequential one, the same as multiple prefetch. However, unlike the multiple prefetch that uses a replicating stage, branch peephole uses a combination of delaying, fast-compare-and branch, and squashing techniques to reduce the pipeline drain. A super-zero-delay jump will occur after the processor peeps out the existence of an unconditional branch, calculates the target address, and sends out the target instruction of the unconditional branch. On the whole, the effect of the super-zero-delay jump is to absorb the branch instruction and directly issue the target instruction. The branch probability (P_b) is divided into four parameters to reflect this architecture, namely, the probability of unconditional branch (P_{b_u}), delayed branch (P_{b_d}), fast-compare-and-branch (P_{b_f}), and squashable delayed branch (P_{b_s}). Therefore the average number of cycles required is

$$\begin{aligned} T_{cyc} = {} & 1 - P_{b\_u} + d_c P_{b\_d} P_{nop} \\ & + (d_c - 1)P_{b\_f} P_{nop} \\ & + P_{b\_s} d_c (P_{null} + (1 - P_{null})P_{nop}) \end{aligned} \quad (24)$$

The objective of fast-compare-and-branch is to handle a simple condition test (such as two registers equal, non-equal, etc.). The branch condition can be decided one stage earlier than the delay branch, therefore the number of the delay slots is less than that of the delay branch by one.

Because the compiler takes the responsibility of using either the three kinds of branch instruction and filling the no operation instruction, the steps of the branch peephole mechanism are stated as follows:

(i) fetch the instruction; if the type of the instruction is branch:

(*a*) fetch both the next instruction and the branch target

(*b*) if the branch type is unconditioned provide the processor with the target address

(ii) if not the end of the instruction fetch go to step (i)

The addition memory request in this model is to discard either the target or sequential path and the combined techniques of the delayed, the squashable delayed, and the fast-compare-and-branch. These can be represented as the following:

$\{Ib_i$: If the ith branch is taken,
It_i: if the ith branch is not taken,
$Ib_i + j$: if the ith branch is a delay or fast-compare branch and $Sj(i)$ equals 1,
$Ib_i, Ib_i + 1, \ldots, Ib_i + d_c$: if the ith branch is squashable branch and takes nullification,
$Ib_i + j$: if the ith branch is squashable instruction and does not take nullification and $S_j(i)$ equals 1$\}$ (25)

4 Evaluation of branch strategies

Evaluation among various branch strategies is a tough job because each branch handling strategy has its own parameters and these parameters are adjustable owing to the advance of software techniques. To deal fairly with these strategies, several sets of parameters are adopted and listed in Table 1.

The evaluations, based on the parameters reported, are classified into two parts: one is the increased instruction fetch miss compared with VB strategies in the lowest level of the memory hierarchy (here we assume level 2 contains all the instructions needed and therefore there is no miss in this level and the higher one); the other is the pipeline drain delay due to the pipeline stage design itself (mainly the determination of condition resolving stage and the branch target generation stage). Both parts of the evaluation will be explained in the following subsection.

4.1 Relative memory hierarchy penalty

Our analysis of how branch handling strategies affect the cache miss ratio is based on extensive trace driven simulation. Fig. 6 shows the flow chart of the analysis process. As shown by the Figure, program traces are preprocessed by the branch handling strategies which take the value d_c as a parameter and create the total memory bandwidth required from the processor viewpoint. Then the real trace from the processor's viewpoint is simulated by the

Table 1: Associated parameters for branch handling strategies

Strategies	Associated parameters
Programs behaviour on the branch instruction	the probability of a branch (P_b) is 12% to 38%, typical 25% [18, 23] the probability of a taken branch (P_t) is 0.6 to 0.7 [23]
Branch prediction	the probability of a wrong prediction (P_w) is 4% to 20% [18]
Delay branch	the probability of no operation (P_{nop}) in 1st, 2nd, or more slots are 0.3, 0.75, and 1 [16]
Branch target buffer	the probability of BTB miss ratio (f_b) is 53%, 17%, and 7% for 16, 64, 256 entires [18] the probability of a wrong prediction (P_w) is 4% to 20% [16]
Branch folding	the probability of a unconditional branch (P_{b_u}) is about $0.2*P_b$ [19] the probability of a wrong selection (P_s) is 6% to 26% [19]
Branch peephole [23]	the probability of fcb (P_{b_f}) is about $\frac{1}{6}*P_b$ the probability of dcb (P_{b_d}) is about $\frac{1}{3}*P_b$ the probability of scb (P_{b_s}) is about $\frac{1}{2}*P_b$ the probability of nullification (P_{null}) is 0.4

cache organisation model and the miss ratio is reported after that.

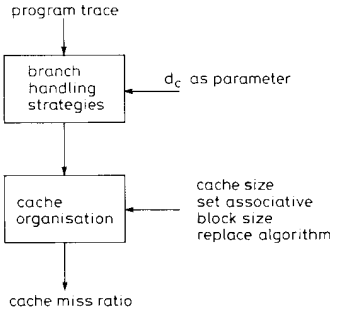

Fig. 6 *Trace driven simulation flow chart*

4.1.1 Trace driven simulation: The trace driven simulation technique drives the simulation model of a system with a continuous trace of event. As far as the memory system is considered, the trace of event is the sequence of (virtual) addresses accessed by a computer program. Trace driven simulation is a very good way to study the memory hierarchy compared to a mathematical model or random number driven simulation. There are a number of ways in which trace driven simulation is not perfect [13], and thus the absolute value of the miss ratio is hard to determine. However, this situation does not cause much of a problem in our analysis because what is of concern is the relative change in miss ratio caused by the branch handling strategy.

4.1.2 Benchmarks: A number of programs are used in the memory hierarchy penalty simulation. Six benchmarks were selected as a workload sample. Four of them from mips@R2000, the remaining from SPARC workstation. The characters of these benchmarks are listed in the following:

C_frontend: C compiler frontend processing from mips

C_backend: C compiler backend processing from mips

NROFF_frontend: a unix text formatting frontend processing from mips

NROFF_backend: a unix text formatting backend processing from mips

I_sim: instruction level simulation processing from SPARC

BOOL_expression: a new Boolean expression processing from SPARC

4.1.3 Results: The miss ratios caused by branch handling strategies and different cache organisations are reported in Reference 23. Because the miss ratios can be greatly cut down by increasing bandwidth and hardware cost, it is not fair to compare them with each branch handling strategy associated with different cache organisations. Therefore, we chose a set of cache organisations used in the CLIPPER cache memory (that is 2-way set-associative, 4 work block size, and LRU replace algorithm in each set, etc.) and used the value d_c and cache size as parameters. Figs. 7 and 8 show the relative miss ratio caused by branch handling strategies under the CLIPPER cache organisation with the cache size and the d_c value as parameters.

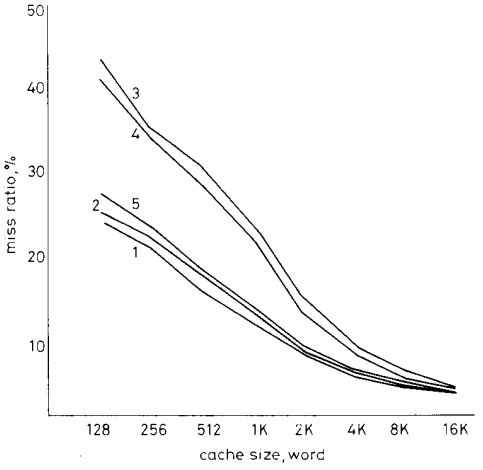

Fig. 7 *Cache miss ratio for branch handling strategies* $d_c = 2$

1 virtual branch
2 BP, BTP, BF
3 delayed branch
4 branch peephole
5 branch bypass

4.2 Pipeline drain penalty

Pipeline drain penalty caused by the branch handling discipline depends mainly on global parameters (e.g. d_c and d_t) and some individual parameters associated with the branch handling strategies (e.g. the prediction precision P_w used in branch prediction, BTB, and branch folding, the delay slot filled with something used in delayed branch, and so forth). Figs. 9 and 10 show the pipeline drain delay for various branch handling methods assuming that $d_t = 1$ and $d_t = 3$, respectively.

As we see from the two Figures, several conclusions can be made for the pipeline drain delay. BTB and branch folding cause minimal drain delay and the delay variance of the d_c value is low. However, if we consider the corresponding hardware cost for a small value of d_c, the performance of delay branch and branch peephole is no worse than that of the BTB and branch folding.

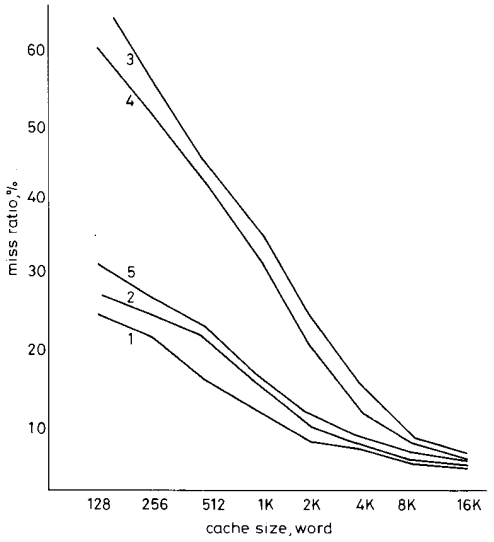

Fig. 8 *Cache miss ratio for branch handling strategies* $d_c = 5$
1 virtual branch
2 BP, BTB, BF
3 delayed branch
4 branch peephole
5 branch bypass

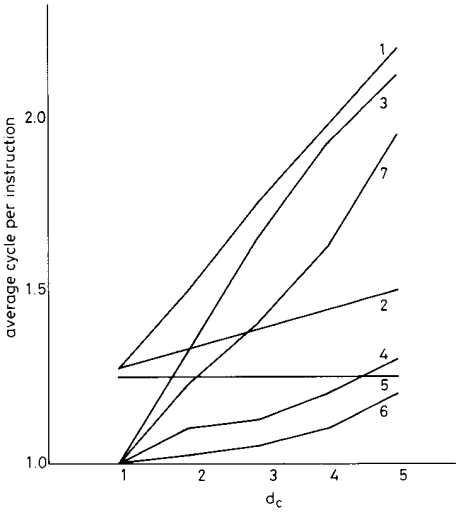

Fig. 9 *Average cycle per instruction needed for branch handling disciplines* $d_t = 1$
1 virtual branch
2 branch prediction
3 delayed branch
4 branch target buffer
5 branch bypass
6 branch folding
7 branch peephole

5 Prediction

We find it rather interesting to apply our branch model to various existing processors and compare the real performance of these machines. SPARC, mips@R2000, and MARS are selected as examples because of the same design philosophy within them (i.e. RISC type).

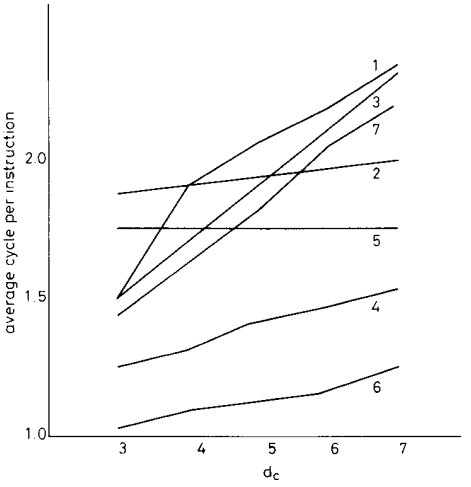

Fig. 10 *Average cycle per instruction for branch handling disciplines* $d_t = 3$
1 virtual branch
2 branch prediction
3 delayed branch
4 branch target buffer
5 branch bypass
6 branch folding
7 branch peephole

SPARC is a register window based RISC processor with 4 pipeline stages (namely, instruction fetch, execution, memory access, write back). This architecture uses the delay branch to handle control transfer and the value of d_c in company with the pipeline design is 1 (shown in Fig. 11*a*)

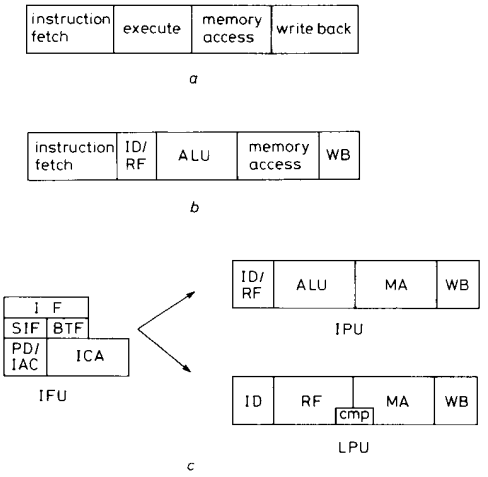

Fig. 11 *Pipeline stages for SPARC, mips R2000 and MARS*
a SPARC pipeline stage
b mips R2000 pipeline stage
c three kinds of pipeline stage in MARS

Mips@R2000 is a commercial version of MIPS-X. It has only 32 registers to reduce the register access cycles and uses 5 pipeline stages (i.e. instruction fetch, instruction decode and register fetch, ALU, memory access, and write back). This architecture uses fcb and the delay branch method to deal with the branch instruction and the value of d_c is 2 (shown in Fig. 11*b*).

MARS is a multiprocessor project developed at the National Taiwan University with the goal to build a workstation for VLSI/CAD and artificial intelligence applications through RISC and multiprocessing approaches. Within the processor board, there are four function units: namely, the instruction fetch unit, the integer processing unit, the floating point processing unit, and the list processing unit. Three kinds of pipeline stages (shown in Fig. 11*c*) are used by different units and work synchronously. MARS uses branch peephole and the value of d_c is 1.

The results of prediction among three processors are shown in Table 2 with 4K instruction cache size. Table 3 shows the performance of five benchmarks executed by these three processors. These five real benchmarks are 20 queens, ackermann(3,8), shell sort, and Hanoi tower ($n = 18$ and $n = 24$).

Table 2: Prediction performance among SPARC, mips R2000 and MARS

Processor	Comparison items			
	Cache ratio caused by branch handle discipline	Average cycle per instruction	Modelled performance	Relative performance
SPARC	6.5%	1.12	0.75	0.47
mips R2000	8.7%	1.27	0.72	0.6
MARS	6.9%	1.03	1	1

Modelled performance: calculated by the following formulas:

SPARC = {1/(*T*/4*1.12*(1 + 10*6.5%))}/MARS

R2000 = {1/(*T*/5*1.27*(1 + 10*8.7%))}/MARS

MARS = {1/(*T*/5*1.03*(1 + 10*6.9%))}

Here the cache miss penalty is assumed 10 cycles.
Relative performance: it is calculated with the real machine cycle time and the values are 100, 60, and 50 ns, instead of *T*/4, *T*/5, and *T*/5 for SPARC, R2000, and MARS.

Compare the results of prediction between R2000 and MARS, they are rather accurate because both processors share the same execution pipeline stage except that the MARS uses the branch peephole strategy instead of delayed branch. However, comparing the results of prediction and benchmarks between SPARC and MARS, they vary a lot because the register window structure of SPARC prolongs the cycle time. Performance in Table 2 also shows that if a processor using delayed branch increases the d_c value, the result would be negative.

6 Conclusion

A new model for the branch handling penalty under hierarchical memory structure is established. With this model, one can evaluate branch handling strategies and optimally choose one of them. Among the disciplines mentioned above, the delayed branch method causes maximal miss ratio and must be used with care. The reason is that delayed branch fills the execution with no-operation instructions which have nothing to do with program execution and occupy the cache space. As discussed in the preceding Section, the branch handling penalty can be divided into two major parts, one is the pipeline drain delay caused by the control transfer, and the other is the cache miss resulting from the branch handling discipline request. For a small size of cache, the delay caused by cache miss dominates. Therefore, it is important to choose a branch handling method which brings about the lowest cache miss ratio. The branch handling disciplines using the prediction function (e.g. BTB, branch prediction, and branch folding) would be the best choice. For a large size of cache, cache miss is almost the same among the various branch strategies and this means that pipeline drain delay must be considered carefully to minimise the branch handling penalty. If a long pipeline stage is considered in the design of a new processor, the BTB handling and the branch folding methods would be the best. However, the extra hardware cost for BTB (i.e. buffer for branch target) and the branch folding (i.e. the encoded instruction, the decoded instruction cache, and big number of bus size) is not cheap. Therefore, a simple solution exists given a small value of pipeline stage (e.g. the delay branch, the branch peephole). Among these two methods, branch peephole will be optimal because it causes less pipeline bubble.

7 References

1 CHOW, C.K.: 'On optimization of storage hierarchies', *IBM J. Res. Dev.*, 1974, pp. 194–203
2 HWANG, K., and BRIGGS, F.A.: 'Computer architecture and parallel processing' (McGraw-Hill, inc., 1984)
3 LILJA, D.J.: 'Reducing the branch penalty in pipelined processors', *IEEE Computer*, 1988, **21**, (7), pp. 47–55
4 HENNESSY, J., and GROSS, T.: 'Postpass code optimization of pipeline constraints', *ACM Trans. Program. Lang. Syst.*, 1983, **5**, (3), pp. 422–448
5 KROFT, D.: 'Lockup-free instruction fetch/prefetch cache organization'. Proc. 8th Symp. on Computer Architecture, June 1981, pp. 81–87
6 DUBOIS, M., SCHEURICH, C., and BRIGGS, F.: 'Memory access buffering in multiprocessors'. Proc. 13th Symp. on Computer Architecture, June 1986, pp. 434–442
7 MOUSSOURIS, J., CRUDELE, L., FREITAS, D., HANSEN, C., HUDSON, E., MARCH, R., PRZYBYLSKI, S., RIORDAN, T., ROWEN, C., and VAN'T HOF, D.: 'A CMOS processor with integrated system functions'. Proc. of COMPCON Spring 86, Mar. 1986, pp. 126–131
8 HUNTER, C.B.: 'Introduction to the CLIPPER architecture', *IEEE Micro*, 1987, **7**, (4), pp. 6–26

Table 3: Benchmarks result among SPARC, R2000 and MARS

Benchmarks	Execution times and ratio				
	Execution time in seconds			MARS/SPARC	MARS/R2000
	MARS	SPARC	R2000		
queen	13.7	49.42	22.81	0.28	0.6
acker	1.36	16.20	2.42	0.08	0.56
shell	0.0076	0.02	0.01	0.39	0.78
hanoi_18	0.1835	0.32	0.33	0.57	0.56
hanoi_24	11.7	21.62	21.16	0.54	0.55
Geometric mean				0.31	0.61

9 COLWELL, R.P., HITCHCOCK III, C.Y., JENSEN, E.D., SPRUNT, H.M.B., and KOLLAR, C.P.: 'Computers, complexity, and controversy', *IEEE Computer*, 1985, **18,** (9), pp. 8–19
10 HENNESSY, J.L.: 'VLSI processor architecture', *IEEE Trans.*, 1984, **C-33,** (12), pp. 1221–1246
11 MOKHOFF, N.: 'RISC and CISC features', *Comput. Des.*, 1986, **25,** (7), pp. 22–25
12 SMITH, A.J.: 'Cache memories', *ACM Computing Surveys*, 1982, **14,** (3), pp. 473–530
13 SMITH, A.J.: 'Line (block) size choice for CPU cache memories', *IEEE Trans.*, 1987, **C-36,** (9), pp. 1063–1075
14 SAKAMURA, K.: 'Architecture of the TRON VLSI CPU', *IEEE Micro*, 1987, **7,** (2), pp. 17–31
15 RADIN, G.: 'The 801 minicomputer', *IBM J. Res. Dev.*, 1983, pp. 237–246
16 McFARLING, S., and HENNESSY, J.: 'Reducing the cost of branches'. Proc. 13th Symp. on Computer Architecture, June 1986, pp. 396–403
17 BIRNBAUM, J.S., and WORLEY, W.S.: 'Beyond RISC: high-precision architecture'. Proc. of COMPCON Spring 86, Mar. 1986, pp. 40–47
18 LEE, J.K.F., and SMITH, A.J.: 'Branch prediction strategies and branch target buffer design', *IEEE Computer*, 1984, **17,** (1), pp. 6–22
19 DITZEL, D.R., and McCLELLAN, H.R.: 'Branch folding in the CRISP microprocessor: reducing branch delay to zero'. Proc. 14th Symp. on Computer Architecture, 1987, pp. 2–9
20 WATSON, W.J.: 'The TI ASC — A highly modular and flexible supercomputer architecture'. Proc. AFIPS Fall Joint Computer Conf., Vol. 41, AFIPS Press, Dec. 1972, pp. 221–228
21 JANG, G.-S., LAI, F., LEE, H.-C., MAA, Y.-C., PARNG, T.-M., and TSAI, J.-Y.: 'MARS — multiprocessor architecture reconciling symbolic with numerical processing, a CPU ensemble with zero-delay branch/jump'. Intel. Symp. on VLSI Technology, Systems, and Applications, May 1989, pp. 365–370
22 MAA, Y.-C., LAI, F., LEE, H.-C., and PARNG, T.-M.: 'Instruction fetch unit in RISC architecture'. Intel. Symp. on Computer Architecture and Digital Signal Processing (CA-DSP), IEE, Oct. 1989, pp. 180–185
23 LEE, H.-C.: 'Architectural tradeoffs in the design of MARS'. PhD dissertation in preparation, National Taiwan University, 1991