

Reducing Branch Delay to Zero in Pipelined Processors

Antonio M. Gonzalez and Jose M. Llaberia

Abstract—A mechanism to reduce the cost of branches in pipelined processors is described and evaluated. It is based on the use of multiple prefetch, early computation of the target address, delayed branch, and parallel execution of branches. The implementation of this mechanism using a Branch Target Instruction Memory is described. An analytical model of the performance of this implementation is presented, which allows us to measure the efficiency of the mechanism with a very low computational cost. The model is used to determine the size of cache lines that maximizes the processor performance, to compare the performance of the mechanism with other schemes, and to analyze the performance of the mechanism with two alternative cache organizations.

Index Terms—Branch instructions, branch target instruction memory, computer architecture, instruction cache memory, instruction dependencies, performance evaluation, pipelined processors.

I. INTRODUCTION

Pipelining is a technique frequently used in the design of processors in order to increase their performance by executing several instructions simultaneously. However, the efficiency brought by pipelining may be significantly reduced by hazards caused by instruction dependencies. Those due to branches, also known as control dependencies, may have a severe impact on the processor performance since these instructions account for a high percentage of executed instructions.

The present work focuses on the design and evaluation of mechanisms for reducing the negative effect due to hazards produced by branch instructions in pipelined processors. We present and evaluate a mechanism called COBRA (Cost Optimization of BRanches) which eliminates most of the hazards caused by branches and allows the processor to execute branches in parallel with the rest of instructions. In this way, the cost of most branches can be reduced to zero. To evaluate the performance of this mechanism, a mathematical model of COBRA is developed and used to tune the design.

The rest of this paper is organized as follows. Section II is a review of previous work on reducing the cost of branches. Section III describes the COBRA mechanism. A mathematical model of COBRA is presented in Section IV. Section V discusses the performance of COBRA and compares it with other schemes.

II. REDUCING THE COST OF BRANCHES

Several mechanisms have been proposed in the literature in order to reduce the cost of branches [14], [15]. They make use of either one or several of the five techniques described briefly below.

a) *Delayed branch*. A delayed branch with length equal to n is a branch instruction that takes effect after the execution of the n instructions below it. The compiler is responsible for benefiting from this mechanism because it is in charge of finding the instructions that must be scheduled in the n delay slots. Among others, the mechanism is used by the MIPS R3000 [16].

If the processor is provided with the possibility of nullifying the execution of the instructions in the delay slots, the number of delay

Manuscript received June 15, 1990; revised March 15, 1992. This work was supported in part by the Comision Interministerial de Ciencia y Tecnologia (CICYT) under grant TIC89/0300.

The authors are with the Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain.

IEEE Log Number 9202843.

slots that can be profitably used increases. This mechanism is called *delayed branch with squashing*. This is the case of the SPARC [4].

b) *Early execution of branches*. Hazards caused by a branch can be reduced by executing some of its operations in advance. For example, the Motorola 68040 [3] has an additional adder to compute the target address as soon as a branch is fetched.

c) *Branch prediction*. Another way of advancing the possible result of a branch is to predict it. As an example we could mention the Intel 80960—Next Generation [11]. In this processor, each branch instruction includes a bit that is used by the compiler to predict the most likely result of the branch.

d) *Multiple prefetch*. It is based on prefetching after each branch some of the instructions at the beginning of each possible path. In this way, when the result of the branch is known, the fetch stage has been already performed, regardless of the taken path. This technique is implemented in the Intel i486 [2].

e) *Parallel execution of branches*. The preceding techniques try to reduce the negative effect caused by control dependencies. A greater increase in performance can be achieved if the execution of branches is completely overlapped with the execution of the rest of instructions. This is the case of the IBM RS/6000 [9].

In many processors we find that several techniques from those types listed above are combined in order to build a particular mechanism to reduce the cost of branches. This is the case of the COBRA mechanism.

III. COBRA MECHANISM

In this section we present the COBRA mechanism. It was devised for pipelined processors with any number of stages and with condition codes. A preliminary study of the COBRA mechanism was presented in [7], [8], and [6].

The COBRA mechanism combines several techniques to allow the processor to execute branches in parallel with the rest of instructions. These techniques are: Early computation of the target address, multiple prefetch, delayed branch and parallel execution of branches. At the time COBRA was first proposed [7], what was novel about it in relation to other mechanisms was the approach used to implement the parallel execution of branches, which is based on early computation of the target address and prefetching the two paths of branches. Besides, it was the first mechanism (as far as we know) that combined all these four types of techniques in order to reduce the branch cost to zero. After that, a few recent commercial processor such as the IBM RS/6000 [9], implement also a mechanism based on the combination of these four types of techniques. The same concept has different implementations that lead to different performance levels, so, the other contribution of COBRA is the way it is implemented. COBRA can be implemented using either a conventional instruction cache or a branch target instruction memory (both terms are defined later). We show in this paper that the implementation using the latter memory organization has a better performance in terms of cost-effectiveness.

To explain the functioning of COBRA, we distinguish two main units in the processor: the Instruction Unit (IU), which is responsible for fetching and sequencing instructions, and the Execution Unit (EU), which executes only data manipulation instructions (all instructions except control transfer instructions). The target address is computed in advance by the use of prefetching techniques. When the IU finds a branch (usually some cycles before it must take effect), it computes its target address and prefetches some of the first instructions of the two possible paths (multiple prefetch). When the

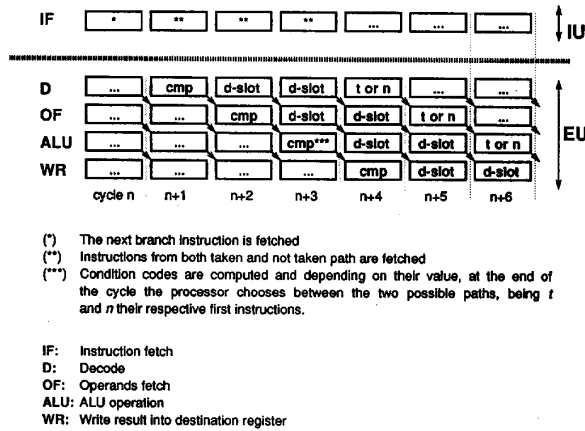


Fig. 1. Execution of a branch instruction using the COBRA mechanism.

result of the branch condition is known, one of the two prefetched flows of instructions is chosen. In this way, the delay introduced by branches is decreased by one unit (in general, it is decreased by the same amount of units as a fetch operation takes). The remaining delay slots are utilized by means of the delayed branch technique. All the operations required by branch instructions are performed by the IU in parallel with the EU activity, that is, with the execution of instructions different from branches. In this way, the time cost of many branches can be reduced to zero.

The scheme proposed by Katevenis in [13] is used to codify the target address of PC-relative branches. The basic idea of this approach is that the instruction contains the least-significant bits of the target address, rather than its offset. This scheme allows the IU to perform the prefetch in cache memory of the instructions at the target address in the cycle next to the fetch of the branch instruction, in parallel with the computation of the most-significant bits of the target address. In this way, the delay cycle cause by the addition operation in the conventional scheme is avoided.

Fig. 1 shows a possible execution of a branch using the COBRA mechanism for a sample pipeline. In this example the IU finds a branch in cycle n . After that, it continues fetching instructions that follow in sequence and also some instructions from the taken path. When the instruction that sets the condition codes finishes its ALU stage (cycle $n + 3$) the IU decides which path must be selected and sends the corresponding first instruction to the EU. From then on, the IU fetches instructions from the selected path until a new branch is found. The delay introduced by computing the condition codes (two cycles in this example) is used by means of the delayed branch technique [10]. If the ALU is the N th stage of the pipeline, with this scheme each branch will have $N - 2$ delay slots.

A. Memory Organization

Two different cache memory organizations have been considered for the implementation of COBRA. We call these organizations *conventional instruction cache memory* and *branch target instructions memory* (BTIM).

In a conventional instruction cache memory the mapping unit is a fixed size *block*. For a branch target instruction memory, the mapping unit consists of the instructions between two consecutive taken branches (including the latter branch). In this case, the mapping unit has a variable size and is defined at execution time. This unit will be called *sequence*.

To reduce the complexity that the management of information units with a variable size implies, a usual approach to implement a BTIM

consists in limiting to a fixed amount the number of instructions of a sequence that are stored in cache memory. If a sequence is greater than this size, the remaining instructions are obtained from the next level of the memory hierarchy. If it is smaller, the line is filled up with the instructions that follow in sequence. An implementation like this is used in the Am29000 processor [12].

Each entry of the cache memory will be called a *line*. A line stores a block in the case of a conventional cache or part of a sequence in the case of a BTIM.

To access the next level of memory, a *burst-mode protocol* is used. With this protocol, transactions are not fixed in length. After sending the instructions corresponding to a given line, the memory can continue sending the instructions of the following lines, one instruction per cycle, without any delay until the processor or memory decides to terminate the transaction. In this way, the latency of the external memory is experienced just once as long as the requested instructions are at consecutive addresses.

Each time a cache miss occurs, an entire new line is loaded into cache memory. The instructions of the line arrive at the rate of one per cycle, in the order they are stored in the line. As soon as the instruction that caused the miss is available, it is passed to the IU and begins execution. If a new cache memory access is required while a line is being loaded (for instance, when the line contains a taken branch), the former line must be completely loaded before beginning the new cache access.

B. Design of the Instruction Unit

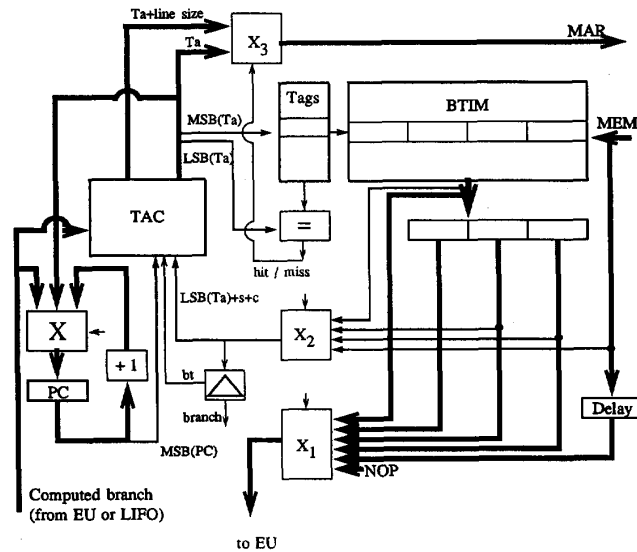
The main components of the instruction unit that implements the COBRA mechanism are shown in Figs. 2 and 3. The IU is composed of a BTIM and the hardware necessary for selecting the instruction that must feed the EU in each cycle, detecting branch instructions in advance and eliminating them from the flow of instructions sent to the EU. The implementation using a conventional instruction cache can be found in [8].

The IU uses the BTIM to prefetch the first line from the taken path of branches. Since the BTIM provides a complete line just in one cycle, the prefetch of the taken line can be postponed until the same cycle in which the condition codes for the branch are set. Accessing the BTIM earlier does not provide any additional benefit except for the case when the requested line is not in the BTIM. In this case, a further anticipation could be used to prefetch the line from external memory but, since the IU has just one path to external memory, this implies suspending the fetching of instructions that follow in sequence before the outcome of the branch is known. In [5] we demonstrated that this alternative does not provide any additional benefit.

In consequence, the IU must only analyze in each cycle the instruction that follows in sequence to the one that is in the first stage of the EU pipeline. If the analyzed instruction is a branch, the BTIM is accessed to obtain (if hit) the taken line. In the same cycle, the instruction that sets the condition codes will be in the ALU stage. In this way, at the end of this cycle, the BTIM line (or the corresponding miss) will be selected or discarded, depending on the condition codes.

The IU has a register to store the line obtained from the BTIM in case of hit. The first instruction of this line does not need to be stored because it must immediately be sent to the EU.

X_1 is a multiplexer that selects the instruction to be sent to the EU. The X_2 multiplexer selects the instruction next to the one selected by X_1 . This instruction is examined by the early branch detection circuit to check if it is a branch (in a RISC architecture it could be as simple as testing just one or very few bits of the op-code). The circuit that generates the control signals for these two multiplexers (not shown in Fig. 2) is basically a counter with the possibility of being incremented by one or two units depending on the result of



MSB: Most significant bits s: Sign bit. Used to compute the MSB of the target address
 LSB: Less significant bits c: Carry bit. Used to compute the MSB of the target address
 Ta: Target address bt: Indicates whether the branch is a computed branch or not.



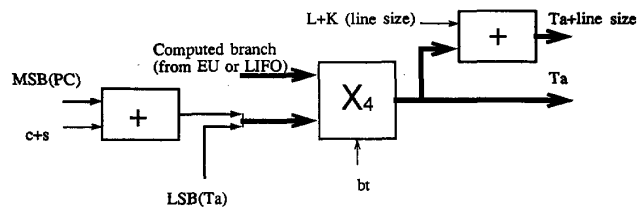
 TAC Target Address Computation circuit (see fig. 3).
 Branch detection circuit.

Fig. 2. Block diagram of the Instruction Unit.



MSB: Most significant bits s: Sign bit. Used to compute the MSB of the target address
 LSB: Less significant bits c: Carry bit. Used to compute the MSB of the target address
 Ta: Target address bt: Indicates whether the branch is a computed branch or not.

Fig. 3. Block diagram of the Target Address Computation circuit (TAC in Fig. 2).

the branch detection circuit. When a branch is taken, this counter is reset to zero.

The instructions supplied by the external memory should arrive at the IU one cycle before the EU can start its execution in order to be analyzed by the branch detection circuit and processed by the IU if they are really branches. A further anticipation, as explained above, does not provide any additional benefit. If for any reason, like a BTIM miss, they arrive later, some bubbles will occur in the EU pipeline, causing a degradation in the processor performance. During the cycle that an instruction supplied by the external memory is processed by the IU, it is held in the Delay register.

When a branch is detected, the BTIM is searched for the target line while the instruction that sets the condition codes is in the ALU stage. At the end of this cycle, the condition codes determine whether the

branch is to be taken. If the branch is taken, the PC block is loaded with the target address and X₃ selects the address that is sent to external memory. If the access to the BTIM produced a cache miss the selected address is the branch target address. Otherwise, it is the branch target address plus the cache line size ($S = L + K$). Note that the burst transaction initiated for the last taken branch is not yet suspended and, therefore, it can be continued if the branch is not taken.

The target address of computed branches is calculated by the EU and sent to the IU. Call and Return instructions are also a particular kind of branches. Call instructions can be sent to the execution unit, like an arithmetic instruction, with the sole objective of storing the return address (the targets address is computed by the IU). Return instructions are also sent to the EU and are treated like computed

TABLE I
NOTATION FOR THE MODELS

<u>From the applications:</u>		<u>From the implementation:</u>	
B:	Probability that an instruction is a branch	L:	Latency of external memory
T:	Probability that a branch is taken	S:	Size of BTIM lines
D(d):	Probability density function of the distance between two consecutive taken branches (length of sequences)	<u>From both the applications and implementation:</u>	
F(d):	Probability density function of the distance between two consecutive branch instructions.	H:	BTIM target hit ratio, which is computed as the number of taken branches whose target sequence is found in the BTIM divided by the total number of taken branches

branches that obtain the target address from the place where the corresponding call instruction stored it. The drawback of this solution is that Call and Return instructions, unlike the rest of branches, spend one cycle in the EU, and, therefore, cannot be completely executed in parallel. A more efficient solution, also more expensive, consists in adding a hardware stack to the IU, where the IU will store the return address of Call instructions in parallel with the EU activity. In this case, when the IU finds a Return instruction, the target address is obtained from the top of this stack, also completely in parallel with the EU activity. In this way, Call and Return instructions can be executed with zero time cost. The results presented in the next section assume that the IU has available this hardware stack.

IV. MODELING COBRA

A mathematical model of COBRA for the implementation that uses a BTIM is developed in this section. The model has some input parameters listed in Table I. These input parameters can be classified in three types: a) Those that depend on the applications ($B, T, D(d), F(d)$), b) those that depend on the implementation (L and S), and c) those that depend on both the applications and the implementations (H). This model will be used to compute the performance of the processor for different system configurations. In addition, an analytical model for the Delayed Branch scheme is presented. Its objective is to compare COBRA with Delayed Branch in order to show the extra performance of COBRA in relation to its hardware cost (shown in the previous section).

A. Pipeline

The efficiency of COBRA and Delayed Branch depend on the length of the pipeline. In this paper we concentrate on a pipeline in which the ALU stage is the second one. For this, pipeline, the Delayed Branch scheme has one delay slot per branch whereas COBRA does not need any delay slot and, in addition, branches are executed in parallel with other instructions. A deeper pipeline will imply an increase in the number of delay slots of both schemes.

B. Analytical Model for COBRA

The peak performance of the processor using COBRA is zero cycles for branches and one cycle for any other instruction. However, to achieve this peak performance several conditions must hold:

- The target line of each taken branch should be in the BTIM. The ratio of lines that are actually found in the BTIM depends on the number of lines of the BTIM, the BTIM organization, and the temporal locality of the program.
- Each cycle, the EU should begin the execution of a nonbranch instruction and, in parallel, the IU should deal with the instruction that follows in sequence. Even when every target line were in the BTIM, there would be no guarantee that this condition is met, since the IU relies on the external memory for part of those sequences whose size is greater than a BTIM line. So

the line size and the external memory latency also affect the performance of the processor.

In the development of the analytical model we assume that two branches never occur without at least one instruction between them. This hypothesis simplifies the model by introducing a negligible error, since in practice this fact happens very rarely.

The processor performance (P) is computed as the average number of useful instructions executed per cycle. Useful instructions are those instructions processed by the EU (all instructions but branches). In this way, $P = (1 - B)/(1 - B + D)$, where D is the average number of lost cycles per instruction (including branches). To compute D , the different sources of penalization will be characterized. Lost cycles are due to five different causes: 1) Memory latency due to BTIM misses, 2) Complete replacement of lines, 3) Memory latency for BTIM hits, 4) Lack of anticipation due to BTIM misses, and 5) Loss of anticipation due to not taken branches. Then, $D = D_1 + D_2 + D_3 + D_4 + D_5$, where D_i represents the average number of lost cycles per instruction due to cause i . Next, expressions for each D_i are developed.

1) *Memory Latency Due to BTIM Misses*: This happens when a branch is taken and a cache miss occurs when the IU accesses the BTIM to fetch the next sequence. The cost of this cache miss is L cycles. The probability that this event happens is $BT(1 - H)$, and, therefore, the average number of lost cycles per instruction due to this cause is $D_1 = LBT(1 - H)$.

2) *Complete Replacement of Lines*: This happens when the IU is dealing with a branch that turns out to be taken, a BTIM miss occurred in the previous taken branch and the distance between these two branches (here called d) is less than $S - 1$. In this case, the IU must finish the replacement of the former line before beginning to search the BTIM for the new line. The additional cycles needed to complete the replacement are $S - 1 - d$, and the average number of lost cycles per instruction due to this cause is

$$D_2 = BT(1 - H) \sum_{d=2}^{S-2} D(d)(S - 1 - d).$$

3) *Memory Latency for BTIM Hits*: This happens when the current sequence was found in the BTIM but it is larger than a line, and therefore, only the first instructions are in the BTIM; the remaining instructions are provided by the external memory. If the external memory latency (L) is greater than the line size (S), then $L - S$ cycles will be lost for each one of those sequences. The average number of lost cycles per instruction due to this cause is

$$D_3 = \begin{cases} BTH(L - S) \left(1 - \sum_{d=2}^S D(d)\right) & \text{if } L > S \\ 0 & \text{otherwise} \end{cases}$$

4) *Lack of Anticipation Due to a BTIM Miss*: This happens for any branch when a BTIM miss occurred in the previous taken branch. In this case, all the instructions between the last taken branch and the next taken one are provided by the external memory at the rate

of one per cycle and therefore branches cost one cycle since they are not detected early enough to overlap its execution with some previous instruction. In this way, while the IU is dealing with the branch a NOP is sent to the EU. The average number of lost cycles per instruction due to this cause is $D4 = B(1 - H)$.

5) *Loss of Anticipation Due to not Taken Branches*: This happens for sequences that are found in the BTIM and are larger than a line. Let assume that Y is the size of the sequence and it contains X branches. The number of cycles needed to read the complete sequence from memory is $Y - S + L$ and the number of useful instructions in the block is $Y - X$. Then, the number of cycles that the EU will be idle is $(Y - S + L) - (Y - X) = X - (S - L)$. When $S < L$, from this amount we must subtract the $L - S$ cycles that have already been taken into account in cause 3. In conclusion, we must count a lost cycle for each branch that is preceded by at least $S - L$ not taken branches, assuming that if $S - L < 0$ the previous sentence must be interpreted as preceded by at least zero not taken branches (this holds for any branch). The average number of lost cycles per instruction due to this cause is (see equation at bottom of page)

where N and I are random variables. N represents the number of not taken branches between the current branch and the previous taken branch and I represents the number of instructions of the sequence to which the branch being analyzed belongs.

Computing $\Pr(N \geq K)$: We assume that the probability that a branch is taken is independent of what happened in the branches executed before, which implies that the random variable N follows a geometric law. Note that in this case, the previous branches correspond to not taken branches and therefore all the previous branches and the one analyzed are different instructions. Then, it is reasonable to assume that each branch instruction is independent of the others, although this is not necessarily true. This introduces some negligible error in our analysis, but not enough to affect the result as the validation of the model (next section) will prove. Therefore,

$$\Pr(N \geq K) = \sum_{j=K}^{\infty} T(1 - T)^j = (1 - T)^K.$$

Computing $\text{Prob}(I > S | N \geq K)$: To compute this probability, we will first calculate $\Pr(I > S)$. To do that, we define B_Y as the average number of branch instructions in a sequence with Y instructions. We have that

$$\begin{aligned} \Pr(I = Y) &= \frac{B_Y D(Y)}{\sum_{j=2}^{\infty} B_j D(j)} \Rightarrow \text{Prob}(I > Y) \\ &= 1 - \frac{1}{\sum_{j=2}^{\infty} B_j D(j)} \sum_{j=2}^Y B_j D(j). \end{aligned}$$

B_Y can be computed using the expression

$$B_Y = \sum_{j=1}^Y j A_j C_j(Y)$$

where A_j is the probability that a sequence is composed of j branches and $C_j(Y)$ represents the probability that a sequence with j branches has a length equal to Y .

Because of the hypothesis made before, the value of A_j is given by the probability density function of a geometric law, which means

that

$$A_j = T(1 - T)^{j-1}.$$

$C_j(Y)$ depends on $F(d)$ and can be computed using the following expressions.

$$\begin{aligned} C_1(Y) &= F(Y) \\ C_j(Y) &= \sum_{k=j-1}^{Y-1} F(Y - k) C_{j-1}(k) \quad \text{if } j > 1. \end{aligned}$$

The evaluation of $\Pr(I > S | N \geq K)$ is similar to the calculation of $\Pr(I > S)$ with the difference that only those sequences with more than K branches must be considered, and the contribution of the first K branches must not be taken into account for computing this probability. Thus, we have that

$$\Pr(I > S | N \geq K) = 1 - \frac{1}{\sum_{k=2}^{\infty} M_K(k) D(k)} \sum_{k=2}^S M_K(k) D(k)$$

where $M_K(k)$ represents the average number of branches left (not including the first K branches) in a sequence with k instructions and assuming that the sequence has at least $K + 1$ branch instructions. Its value is equal to

$$M_K(k) = \sum_{j=K+1}^k (j - K) A_j C_j(k)$$

where A_j and $C_j(k)$ are the functions above defined.

6) *Validation of the Model*: The correctness of the analytical model was validated by comparing its results with the ones obtained by simulation of the execution of four benchmark programs: LEX, NROFF, PCC, and YACC¹ (9, 12, 21, and 42 million of executed instructions, respectively). These programs written in C language were compiled to RISC-II Assembly language [13] and their execution was simulated using the approach presented in [1]. From this simulation, in addition to the COBRA performance, the input parameters to the model (see Table I) were also obtained. The simulation was carried out for several values of the cache size, line size, and external memory latency. In this way, the processor performance was obtained for 31 sets of different values for these three parameters. The processor performance predicted by the model and the performance obtained by simulation was always less than 3.76% different and the average difference for the 31 simulations was 1.36%.

C. Analytical Model for Delayed Branch

For the memory organization that we call a BTIM, a line size equal to the external memory latency ($S = L$) is enough to obtain the maximum benefit from the delayed branch mechanism in terms of instruction execution rate. A further increase in the line size would reduce the external memory traffic but would not provide any additional gain in terms of execution rate since these extra instructions can be supplied by the external memory without any performance degradation. In consequence, the following model assumes that S is equal to L . The average number of lost cycles per instruction is the sum of the following four terms:

a) Execution of branch instructions: B

¹Unix utilities Unix is a trademark of AT&T Bell Labs.

$$D5 = \begin{cases} BH \Pr(N \geq (S - L) \cap I > S) = BH \Pr(I > S | N \geq (S - L)) \Pr(N \geq (S - L)) & \text{if } S \geq L \\ BH \Pr(N \geq 0 \cap I > S) = BH \Pr(I > S | N \geq 0) \Pr(N \geq 0) & \text{if } S < L \end{cases}$$

- b) No optimization of the delay slot: $B(1 - P_o)$. The value of P_o for each benchmark was obtained by the simulation of its execution.
- c) BTIM miss for a taken branch: $BT(1 - H)$
- d) A taken branch occurs before concluding the replacement of the line corresponding to the previous BTIM miss: $BT(1 - H)N_c$, where N_c is the average number of entries in the cache line that have not yet been filled. It can be calculated by the following expression:

$$N_c = \sum_{d=2}^{L-2} D(d)(L - 1 - d).$$

Then, the processor performance computed as the average number of useful instructions executed per cycle is equal to

$$P = \frac{1 - B}{1 + B(1 - P_o) + T(1 - H) + T(1 - H)N_c}.$$

The difference between the processor performance estimated by means of this model and the results obtained by simulation of the four benchmarks for 15 different sets of parameters was always less than 0.22%, and the mean value of the difference was 0.05%.

V. PERFORMANCE MEASURES

In this section, the efficiency of the COBRA mechanism is analyzed. First, we investigate which is the BTIM line size that maximizes the performance of COBRA. Next, the improvement achieved by COBRA in relation to the delayed branch mechanism is shown. Finally, the performance of COBRA with two alternative cache memory organizations are compared.

A. Size of the Cache Line

The first application of the mathematical model was to determine the optimum size of BTIM lines for COBRA mechanism. A typical value of the external memory latency (three cycles) was assumed for this analysis. In this section we show that, for the assumed external memory latency, the best tradeoff between cost and performance is provided by a cache line equal to four instructions.

The performance of the processor was obtained for a BTIM line size ranging from 1 to 6 instructions and a hit ratio ranging from 0 to 1 (note that the hit ratio, as it is defined in Table I, only depends on the number of lines, not on the line size). The other input parameters to the model (B , T , $F(d)$, and $D(d)$, see Table I), which depend on the applications, were assumed to be equal to the average of the values obtained for the four benchmarks. The results are shown in Fig. 4.

The main conclusion that can be drawn from Fig. 4 is that for a given hit ratio, the processor performance is improved when the line size augments, but only until a given size. A further increase in the line size produces a decrease in the processor performance due to the cost of loading a new line on cache misses. In this figure we can also see that the higher the hit ratio, the greater the size from which the performance begins to decrease. At the left end of the graphs (hit = 0) performance decreases as the line size increases whereas at the right end, performance augments as the line size gets larger.

When the line size is lower than the external memory latency (1 or 2 instructions) the performance of the system is rather low. If we compare line size of three with line size of four in Fig. 4, we can observe that the performance of the latter is better from low values of hit ratio on (hit \geq 0.4), and the difference between them is substantial for typical values of the target hit ratio (0.7–0.9). A further increment in the line size (5 instructions) is useful only if the hit ratio is greater than 0.7 and, in this case, the increase in performance is so low

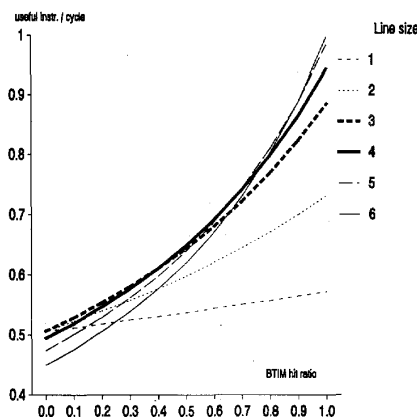


Fig. 4. Processor performance for different values of the BTIM hit ratio and line size, assuming an external memory latency of three cycles.

that it does not justify the additional occupied chip area. So, we can conclude that the best tradeoff between cost and efficiency is a line size of four instructions.

B. COBRA Versus Delayed Branch

In this section we show the benefits brought by COBRA. We have already seen the hardware cost needed to implement it. Here we compare the performance of COBRA against the delayed branch mechanism. Since this latter mechanism does not use any additional hardware, we can have an idea of the extra performance in relation to the additional hardware of COBRA.

Fig. 5 shows the performance of COBRA and delayed branch mechanisms. In both cases, the same cache memory organization has been assumed, that is, a BTIM with direct mapping and 32, 64, 128, or 256 lines. The line size is equal to the memory latency (3 instructions) for the delayed branch scheme and equal to the latency plus one unit (4 instructions) for the COBRA mechanism. The line size for COBRA is justified in the previous section whereas the choice for delayed branch, as explained in Section IV-C, is due to the fact that having a line greater than the external memory latency does not provide any additional increase in the instruction execution rate. In consequence, for a four instruction line size, the performance figures (useful instruction per cycle) of the delayed branch mechanism with a BTIM will be the same as the ones depicted in Fig. 5. The other input parameters to the analytical models (H , B , T , $F(d)$, $D(d)$, see Table I) were obtained from the simulation of the execution of each benchmark.

The efficiency of the COBRA mechanism is between 36% (BTIM with 32 lines) and 40% (BTIM with 256 lines) higher than the delayed branch for LEX; between 6 and 21% for NROFF; between 12 and 21% for PCC and between 24 and 26% for YACC. The higher the cache hit ratio, the greater the difference between them.

C. BTIM Versus Conventional Instruction Cache

It is also interesting to compare the efficiency of COBRA for different cache organizations. Fig. 6 shows the performance of the COBRA mechanism with a BTIM and with a conventional instruction cache. In both cases we assume the same number of cache lines, the same size of lines (4 instructions), a direct mapping and a three-cycle external memory latency. The performance figures for a conventional cache were obtained using the approach presented in [8].

Fig. 6 shows that, for the cache parameters evaluated, a conventional instruction cache and a BTIM have a similar performance for

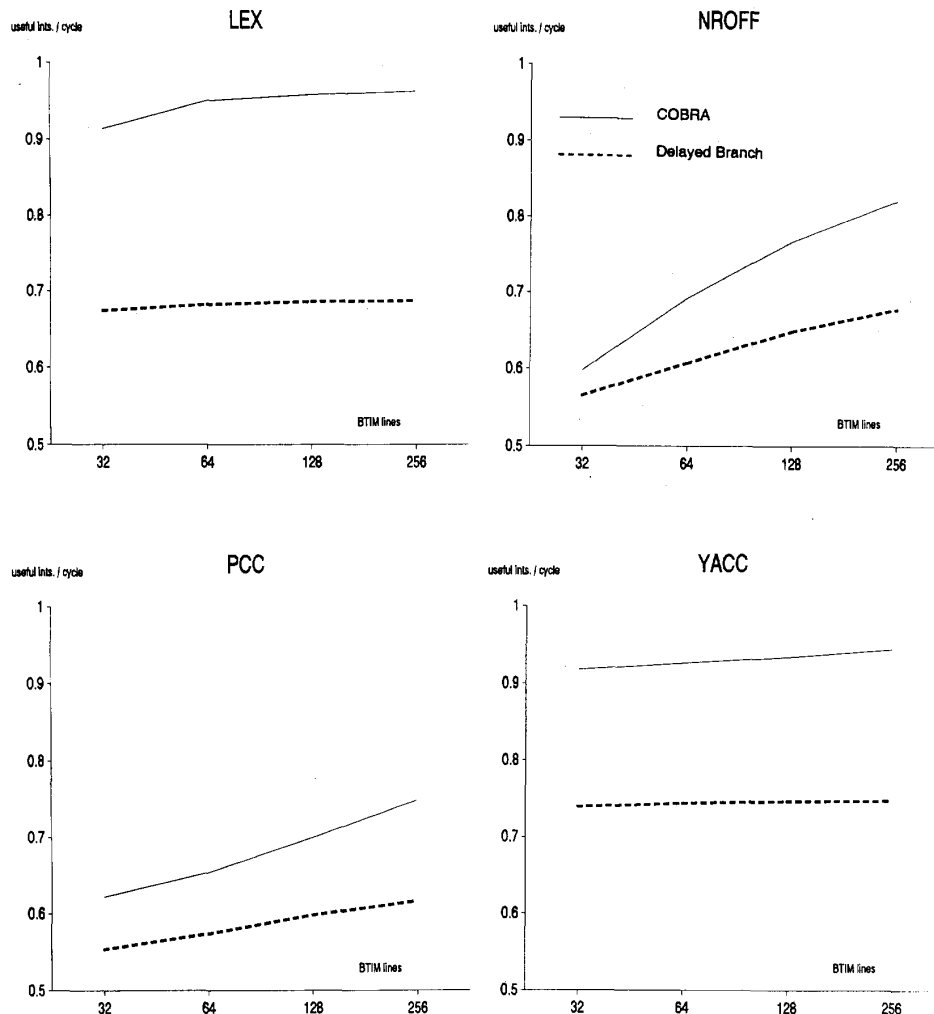


Fig. 5. COBRA versus delayed branch.

LEX and YACC (a little better for a conventional cache) whereas for NROFF and PCC, the performance of a BTIM is considerably better than a conventional cache. The improvement of the BTIM in relation to the conventional cache ranges from -1 to -3% for LEX, 29 to 2% for NROFF, 18 to 12% for PCC, and 4 to -5% for YACC. The main difference between LEX, YACC and PCC, NROFF is that the former two programs exhibit a higher temporal locality. We can also observe in Fig. 6 that the improvement of a BTIM in relation to a conventional cache increases as the number of lines (and therefore the hit ratio) increases. So the conclusion just regarding efficiency is that both schemes provide about the same efficiency when the cache hit ratio is very close to 1 and the performance of the BTIM is considerably better when the hit ratio is not so high.

On the other hand, the BTIM generates much more traffic than a conventional cache. For LEX the BTIM traffic is between 424 and 5220% higher than the conventional cache traffic; 28 – 234% for NROFF; 46 – 104% for PCC; 422 – 5956% for YACC. The reason is that, in a BTIM, there are many instructions that must always be supplied by the external memory, regardless of the number of lines of the cache and the cache hit ratio. These instructions are due to sequences greater than a cache line. In this case, the BTIM only

stores the first instructions of the sequence (just a line) and the rest of instructions are supplied by external memory even when a BTIM hit occurs for that sequence. Note that this extra traffic does not mean any penalization in the processor speed since the access to external memory is overlapped with the execution of instructions provided by the BTIM.

Finally, regarding hardware cost, the implementation of the IU requires a simpler hardware for a BTIM. The design of the IU for a conventional cache can be found in [8]. In conclusion, a BTIM offers a better cost-efficiency performance than a conventional cache since the former simplifies the implementation of the IU and in addition it provides in many cases an efficiency quite higher than a conventional cache.

VI. CONCLUSIONS

We have presented and evaluated a mechanism (COBRA) for reducing the cost of branches in pipelined processors. The mechanism is based on the following techniques: a) early computation of the target address, b) multiple prefetch, c) delayed branch, and d) parallel execution of branch instructions.

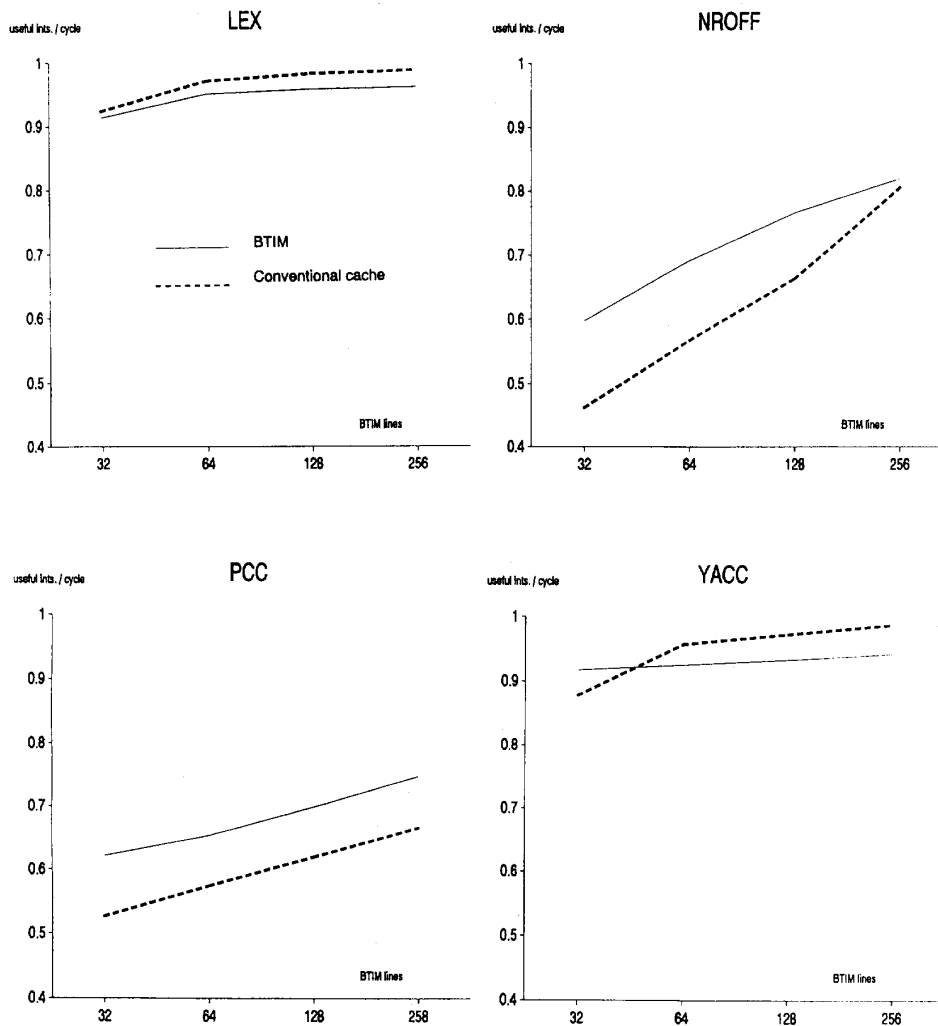


Fig. 6. COBRA with a BTIM versus COBRA with a conventional instruction cache.

An implementation of the mechanism using a Branch Target Instruction Memory (BTIM) is proposed. The behavior of the system has been characterized by means of an analytical model. This model has been used to select the most adequate size of BTIM lines which, for a external memory with latency equal to three cycles, resulted to be equal to the latency plus one unit.

The efficiency of the COBRA mechanism is in average about 25% higher than the Delayed Branch and the additional hardware needed to implement COBRA is quite simple. We have also compared two implementations of the COBRA mechanism, each one using a different cache organization. The conclusion was that, in terms of cost-effectiveness, the BTIM has a better performance than a conventional instruction cache although the former generates a higher memory traffic. This extra traffic does not mean any penalization in the processor speed since it is overlapped with the execution of instructions provided by the BTIM.

ACKNOWLEDGMENT

We would like to thank T. Lang and the anonymous referees for many suggestions that improved the quality of this paper.

REFERENCES

- [1] J. Cortadella and J. M. Llaberia, "Low cost evaluation methodology for new architectures," in *Proc. IASTED Int. Symp. Appl. Informatics*, Feb. 1987, pp. 192-195.
- [2] J. H. Crawford, "The i486 CPU: Executing instruction in one clock cycle," *IEEE Micro*, vol. 10, no. 1, pp. 27-36, Feb. 1990.
- [3] R. W. Edenfield, "The 68040 Processor. Part 1, Design and implementation," *IEEE Micro*, vol. 10, no. 1, pp. 66-78, Feb. 1990.
- [4] R. B. Garner *et al.*, "The scalable processor architecture (SPARC)," in *Proc. 33rd. IEEE Int. Comput. Soc. Conf.*, COMPCON'88, Feb 1988, pp. 278-283.
- [5] A. González, "Designing an instruction cache for reducing the cost of branches," Rese. Rep. UPC/DAC RR-91/02, Comput. Architecture Dep., Polytechnic Univ. of Catalonia, Barcelona, Jan. 1991.
- [6] A. González and J. M. Llaberia, "Instruction fetch unit for parallel execution of branch instructions," in *Proc. 3rd. Int. Conf. Supercomput.*, ACM SIGARCH ICS-89, June 1989, pp. 417-426.
- [7] A. González, J. M. Llaberia, and J. Cortadella, "Zero-delay cost branches in RISC architectures," in *Proc. IASTED Int. Symp. Appl. Informatics*, Feb. 1988, pp. 24-27.
- [8] —, "A mechanism for reducing the cost of branches in RISC architectures," *Microprocessing and Microprogramming*, vol. 24, no. 1-5, pp. 565-572, Aug. 1988.

- [9] G. F. Grohoski, "Machine organization of the IBM RISC System/6000 Processor," *IBM J. Res. Develop.*, vol. 34, no. 1, pp. 37–58, Jan. 1990.
- [10] T. R. Gross and J. L. Hennessy, "Optimizing delayed branches," in *Proc. 15th Annu. Workshop Microprogramming, ACM SIGMICRO*, Oct. 1982, pp. 114–120.
- [11] G. Hinton, "80960 — Next generation," in *Proc 34th. IEEE Comput. Society Conf. COMPCON'89*, Feb. 1989, pp. 13–17.
- [12] M. Johnson, "System considerations in the design of the Am29000," *IEEE Micro*, vol. 7, no. 4, pp. 29–41, Aug. 1987.
- [13] M. G. H. Katevenis, *Reduced Instruction Set Computer Architecture for VLSI*. Cambridge, MA, MIT Press, 1985.
- [14] D. L. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Comput. Mag.*, vol. 21, no. 7, pp. 47–55, July 1988.
- [15] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proc. 13th Int. Symp. Comput. Architecture*, 1986, pp. 396–403.
- [16] T. Riordan *et al.*, "System design using the MIPS R3000/3010 RISC Chipset," in *Proc. 34th IEEE Comput. Soc. Conf., COMPCON'89*, Feb. 1989, pp. 494–498.

Constant Geometry Fast Fourier Transforms on Array Processors

George Miel

Abstract—Matrix algebra is used to design and validate parallel algorithms for large constant geometry FFT's on fixed-size array processors. The N -point radix 2 case for a linear array processor with $N/2$ cells is identical to the usual procedure corresponding to the matrix factorization of M. C. Pease. The algorithms are engendered by matrix factorizations, which themselves depend on a basic factorization of the perfect shuffle. The resulting data movement is realized in parallel as relatively small perfect shuffles inside each local memory and along each row and column of the array processor, without requiring that the complete array itself have the shuffle-exchange network.

Index Terms—Array processing, fast Fourier transforms, parallel algorithms.

I. INTRODUCTION

The matrix approach, as a means to design and validate algorithms for parallel architectures, was used and advocated by Pease [9] in his modification of the Cooley–Tukey procedure. The resulting algorithm is often called a constant geometry FFT because its communication pattern, namely, the addressing of operands for the butterfly operations, is kept the same from stage to stage. For the N -point radix 2 case, the algorithm consists of $\log_2 N$ stages each preceded by a perfect shuffle of the data. The most natural mapping of this algorithm is onto a linear array architecture with $N/2$ cells and a shuffle-exchange interconnection network [2], [14], [15]. Thompson [16] has shown that the VLSI design of this architecture achieves area*time² performance of $\Omega(N^2 \log_2^2 N)$, which is the optimum theoretical limit for the N -element Fourier transform established by Vuillemin [17].

The matrix factorization of the Fourier transform given by Pease is invaluable in the study of parallel FFT's. The problem of parallelizing

Manuscript received June 15, 1990; revised March 15, 1992. This work was done at and supported by Hughes Research Laboratories, Malibu, CA 90265. The author is with the Department of Mathematical Sciences, University of Nevada, Las Vegas, NV 89154.

IEEE Log Number 9202844.

an FFT is essentially that of scheduling onto a targeted architecture the tasks engendered by the matrix factors in the corresponding factorization. This approach was used by Norton and Silberger [8] in the parallelization and performance prediction of FFT algorithms for MIMD shared-memory architectures. Recently, Whelchel and others [18] used the Pease factorization to describe a pipeline architecture, based on matrix factors called systolic phase rotations, which eliminates delay commutator switches used in the Purdy McClellan processor.

Our aim is to decompose the Pease factorization in order to map large constant geometry FFT's onto fixed-size rectangular array processors. Section II shows that our results depend fundamentally on a factorization of the perfect shuffle permutation. The resulting data movement is realized in parallel as relatively small perfect shuffles inside each local memory and along each row and column of the array processor, without requiring that the complete array itself have the shuffle-exchange interconnection network. Section III uses these results to validate parallel algorithms for rectangular array processors.

The effectiveness of a mapping of a constant geometry FFT onto an array processor depends primarily on two items. The first item is the efficiency with which the interconnection network of the array processor realizes the data movement required by the algorithm. The second item involves a divide-and-conquer strategy for the SIMD evaluation of specialized matrix-vector products. Suppose that a product Dz , where D is the direct sum

$$D = \bigoplus_{i=0}^{N-1} A_i \quad (1)$$

with each A_i of dimension $M \times M$ and z is an MN -vector, is to be computed on an array processor with N cells. The vector is first divided into N M -tuples

$$z = (\bar{z}_0, \bar{z}_1, \dots, \bar{z}_{N-1})^t, \quad \bar{z}_i = (z_{iM}, \dots, z_{(i+1)M-1}),$$

each cell computes in parallel a product $A_i \bar{z}_i^t$, and the subvectors are then concatenated to get the result. Whereas the first item deals with the communication complexity of the mapping, the second item pertains to its parallel arithmetic complexity.

II. MATRIX FACTORIZATIONS

A *perfect shuffle* is a permutation that transforms the $2m$ -vector

$$z = (0, 1, \dots, m-1, m, m+1, \dots, 2m-1)^t$$

to the vector

$$S_{2m} z = (0, m, 1, m+1, \dots, i, m+i, \dots, m-1, 2m-1)^t. \quad (2)$$

Components that were m apart become adjacent as a result of the perfect shuffle. For simplicity, we henceforth call (2) the *shuffle* of z .

Permutations by cutting and shuffling were studied by Golomb [3]. Computational applications of the shuffle were conceived by Batcher [1] for bitonic sorting and by Singleton [13] and Pease [9] for the fast Fourier transform. In particular, Pease presented a matrix factorization of the transform, (4)–(5) below, suitable for parallel implementation. The relevance of the shuffle permutation in parallel processing was further established by Stone [14]. The shuffle-exchange interconnection network in a multiprocessor system provides useful capabilities [2]. For instance, Wu and Feng [19] have shown that a shuffle-exchange network of size N can realize an arbitrary permutation in $3 \log_2 N - 1$ passes.