Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero

David R. Ditzel
Hubert R. McLellan

AT&T Bell Laboratories
Murray Hill, N.J. 07974

Abstract

A new method of implementing branch instructions is presented. This technique has been implemented in the CRISP Microprocessor. With a combination of hardware and software techniques the execution time cost for many branches can be effectively reduced to zero. Branches are *folded* into other instructions, making their execution as separate instructions unnecessary. Branch Folding can reduce the apparent number of instructions needed to execute a program by the number of branches in that program, as well as reducing or eliminating pipeline breakage. Statistics are presented demonstrating the effectiveness of Branch Folding and associated techniques used in the CRISP Microprocessor.

Introduction

The efficient implementation of branches is of major concern in the implementation of high performance pipelined computers. This paper discusses branch problems and attempted solutions in various machines. The implementation of branches in the CRISP Microprocessor is then described. The two primary results are shown. First, pipeline breakage is reduced. Second, the total number of instructions executed by the execution pipeline is reduced by the number of (folded) branches. Since branch instructions account for a large fraction of all computer instructions executed, this reduction offers a correspondingly large execution time speedup.

The CRISP Microprocessor

The CRISP Microprocessor is a high performance single chip general purpose microprocessor.[1] The CMOS chip contains 172,163 transistors and is capable of achieving a peak execution rate of greater than 16 MIPS with a clock frequency of 16 Mhz. Part of CRISP's performance comes from a high degree of pipelining.[2] Early in the design we realized that the full potential of pipelining could be achieved only if some method was found to eliminate traditional problems with branches. Our solution involves the synergistic combination of three techniques. First, from new hardware structures to implement Branch Folding. Second, from software techniques, in particular the application of compiler technology. Third, from designing the instruction set to match a high performance implementation and the available compiler technology.

The Branch Problem and Some Solutions

The implementation of branch instructions is one of the hardest and most important problems to be dealt with in the implementation of high performance pipelined computers. Branch instructions tend to interrupt the smooth flow of instructions through an instruction pipeline making the average instruction throughput rate much lower than the peak rate. For example, early studies for the pipelined MU5 computer showed that if branches occurred in only one out of ten instructions then performance would be reduced by a factor of three, unless special precautions were taken.[3] Branches occur very frequently in programs; various studies show the dynamic frequency of branches can be as much as one third of all instructions executed.[4,5,6]

The peak rate of instruction flow in a pipelined machine can only be obtained if the pipeline can be kept full. Discontinuities can occur because a branch may be several stages deep in the pipeline before it takes effect. In this case, all instructions introduced to the pipeline after the branch would have to be flushed, causing useless empty pipeline bubbles to occur. Conditional branches have two possible paths to follow. Unless instructions can be cancelled or backed up without side effects, it would be necessary to stall the pipeline until the correct outcome of the branch was known. Finally, a branch can interfere with program prefetching strategies, causing waits due to main memory latency.

The importance of dealing with the performance degradation due to branch instructions has been recognized for a long time. A description of the design of the STRETCH (IBM 7030) computer relates how branches can spoil the flow of instructions to the instruction unit, and how a computer can use branch prediction to improve performance.[7] Implementing branch instructions so that a branch transfer does not take effect until instructions after the branch are also executed can be used to reduce branch delay; this technique is commonly referred to as "delayed branch." The technique of delayed branch was used as early as 1952 in the Los Alamos MANIAC computer. Delayed branch has seen a recent resurgence in popularity with machines such as the IBM 801,[8] the Berkeley RISC I,[9] Stanford MIPS,[10] the MIPS Inc. R2000,[11] and HP Spectrum[12] computers. McFarling and Hennessy have reviewed the costs associated with variations on the delayed branch

scheme.[13]

The Manchester MU5 computer system uses a *Jump Trace* that caches the address of instructions which have caused a branch. A cache hit causes pre-fetching to continue at the *jumped-to* address stored in the jump trace cache, rather than with the next sequential instruction.[14,15] Lee and Smith describe this technique as a Branch Target Buffer, and show improvements over the MU5 approach.

A number of hardware strategies specifically to deal with branches have been employed in recent machines. The IBM 360 series model 91[16] duplicates some logic to start prefetching the unpredicted branch target as well as the predicted path. A more extreme approach is taken by the IBM 3033[17] by simultaneously following both possible paths of conditional branch instructions. Other proposed schemes include using up to 24 separate condition code registers to allow the compiler to precompute tests,[18] and using high level language control semantics embedded in the object code.[19]

Lee and Smith[20] provide a more general survey of previously used approaches, and in particular results on the efficacy of the Branch Target Buffer approach.

Branches in the CRISP Instruction Set

Branches may be either conditional or unconditional. Conditional branches are conditioned on the value of a single flag bit, kept in the Program Status Word register. Two instruction forms of conditional branch are provided; one that branches if the flag bit is true, and another that branches if the flag bit is false. Conditional branches also contain a single static branch prediction bit, which may be set by the compiler. This bit is used as a hint to the hardware as to whether the branch will transfer or not.

CRISP instructions are encoded in three different lengths, composed of either one, three or five 16-bit instruction parcels. Two instruction lengths are provided to encode conditional and unconditional branches. A one parcel branch instruction contains a 10-bit PC relative offset. Since CRISP instructions are aligned on 16-bit boundaries, this allows for a range of -1024 to +1022 bytes. A three parcel branch instruction contains a 32-bit branch specifier field. This field is typically used to hold an absolute address, but may also specify a branch indirect through an absolute address, or a branch indirect though the address specified by a 32-bit offset from the Stack Pointer. Dynamic instruction measurements show that around 95% of the branches executed are encoded in the one parcel instruction format. Most of the remainder use the three parcel form with an absolute address. Indirect branches are only occasionally generated by our compiler for such constructs as case statements.

The condition code flag can only be modified as the result of a compare instruction. A compare instruction can compare two operands located in memory via four standard addressing modes. Details on these addressing modes and the rest of the instruction-set are not further relevant to the discussion on branches, and may be found elsewhere.[21]

Three specific choices were made in the design of the instruction set to match high performance implementations. First, we chose to have separate compare and conditional branch instructions, rather than an integrated compare and branch. The motivation was that the outcome of a branch needs to be determined at the head of the pipeline, so that instructions following the branch may be entered into the pipeline as soon as possible. Unfortunately, the resolution of a compare is generally only known at the tail of the pipeline. It is not possible for a single compare-and-branch instruction to be both places at once. (Some optimizations are possible, e.g. Katevenis' Fast Compare and Branch[22] scheme.) Second, the compare instruction is the only instruction that can modify the condition code flag. This reduces the number of instructions in the pipeline that can affect the outcome of a conditional branch, making code motion and branch prediction techniques more effective. Avoiding the modification of the condition code by non-comparison instructions has other benefits as well.[23,24] Third, the instruction set was designed to avoid side effects (such as auto-increment), so that instructions could be easily cancelled before the result write of the last pipeline stage. This allows mis-predicted branches and instructions following them in the pipeline to be easily cancelled.

Branch Prediction

Up to 3 cycles may be lost if the initial path of a conditional branch is incorrectly chosen in the Execution Unit. It is therefore important to chose the correct path with high probability. Because basic block sizes in CRISP are typically short, on the order of 3 instructions, we decided that branch prediction would be a better technique than delayed branch. Delayed branch might be more effective for load/store machines where the basic blocks are somewhat larger, and offer more opportunities for scheduling branch delay slots. For our branch prediction technique we wanted to make the best compromise between performance and simplicity of implementation. A variety of branch prediction techniques have been suggested by J. Smith[25] and Lee and Smith.[26]

The decision to use a single static branch prediction bit in CRISP was made after comparing the effectiveness of this technique against other proposed schemes. Static prediction is clearly the simplest approach. The more complex schemes involve keeping a dynamic history in an instruction cache or branch target buffer. Because our instruction set and workload are somewhat different than those described in the literature, we made additional tests to judge the effectiveness of the various schemes. Rather than the traditional evaluation method of using trace tapes, we modified a VAX C compiler to generate additional code which would simultaneously apply several different branch prediction techniques as the program ran. This allowed us to measure many long running programs with ease. The branch prediction strategies measured were static prediction, and one, two and three bits of dynamic prediction. The static prediction numbers report accuracy for optimal setting of a branch prediction bit in the branch instruction. One bit of dynamic history predicts to branch the same as the last time. The two and three bit dynamic history algorithms provide weighting, as descibed by J. Smith. The dynamic history assumes an infinite size table, this makes the dynamic numbers somewhat optimistic. In practice only a small number of recent predictions would be cached.

Table 1 shows the prediction accuracy for several benchmarks. The benchmarks include three large programs and three

common benchmark programs. The large programs are the troff text processor, the C compiler, and a VLSI design rule checker. The results show that the dynamic techniques are not significantly better than what is possible with static prediction.

On the commonly used benchmarks (Dhrystone, Cwhet and Puzzle), static prediction was actually superior to the more complex dynamic schemes. This result is somewhat unexpected and is due to the conditional branches either branching one direction all the time, or alternating. For the case of branching in one direction, all schemes get essentially 100% correct prediction. For the case where branches alternate direction, static prediction gets 50% correct, while all the dynamic schemes get 0% correct.

Given the increased complexity of the dynamic strategies, the use of a single static prediction bit in CRISP seems to be a reasonable choice. The setting of CRISP's branch prediction bit is normally be done by the compiler, though other techniques are possible. The particular heuristics used by the compiler are discussed elsewhere.[27]

Program	static branch prediction	1 bit of dynamic prediction	2 bits of dynamic prediction	3 bits of dynamic prediction	Number of branches executed
Troff	.94	.93	.95	.95	22 Million
C compiler	.74	.77	.77	.74	1.5 Million
VLSI DRC	.89	.95	.95	.95	38 Million
Dhrystone	.86	.72	.79	.79	1.5 Million
Cwhet	.84	.68	.79	.79	33,550
Puzzle	.92	.87	.87	.87	741

Table 1. Accuracies of branch prediction techniques.

Branch Spreading

Branch prediction is useful when a conditional branch instruction enters the pipeline before the result of a preceding compare can be computed. If, however, there are no compare instructions in the pipeline then there is no need for branch prediction because the outcome of the conditional branch is known with certainty. In such a situation, the correct next instruction address can be determined for the conditional branch as soon as it enters the execution pipeline.

Because CRISP has separate compare and conditional branch instructions it is possible to have the compiler assure that no comparison instructions will be in the pipeline when a conditional branch is read from the instruction cache. The simplest approach is to have the compiler generate a sufficient number of no-op instructions between a compare instruction and the corresponding conditional branch instruction. Use of code motion can do much better by moving useful non-condition code setting instructions between the compare instruction and the conditional branch instruction.

This form of code motion is similar to that used with delayed branch instructions, and has been found to be very effective.[28] The main difference is that CRISP does not require delay slots as part of the definition of the instruction-set. The use of delayed branch instruction still costs the delay of a full instruction to simply move the branch through the pipeline. This branch slot is not required in CRISP when Branch Folding is used.

Branch Folding

A simplified block diagram of the CRISP architecture is shown in Figure 1. Instructions are fetched from main memory by a three stage pipelined Prefetch and Decode Unit (PDU), decoded into a more easily executed internal form, and placed into a Decoded Instruction Cache. A three stage pipelined Execution Unit (EU) reads the instructions from the cache and executes them. Placing the Decoded Instruction Cache in the middle of what would otherwise be a six stage pipeline is to key a number of performance and implementation details. First, the cache decouples the PDU from the EU, allowing each to operate independently. If the PDU has to wait for memory, this does not necessarily stall the EU. Second, pipeline breakage problems have been reduced by cutting the length of the pipeline in half. Third, executing decoded instructions is considerably easier than encoded instructions. Encoded instructions may be as short as 16-bits, the decoded instruction cache makes all instructions appear in a canonical fixed 192-bit length, similar to a horizontal microinstruction.

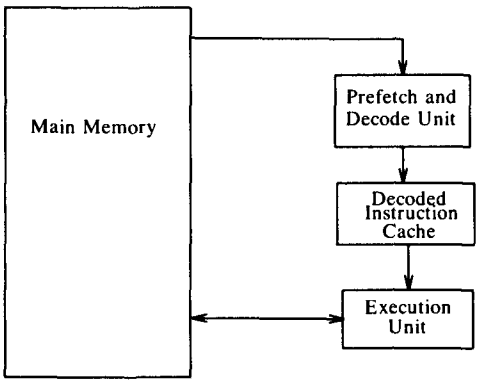

Figure 1. Simplified block diagram of CRISP Microprocessor.

An instruction may be executed several times from the instruction cache and each time (except for conditional branches) the next address is always the same. Instead of recalculating this address every time the instruction is executed, the next address logic is moved to the input side of the instruction cache and a 31-bit (parcel aligned) "next-address" field is added to the cache. As instructions are placed in the cache, their next-address value (Next-PC) is stored with them. When the EU reads an instruction from cache, the next address value is immediately available to address the next instruction from the cache.

This machine organization is similar to many speed optimized microprogrammed machines where each microinstruction has its own next address field. What we have done differently is to achieve the same benefits for computer macro instructions by dynamically generating the contents of the next address field rather than storing it with each instruction in main memory.

Providing a next address field for every instruction in the cache has the same effect as turning every instruction into a branch instruction. But since every instruction in the cache can perform a branch, there is no need for separate branch instructions. During decoding the CRISP PDU recognizes when a non-branching instruction is followed by a branch instruction and "folds" the two instructions together. This single

instruction is then placed into the Decoded Instruction Cache. The separate branch instruction disappears entirely from the Execution Unit pipeline and the program executes as if the branch were executed in zero time. We refer to this technique as *Branch Folding*.

For the conditional branch instruction the next instruction address is not invariant after an instruction is read from the instruction cache. Conditional branch instructions will either execute the next sequential instruction or will branch to another instruction, depending on the value of the condition code flag. If conditional branches are also to be folded, then some additional mechanism is needed. We therefore added a second 31-bit address field to the instruction cache called the Alternate Next-PC. This field is used to hold the second possible next instruction address of a conditional branch instruction. When a folded conditional branch instruction is read from the instruction cache one of the two paths for the branch is selected for the next instruction address, and the address that was not used is retained with each instruction as it proceeds down the execution pipeline. The Alternate Next-PC field is to be retained with each pipeline stage until the logic can determine whether the selected branch path was correct or not. When the outcome of the branch condition is known, if the wrong next address was selected, any instructions in progress in the pipeline following the conditional branch are flushed and the Alternate Next-PC from the folded conditional branch is re-introduced as the next instruction address at the beginning of the instruction pipeline.

Implementation of Branch Folding

Figure 2 shows a schematic of the essential datapaths used in the CRISP microprocessor for the implementation of Branch Folding. The implementation is shown to illustrate the amount of extra logic needed to implement branch folding, and to show the actual implementation and design decisions used in CRISP. Not shown in the schematic are the paths in the PDU used to fetch instructions from main memory, as this is independent of Branch Folding.

CRISP does not try to fold all branch instructions, only those that occur with the greatest frequency. CRISP's policy is to only fold one and three parcel non-branching instructions with one parcel branches. Doing the remaining cases significantly increases the amount of hardware required, with only a marginal increase in performance.

In the PDU, the instruction to be decoded comes from the output of an Instruction Queue in the Prefetch Decode Register (PDR) pipeline stage. The address of the first parcel in the Queue (QA) is held in the PDR.PC pipeline register. The output of five instruction parcels, QA-QE, are available simultaneously for decoding. Decoded instructions are clocked into the Prefetch Instruction Register (PIR) stage one clock cycle after they enter the PDR stage. The Next-PC field (or Alternate Next-PC field) can have one of three sources. For a sequential instruction, or the sequential part of a conditional branch, the next address is found by adding the instruction length (ilen<0:2>) to instruction address (PDR.PC). Second, if the branch uses a 32-bit address, this address is selected directly from the QB and QC parcels. Third, the branch target address may come from a one parcel branch, either folded or not.

For folded instructions the branch is always the one parcel format, and must follow a one or three parcel instruction. It follows then that the 10-bit PC relative offset is found in the QB parcel if the previous instruction was one parcel, or in the QD parcel if the previous instruction was three parcels long. This 10-bit offset is selected by the *tpcmx* multiplexor and then added to a 2-bit branch adjust. The branch adjust is necessary because the PC relative offset is relative to the address of the branch, not the instruction it is being folded with. The value of the branch adjust is simply the size of the instruction starting in the QA parcel. The adjusted offset is then added to the address of the instruction being folded (in the PDR.PC register) to yield the branch target address. For cases where a one parcel branch is not folded, (for example, a branch after a call), the offset is selected from the QA parcel, and the branch adjust field is zero.

If the branch prediction bit is set to indicate that the branch will likely be taken, then the Next-PC of the PIR stage will the appropriate branch target address, and the Alternate-PC will contain the address of the next sequential instruction. If the branch prediction bit is set to indicate that the branch will not likely be taken, then the Next-PC will contain the sequential instruction address and the Next-PC will contain the branch target address.

In the Execution Unit, there are three pipeline stages. Instructions are read from the Decoded Instruction Cache into the Instruction Register (IR) stage, operands are accessed and placed into the Operand Register (OR) stage, then an ALU operation takes place and the ALU result placed in the Result Register (RR) stage, and finally the result write occurs. The flow of instructions in the EU is controlled by the IR stage Next-PC register. This register specifies the next instruction to be loaded into the IR stage, the low five bits are used to address the Decoded Instruction Cache. When the CPU is initially reset, the IR.Next-PC is set to zero, this is where the first instruction fetch will start.

Under the normal condition of loading a new instruction into the IR, the IR.Next-PC is loaded directly from the Next-PC field of the instruction coming from the Decoded Instruction Cache. For this case, the alternate address for a conditional branch is loaded into the IR stage Alternate-PC registers. The PC of each instruction is carried with each each pipeline stage to identify the instruction in the case of an interrupt or other exception.

For the case of indirect jumps, the IR.Next-PC may be loaded from the Stack Cache, or from off-chip via the data_in bus.

Conditional branches may or may not be predicted correctly via the single static branch prediction bit. The direction not predicted is kept with each pipeline stage in the Alternate-PC field. CRISP tries to minimize the penalty for in incorrectly predicted branch by recovering as soon as the correct instruction path is known.

If the conditional branch has been folded with a compare in the same instruction, the true outcome of the branch is not resolved until the instruction reached the RR stage. If the branch is determined to be incorrectly predicted when the branch reaches the RR stage, the RR.Alternate-PC will be placed in the IR.Next-PC at the next clock, and the instructions

being clocked into the IR, OR and RR will have their valid bit set to zero. Because side-effects were avoided in the instruction set, any instruction may be turned into a no-op by resetting the valid bit associated with each stage. Three clock ticks will be lost in this case.

If the compare is one stage ahead of the conditional branch, the correct address is obtained from the OR.Alternate-PC and only two clock ticks are lost. If the compare is two stages ahead of the conditional branch then the correct address is obtained from the IR.Alternate-PC and only one clock tick is lost. If however, the compare is three stages ahead of the conditional branch, the compare will have left the pipeline before the conditional branch is place in the IR. For this case, the condition code will not change, and the branch need not be predicted. If the current condition code is not the same as that predicted for the conditional branch about to be loaded into the IR, then the IR.Next-PC is taken from the Alternate-PC field of the Decoded Instruction Cache and zero cycles can be lost due to incorrectly setting the branch prediction bit. The conditional branch has effectively been turned into an unconditional branch. Branch Spreading tries to take advantage of this last case.

Evaluation

Branch Folding, Branch Prediction, and Branch Spreading are techniques used in the CRISP Microprocessor to reduce the execution time incurred for branches. A simple example will illustrate their use, and quantitatively show their effect in our implementation. The short C program shown in Figure 3 was picked to isolate the performance effects related to branches, rather than other CRISP features. The loop count of 1024 is high enough to overcome about 50 cycles of initial overhead in calling the main routine without significantly affecting statistics. This allows our tools to measure the entire program rather than having to measure isolated instructions. The results are relatively independent of the actual loop count. The CRISP code generated for the loop is straightforward, and contains two conditional branches, one for the **if** and one at the end of the code to branch to the top of the loop. An unconditional branch is used to branch around the **else** clause. The **if** statement in the program was chosen to be difficult for branch prediction, as the expression will alternate between evaluating true and false. The conditional branch at the end of the loop would be predicted by a compiler to branch back to the top of the loop. No special optimizations other than those described were used.

To show the number of branches, and for comparison purposes, the program of Figure 3 was analyzed for both CRISP and the VAX. The code for both machines was generated directly from our standard compilers. The instruction distribution is shown in Table 2. The result in terms of number of instructions executed was essentially identical.

This same program was run in four different ways, selectively enabling the use of Branch Folding, Branch Prediction and Branch Spreading. Branch prediction *yes* means that the end of loop branch is set to branch taken, *no* means not taken. The prediction bit for the conditional branch of the **if** was set to *yes* in all cases, as the particular setting is irrelevant. *No* Branch Spreading means that compares were immediately followed by the conditional branches, *yes* means that code motion was used to separate the compare and branch. For comparison, the CRISP code for the loop before and after branch folding is shown in Table 3.

The results of running these programs are shown in Table 4. The CRISP Execution Unit is capable of issuing a new instruction every clock cycle. In most pipelined machines, one instruction per cycle is the theoretical peak rate. Case A is used as a performance reference. Because of pipeline breakage case A requires an average of 1.48 cycles to execute each instruction. (Note that pipeline breakage might be much worse without the Decoded Instruction Cache, which shortens the execution pipeline.) By simply setting the branch prediction bit properly, case B speeds up by a factor of 1.3. In case C branch folding is turned on. The effect here is that the Execution Unit does not have to issue a separate instruction for the branches. Technically, the Execution Unit is only issuing instructions at a rate of 1.22 cycles per instruction, when instructions mean those issued by the the pipeline. However, the apparent number of number of clocks per instruction when viewed as a black box is 0.90, so CRISP appears to be executing 1.1 instructions every clock cycle. The performance for case C is still degraded due to incorrect prediction for the conditional branch of the **if** every other time through the loop. Use of branch spreading in case D allows the proper branch direction for this conditional branch to be determined correctly every time. The performance for case D is twice that of case A, with an apparent instruction execution rate of 1.35 instructions per cycle. In the loop (discounting the initial cycles of overhead) CRISP is issuing exactly 1 new decoded instruction every cycle despite branches. Effectively, all unconditional and conditional branches are executed in zero time.

Finally, case E provides a comparison against delayed branch schemes. Delayed branch is similar to branch spreading where it is mandatory to schedule instructions after the branch. By turning off branch folding, but using branch spreading, the relative improvement is 1.5, only half as much improvement as with branch folding. For this example both machines are executing 1.01 cycles/issued-instruction, CRISP's advantage over delayed branch is in executing fewer instructions.

The performance improvements shown for the example are meant to be illustrative, not an indication that this performance improvement will happen in every case. The actual improvement is a function of the particular application being run.

```
main()
{ int i, j, zeros, ones, sum;
        j = ones = zeros = 0;
        for( i=0; i < 1024; i++ )
        {     sum += i;
              if     ( i & 1 )
                       odd++;
              else     even++;
              j = sum;
        }
}
```

Figure 3. A C program for evaluation.

CRISP Total of 9734 instructions			VAX Total of 9736 instructions		
Opcode	Count	Percent	Opcode	Count	Percent
add	3072	31.55%	incl	2048	21.04%
if-jump	2048	21.04%	jbr	1536	15.78%
cmp	2048	21.04%	movl	1026	10.54%
move	1027	10.55%	cmpl	1025	10.53%
and	1024	10.52%	jgeq	1025	10.53%
jump	513	05.27%	addl2	1024	10.52%
enter	1	00.01%	bitl	1024	10.52%
return	1	00.01%	jeql	1024	10.52%
			clrl	2	00.02%
			ret	1	00.01%
			subl2	1	00.01%

Table 2. Instruction counts for the program of Figure 3.

CRISP code without Branch Spreading			CRISP Code with Branch Spreading		
_4:	add	sum,i	_4:	and3	i,1
	and3	i,1		cmp.=	Accum,0
	cmp.=	Accum,0		add	sum,i
	ifTjmpy	_5		add	i,1
	add	odd,1		mov	j,sum
	jmp	_6		ifTjmpy	_5
_5:	add	even,1		add	odd,1
_6:	mov	j,sum		jmp	_6
	add	i,1	_5:	add	even,1
	cmp.s<	i,1024	_6:	cmp.s<	i,1024
	ifTjmpn	_4		ifTjmpy	_4

Table 3. CRISP Code for loop before and after Branch Spreading.

Comparison to Other Schemes

The techniques used in the CRISP microprocessor for reducing the cost of branches are a significant improvement over schemes which have been implemented in other machines. With delayed branch, the branch itself must still be executed; this requires at least one clock cycle. Machines having a single compare and branch can get some of the effect that CRISP requires branch folding to achieve, but they have no assistance with unconditional branches.

The use of a Branch Target Buffer, as described by Lee and Smith, can provide reasonable branch prediction based on dynamic history. On most machines using a BTB, the benefit is in avoiding pipeline breakage, both conditional and unconditional branches still require at least a clock for their execution. McFarling and Hennessy discuss the possibility of storing the target instruction in the BTB as a way to eliminate the cost of unconditional branches. The effectiveness of BTB's varies. Results for the MU5 show only a 40-65 percent correct prediction rate for an eight entry jump-trace, barely better than tossing a coin. Lee and Smith report effectiveness as high as 78 percent for a BTB of 128 sets of 4 entries. The BTB is usually implemented as a separate associatively addressed function unit. Branch Folding in CRISP is different in that it adds the next address fields to each instruction in the Decoded Instruction Cache, and occurs later in the pipeline. Branches in CRISP need not occupy a pipeline slot to themselves. A final deciding factor against the BTB in microprocessors might be the implementation cost. A 128 set 4 entry BTB would be nearly as large as our entire microprocessor chip.

Practical Considerations

Use of a Decoded Instruction Cache simplifies the implementation in many ways. For example, one of the decoded instruction bits is used exclusively to specify whether the instruction can modify the condition code flag. This bit is carried with each pipeline stage in the execution unit, and used to evaluate if the correct branch target can be determined without prediction. The use of a Decoded Instruction Cache helps improve pipeline performance by separating decoding from instruction execution. This can smooth out pipeline blockages giving the same advantages as Kogge's description of the use of a FIFO queue,[29] but without the problems of having to flush the queue when a branch occurs.

For our machine, including a Next-PC field and alternate Next-PC field in the Decoded Instruction Cache caused the cache to grow 64-bits wider than it otherwise might have been. This extra memory turned out not to cost any area in CRISP since the pitch of the datapath was the constraining factor. Many of our initial design decisions were based on trying to make CRISP a memory intensive, rather than control-logic intensive design. In retrospect, this decision turned out to be far wiser than we had originally imagined.

Finally, true zero delay for branches can only occur if the instruction cache has a hit. Being careful with the design of the instruction prefetch unit and instruction cache should not be overlooked.[30]

Conclusion

We have described a general technique called branch folding, and associated hardware/software techniques that have the potential to totally eliminate many of the problems with branches that have plagued high performance computer designs. The number of instructions issued to the pipeline to execute a given program can be reduced by the number of branches in the program. This can be as much as 30% of instructions being executed. The resulting reduction in number of instructions executed and eliminating pipeline breakage can lead to significant performance improvements.

Although the implementation was described only for the CRISP Microprocessor, the technique may prove valuable for other computer designs, particularly those for which compatibility does not allow the use of other techniques such as delayed branch.

Case	Branch Folding	Branch Prediction	Branch Spreading	Cycles to Execute	Instructions Issued	Relative Perf.	Issued Cycles/Instr	Apparent Cycles/Instr
A	no	no	no	14,422	9,734	1.0	1.48	1.48
B	no	yes	no	11,359	9,734	1.3	1.16	1.16
C	yes	yes	no	8,789	7,174	1.6	1.22	0.90
D	yes	yes	yes	7,250	7,174	2.0	1.01	0.74
E	no	yes	yes	9,815	9,734	1.5	1.01	1.01

Table 4. Execution Statistics on CRISP for program of Figure 3.

References

1. A. D. Berenbaum, B. W. Colbry, D. R. Ditzel, R. D. Freeman, H. R. McLellan, K. J. O'Connor, and M. Shoji, "A Pipelined 32b Microprocessor with 13Kb of Cache Memory," *Proceedings of the 1987 International Solid State Circuits Conference*, pp. 34-35 (February, 1987).
2. D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor," *Proceedings of the 14th Annual Symposium on Computer Architecture* (June 2-5, 1987).
3. D. Morris and R. N. Ibbet, *The MU5 Computer System*, Springer-Verlag (1979), p. 59.
4. Douglas W. Clark and Henry M. Levy, "Measurement and Analysis of Instruction Use in the VAX-11/780," *The 9th Annual Symposium on Computer Architecture* **10**(3), pp. 9-17 (April, 1982).
5. Cheryl A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 177-184 (March 1982).
6. L. J. Shustek, *Analysis and Performance of Computer Instruction Sets*, Stanford Linear Accelerator Center (May 1978). Ph.D. Dissertation
7. Werner Bucholz, Editor, *Planning a Computer System: Project Stretch*, McGraw-Hill (1962), pp. 238-239.
8. George Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47 (March, 1982).
9. David A. Patterson, "RISC-I: A Reduced Instruction Set VLSI Computer," *Proceedings of the 8th International Symposium on Computer Architecture* (May 1981).
10. J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," *Proceedings of the CMU Conference on VLSI Systems and Computations* (October 1981).
11. J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riordan, C. Rowan, and D. Van't Hof, "A CMOS RISC Processor with Integrated System Functions," *Spring COMPCON 1986*, p. 126.
12. J. S. Birnbaum and W. S. Worley, "Beyond RISC: High-Precision Architecture," *Spring COMPCON 1986*, p. 40.
13. S. McFarling and J. Hennessy, "Reducing the Cost of Branches," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 396-403.
14. R. W. Holgate and R. N. Ibbet, "An Analysis of Instruction-Fetching Strategies in Pipelined Computers," *IEEE Transactions on Computers* **C-29**(4), pp. 325-329 (April 1980).
15. D. Morris and R. N. Ibbet, *The MU5 Computer System*, Springer-Verlag (1979).
16. D. W. Anderson, "The System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development* **11**(8), pp. 8-24 (January 1967).
17. W. D. Connors, "The IBM 3033: An Inside Look," *Datamation*, pp. 198-218 (May 1979).
18. H. Schorr, "Design Principles for a High-Performance System," *Proceedings of the Symposium on Computers and Automata* **XXI**, pp. 165-192 (April, 1971).
19. Robert G. Wedig and Marc A. Rose, "The Reduction of Branch Instruction Execution Overhead Using Structured Control Flow," *The 11th Annual International Symposium on Computer Architecture* **12**, pp. 119-125, 3 (June, 1984).
20. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* **17**(1) (January, 1984).
21. A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan, "Introduction to the CRISP Instruction Set Architecture," *Proceedings of the 1987 Spring COMPCON*, pp. 86-90 (February, 1987).
22. M. G. H. Katevenis, *Reduced Instruction Set Computers for VLSI*, MIT Press (1984), p. 150.
23. R. D. Russell, "The PDP-11: A Case Study of How Not to Design Condition Codes," *Proceedings of the 5th Annual Symposium on Computer Architecture*, pp. 190-194 (April 1978).
24. J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11 (March 1982).
25. James E. Smith, "A Study of Branch Prediction Strategies," *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148 (June, 1981).
26. J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer* **17**(1) (January, 1984).
27. S. Bandyopadhyay, V. Begwani, and R. Murray, "Compiling for the CRISP Microprocessor," *Proceedings of the Spring 1987 COMPCON*, pp. 96-100 (February, 1987).
28. J. L. Hennessy and T. R. Gross, "Optimizing Branch Delays," Computer Systems Lab Technical Report, Stanford University (1981).
29. Peter M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill (1981), pp. 237-243.
30. Hubert Rae McLellan, Jr., "Instruction Prefetch Strategies in a Pipelined Processor," *Master of Science Thesis*, Massachusetts Institute of Technology (February 1983).

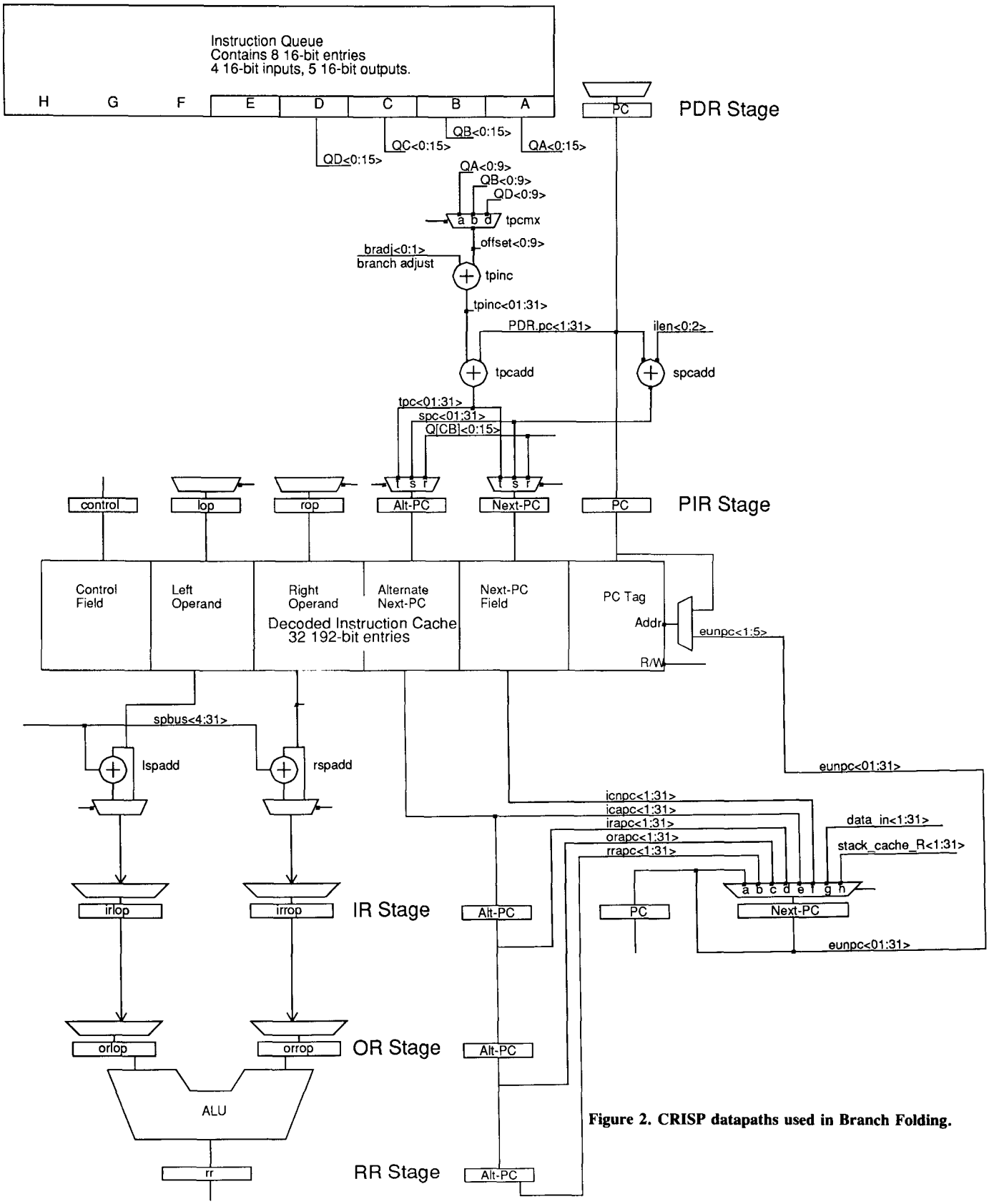

Figure 2. CRISP datapaths used in Branch Folding.