Future Branches – Beyond Speculative Execution

Bill Appelbe[1], Raja Das[2], and Reid Harmon[2]

[1] Department of Computer Science, RMIT, Melbourne, Victoria 3001, Australia
[2] College of Computing, Georgia Institute of Technology Atlanta, GA 30327, USA

Abstract. Speculative execution of conditional branches has a high hardware cost, is limited by dynamic branch prediction accuracies, and does not scale well for increasingly superscalar architectures.
Future branches are additional branch instructions that overcome the performance bottleneck of conventional branches. Future branch instructions includes a branch source address (the location of the impending conditional branch) as well as the branch target. The branch actually occurs when the program counter reaches the source address. If a future branch is executed before instruction fetching reaches the branch source, then there are no pipeline stalls or prediction necessary.
Benchmark micro-architecture simulation studies show that at high superscalarities, losses to speculative execution consistently are higher than 10%, and these losses can be avoided by future branches. In addition, a hardware implementation of future branches for the PowerPC 604 has a very modest cost.

1 Introduction

Studies have shown that from 1.6% to 22% of instructions executed are branches or conditional branches, with most non-scientific programs much closer to 22%. A branch instruction interrupts the instruction pipeline unless the branch target address, and the instructions at that address, are known before the branch instruction is decoded, and the direction of a conditional branch is known. Branch caches enable the branch target address and subsequent instructions to be determined with high probability at a modest hardware cost [9]. By contrast, determining the direction of a conditional branch with high probability is far more difficult. Most modern superscalar processors use *speculative execution*, in which the direction of the branch is predicted dynamically using a history of previous branch directions.

Studies of the prediction accuracy of processors with sophisticated dynamic branch prediction show prediction accuracies around 90% for non-scientific applications (e.g., Ultra SPARC [11]: SPECint 88%, SPECfloat 94% [11]; Power 620 [9]: SPEC composite 90%). Performance analysis of the Pentium (at most two-way superscalar, but with sophisticated branch prediction), show that on several of the SPEC benchmarks the overhead of incorrect branch prediction is between 5% and 8% of overall execution time [2]. Research continues into more

sophisticated mechanisms for using branch histories to improve branch prediction accuracy, but gains tend to be slight[6] and increase hardware costs further.

Aside from its high hardware cost and imprecision, another fundamental problem with speculative execution is that it does not scale well with increasingly superscalar processors. The frequency of branch instruction fetches and predictions per cycle is proportional to the mean IPC (Instructions Per Cycle executed). In addition, highly superscalar processors need large instruction buffers to be able to find more ILP (Instruction Level Parallelism). This increases the mean number of cycles between a conditional branch fetch (prediction) and execution (determining the branch outcome). This implies that the number of cycles lost to a missed prediction should increase with increasing superscalarity.

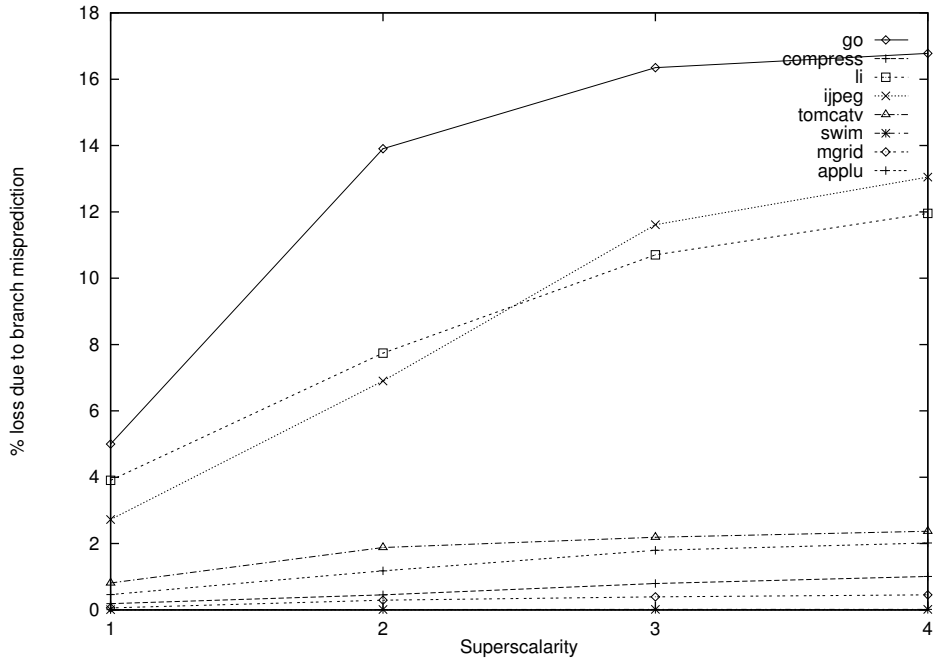

Fig. 1. Cycles Lost to Branch Misprediction v. Superscalarity

Figure 1 plots the percentage of cycles lost to speculative execution against increasing superscalarity for 8 of the SPEC95 benchmarks, using the DLX microarchitecture simulator described in the next section. Loses flatten out at a superscalarity of 3 or 4 because these benchmarks have limited available ILP. Benchmarks which have "unpredictable" branches, (*go*, *li*, and *ijpeg*), have losses due to branch misprediction which increase rapidly with increasing superscalarity until ILP is exhausted.

Decoupling the branch decision from the branch itself has been used in many

architectures, ranging from those which have separate instructions for computing the branch condition to branch instructions with a variable number of delay slots[4]. However, these have the limitation that the computation of a branch condition (direction) and the branch itself must be close together. This usually limits placing the instructions that evaluate the branch condition in the same basic block as the branch instruction itself.

To avoid dynamic branch prediction or pipeline stalls, the branch condition must be evaluated before the branch instruction is fetched. To ensure this, the distance between the branch condition's instructions and the actual branch instruction (in instructions fetched), must then exceed the size of the instruction or reorder buffer (of fetched instructions which are awaiting execution and retirement). Modern superscalar processors have instruction buffers of up to 50 instructions, which far exceeds the mean size of basic blocks. Hence any effective scheme for evaluating branch conditions before branch instructions must be able to move the branch condition far ahead of the branch itself, outside its basic block.

Future branches encode both the branch source address (when the branch will occur) together with the branch target address (where the branch will go to) in a single instruction. Future branches allows a branch condition to be evaluated anywhere in a program, subject to just two constraints:

- Data and control dependencies must be preserved (using program transformations such as *code hoisting* for the branch condition[1])
- Future branches for the same source address must be executed in order

The remainder of this paper gives performance results for future branch instructions, followed by an outline of an architecture definition for future branch instructions, and hardware design for incorporation in the PowerPC 604.

2 Simulation Studies

Determining the effectiveness of future branches requires both an implementation of future branches in "hardware" and a compiler that can transform regular conditional branches in source programs into future branch instructions for the target hardware. Instead of a hardware implementation, we extended a micro-architecture simulator, and have incorporated algorithms for generating future branches into a compiler toolkit called *GiL* based upon *gcc*.

Future branches can be added to almost any instruction set architecture. For the simulation studies we used the *DLX* instruction-set[5], as we had access to a reconfigurable micro-architecture simulator for *DLX*, *SuperDLX* from McGill university. We have extensively modified the simulator so that it now includes features such as: future branches, *oracle branches* (100% accurate speculative execution), and a reconfigurable multilevel cache simulator. latency of all functional units to be specified.

2.1 Compiling for Future Branches

To compare future branches with oracle or speculative execution, it is necessary to transform the program to use future branches. At the source level, most branches are predictable well before we reach them. In a classic `for loop` a programmer can easily determine whether the next iteration will be taken at the start of the current iteration. If the for loop test is `I <= N`, then if `I <= N-1` at the start of the iteration we know that the next iteration will be taken.

In practice, many loops and branches within loops are far more difficult to predict many instructions beforehand. To implement future branches, we have modified *GiL*, based upon *gcc*, to support dependence analysis and have implemented a simple algorithm to recognize when a branch can be converted to a future branch. We analyzed 27 files in the *gcc* distribution looking for branches whose predicate can be *trivially hoisted*: all the virtual registers needed for its evaluation are defined *exactly* once within the loop it belongs to. A trivially hoistable conditional branch can always be moved to the start of the body of the loop containing it.

Total number of conditional branches	4497
Total number of unconditional branches	2270
Not trivially hoistable	764

Table 1. Branch distribution in GCC

Table 1 shows the distribution of branches in the files we have analyzed. Only about one seventh of the branches are not trivially hoistable. Many of the non hoistable branches could be hoisted by a better algorithm[3]

Hoisting a branch may increase the size of the program or the number of registers needed. However, analysis of the same same branches in *gcc* showed that more than 60% of the conditional branches required hoisting 3 instructions or less, and more than 85% of conditional branches required hoisting 6 instructions or less.

Analysis of a suite of C benchmarks indicates that only .2% (espresso) to 5% (gcc, spice) of branches are for a relative distance of more than 256 instructions[4]. Hence both the relative source and target address can be encoded in a 16 bit displacement common to many instruction sets.

[3] Hand inspection has revealed that the reason why most branches are not trivially hoistable is that one definition is inside a conditional. The branch is usually still hoistable, to a point just after the conditional. To recognize this we need ϕ functions and SSA form.

[4] Branches for longer distances can always use a branch to an unconditional branch. Further, many architectures can support a larger branch address space.

Thus, it appears that the majority of conditional branches can be hoisted significantly earlier in the instruction stream and hence converted to future branches. However, there are overheads in hoisting branches that imply that future branches will will never equal the performance of an oracle branch prediction unit:

- A few branches cannot be hoisted[5]. Fortunately, such cases are not common in practice.
- Hoisting branches may require copying registers
- Hoisted branches require unbranches, or future branch cancelation, on some exits

2.2 Speculative Execution vs Future Branches

Our compiler cannot yet generate future branch instructions for large benchmarks. Hence we have resorted to hand-coded assembler implementation of future branches. Our hand-coded implementations focuses upon two benchmarks: *Livermore Loops #24* and the procedure *essen_parts* of the SPEC benchmark *espresso*[6]

Table 2 summarizes the results for LL #24 (Superscalarity means the number of fetch and decode units, for this benchmark there were two of all other units).

Superscalarity	**Speculative Branch Depth**				**Future**	**Percentage**
	2	3	4	∞	**Branches**	**Speedup**
3	578	541	541	541	516	4.8%
4	543	526	490	490	428	14.5%
5	528	514	498	490	428	14.5%

Table 2. Cycles Required to Execute Benchmark LL #24

As Table 2 shows, the use of Future Branches resulted in significant performance improvement, despite a speculative branch prediction accuracy of 83.2%. Table 2 also illustrates the need for increasingly deep, and expensive, branch prediction at higher superscalarity. The performance gain using future branches is a little surprising because of the four extra instructions in the inner loop of the future branched version. The extra instructions are needed because the inner condition loop condition is a recurrence[7].

[5] It is not very difficult to construct a pathological case in which no branch can be hoisted, as all the computation in a routine is devoted to determining a branch outcome[1]. However, even in these cases, future branches can be used similarly to static branch prediction

[6] We first used LL #24 as it was the shortest benchmark; we used the routine *essen_parts* in *espresso* as most time was spent in this routine, as measured by *gprof*

[7] The test in each iteration is dependent upon the test result in the previous iteration

The *essen_parts* routine is a much larger benchmark than LL #24. The routine was about 50 lines of hand-optimized C, corresponding to about 500 lines of DLX assembler. It contained 3 loops, 10 if statements, and 4 goto's. We obtained the times for this routine by extracting them from the SuperDLX log files for the first call to *essen_parts* (timing from execution of the entry to subroutine return) for a 4-way superscalar processor.

Table 3 summarizes the results for *essen_parts*.

Superscalarity	**Non-Speculative**	**Speculative Branch Depth Oracle**		**Future Branches**
		∞		
3	293	208	122	172

Table 3. Cycles Required to Execute *essen_parts* in *espresso*

In this case, the future branched version was about 17% percent faster than the speculatively executed version. Other runs and calls of *essen_parts* gave similar results. It is interesting to note that every one of the branches in *essen_parts* could be converted to future branches, despite the convoluted C code. No effort whatsoever was spent trying to hand optimize the future branch assembler code to skew the results (life has many more worthwhile challenges).

3 Future Branch Instructions

The future branch instructions proposed in this section are extensions of the PowerPC instruction set. We have also designed future branches for the Motorola 88110 instruction set[7].

The PowerPC instruction set supports four different branch instructions [12] [8] including unconditional branch b, branch conditional bc, branch conditional to counter register bcctr, and branch conditional to link register bclr. Of these instructions, just the b and bc instructions are considered here for use with future branching because the future branch mechanism described in this paper presently only supports branches with immediate targets. The bcctr and bcclr instructions branch to a target contained in a register and are far less commonly used than immediate targets.

The additional instructions to support future branching are the fb (future branch unconditional), fbc (future branch conditional), and ufb (undo future branch). The fb instruction is the future branch counterpart to the b instruction, and the fbc instruction is the counterpart to the bc instruction. The ufb instruction has no corresponding PowerPC branch instruction as it is unique to the future branch design. Figure 2 illustrates the structure of the future branch instructions.

The fb instruction is a modification of the PowerPC b instruction. The fb instruction includes an additional Source field not found in the b instruction. that

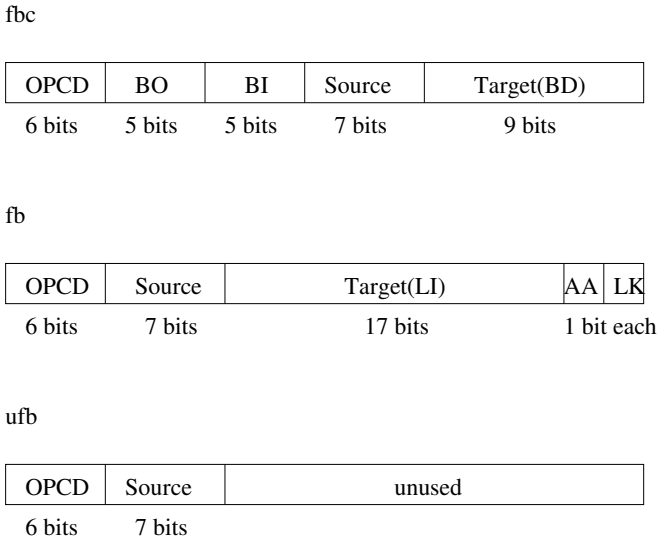

Fig. 2. PowerPC Future Branch Instructions

tells where the corresponding b instruction will be, relative to the fb instruction. The displacement is encoded in seven bits which provide for ±64 instruction displacement (± 256 bytes). Thus the compiler can place the b instruction and fb instructions up to 64 instructions apart in either the forward or reverse direction. The Target field of the fb instruction (which corresponds to LI field in PowerPC b instruction) is reduced from 24 bits in the b instruction to 17 bits in the fb instruction. This provides for a displacement of ±64K instructions. Displacements larger than 64K instructions will not be able to take advantage of the fb instruction. It is important to note that in the future branch implementation, the Target displacement is calculated *relative to the* fb *instruction not the branch source*. This implementation was adopted to speed up the target address calculation in the decode stage of the processor pipeline (See Section 3). As Figure 2 shows, the AA (absolute address) and LK (branch and link) bits were retained for the fb instruction. This was done to make the fb instruction fully compatible with the b instruction except for the target range.

The fbc instruction is the conditional branch counterpart instruction in the future branch design. The fbc instruction is very similar to the bc instruction. The major differences are the lack of the AA and LK fields, the 7-bit source field, and a reduced width Target field (BD in the PowerPC bc instruction). The BO (branch condition) and BI (condition register bit index) fields are identical to the PowerPC bc instruction [12] [8]. These two fields determine the branch predicate.

The Source field encodes the relative distance to the corresponding bc instruction as did the Source field in the fb instruction. The 7-bit Source provides the future branch mechanism with ± 64 instructions displacement between the

future branch instruction (fbc) and the branch source instruction (bc). The Target (BD) field in this instruction is reduced from the 14 bits provided in the PowerPC bc instruction to 9 bits. This allows the fbc instruction to be paired with conditional branches that are ± 256 instructions from the fbc instruction. This amounts to a significant reduction in displacement range for the conditional branch (± 256 instructions vs. ± 32 K instructions), but the benchmarks that we have run indicate that 95% of all branches fall in this range and can thus use future branches without code motion or other tricks. As in the fb instruction, *the displacement calculation is relative to the fbc instruction not the bc instruction.* The AA and LK fields of the bc instruction are omitted from the fbc instruction. This was done to maximize the sizes of the Source and Target fields so as to increase the source and target range of the fbc instruction. The elimination of the AA and LK fields means that the compiler can not generate future branches for bc instructions that use absolute addressing or store subroutine return addresses in the link register.

The ufb instruction is a mechanism for the compiler to specify removal of future branch entries from the PBT (*Pending Branch Table*). This is handled by calculating a Source value using the 8-bit Source field. The source is used to index the PBT and remove the corresponding entry. Eight bits are used for the source of the ufb instruction to increase the range of the ufb instruction. This is in case the execution of the program has proceeded more than 64 instructions from the future branch source instruction when the ufb instruction is encountered.

4 Future Branch Hardware Implementation

The implementation of the future branch hardware is based on the hardware used in the Power PC 604 processor [9]. Thus our implementation of future branches for the PowerPC instruction set tries to exploit the existing 604 hardware rather than modifying it extensively. Obviously, this is by no means the only way to implement future branching for the 604 or other processors (we also developed an implementation for the 88110). This implementation was simply chosen as a reference point. Further details of the implementation are available from *ftp://ftp.cc.gatech.edu/pub/people/bill/papers/fb-hardware.ps.*

4.1 PowerPC 604 Instruction Pipeline

The PowerPC 604 pipeline [9] was used in developing the future branch hardware implementation. Figure 3 shows the 604 pipeline and what future branch processing occurs at each stage.

The following list identifies the future branch processing that takes place at each stage in the processor pipeline.

- Fetch
  - Comparison of the Pending Branch Table entries with the Fetch Counter
  - Modify Fetch Counter on Correct Comparisons

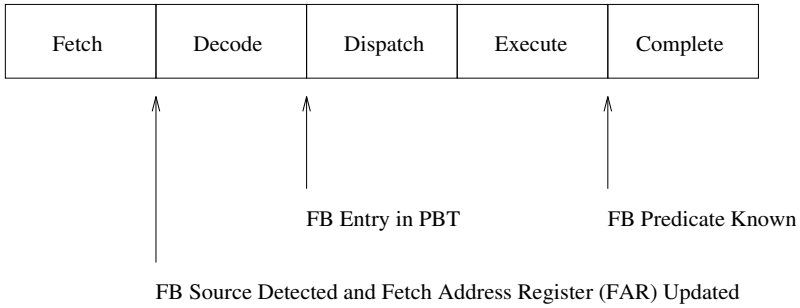

Fig. 3. PowerPC Instruction Pipeline with Future Branches

- Decode
  - Future Branch Source Calculation
  - Future Branch Target Calculation
  - Modify Pending Branch Table
- Dispatch
  - Future Branch Instruction Sent to EX unit or Reservation Station
- Execute
  - Future Branch Predicate Calculated
  - Pending Branch Table entry Modified to Reflect Calculation

The PBT entry (Section 4.2) is made at the end of the Decode cycle. This creates a gap of up to 8 instructions between the fetch of the future branch instruction and the updating of the PBT. This requires that at a minimum, the compiler hoist the branch instruction 8 instructions above the original branch source. This is required so that the PBT modifications that result from executing the future branch instruction are present in the PBT before the branch arrives. The decode stage is treated in greater detail later. Eight instructions are adequate only for unconditional future branches fb since predicate calculation must be done for conditional future branches.

The predicate is not known until at least the fourth cycle after fetching the future branch instruction, which means that the compiler must hoist the future branch at least 16 instructions (worst case) before the branch source for the predicate to be available. If the predicate is not available when the source is fetched, the branch prediction mechanisms of the PowerPC 604 will take over (BTAC, BHT)[9].

During the Fetch stage, the Fetch Address Register (FAR) is being compared with the entries in the PBT. When a match is indicated, the FAR is modified based on the outcome of the branch. If the future branch predicate has already been calculated, the FAR is modified and instructions are fetched non-speculatively. If, however, the predicate has not yet completed, the BTAC entry is used to predict the outcome of the branch. The fetch counter is still modified based on this prediction, but the instructions are fetched speculatively. The Fetch stage is treated in greater detail later.

4.2 Pending Branch Table

The PBT contains the results of executing a future branch instruction (fbc, or fb). The PBT is fully associative and contains, at this time, an unspecified number of entries (the number of entries limits the number of pending future branches, but the compiler determines this statically, and a dozen entries seems more than adequate for benchmarks we have studied). Figure 4 details one such entry in the PBT.

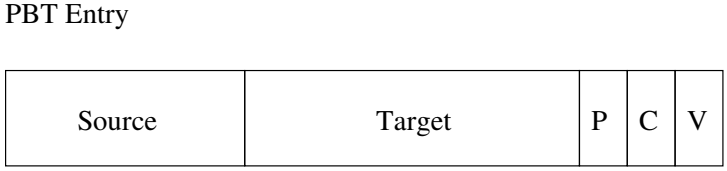

Fig. 4. Pending Branch Table

The Source entry is the absolute address of the branch source instruction. The branch source instruction is the actual branch instruction (b, bc) that corresponds to the future branch instruction that made the cache entry. The Source entry is made during the Decode stage of the pipeline. Likewise, the Target entry is the absolute address of the branch target, and it is also entered into the cache during the Decode stage.

The P bit (Prediction) indicates whether the branch is taken or not. If the P bit is set, the branch Target is the address of the instruction that follows the branch source instruction. If the P bit is clear, the instruction executed after the branch source is the instruction at the next sequential address in program memory. The C bit (Calculated) indicates whether the P bit is a predicted or calculated value. If the C bit is set, the value of the P bit is a value that was obtained by evaluating the predicate conditions. If the C bit is clear, the P bit is a prediction. In this implementation using the PowerPC 604 architecture, a cleared C bit defers the branching decision to the BTAC (Branch Target Address Cache). The V bit (Valid) simply indicates whether the entry in the cache is valid or not. A set V bit indicates a valid entry. The ufb instruction clears the V bit for the specified entry as the means of removing the future branch instruction from the cache. Entries with a cleared bit that match the Source field are ignored unless the BTAC indicates that the instruction is a branch.

4.3 Fetch Using Pending Branch Table

The entries in the PBT specify the addresses of branch instructions that will be encountered later in the flow of instructions. These addresses (Source fields in the PBT) are being compared with the Fetch Address Register (FAR) every cycle

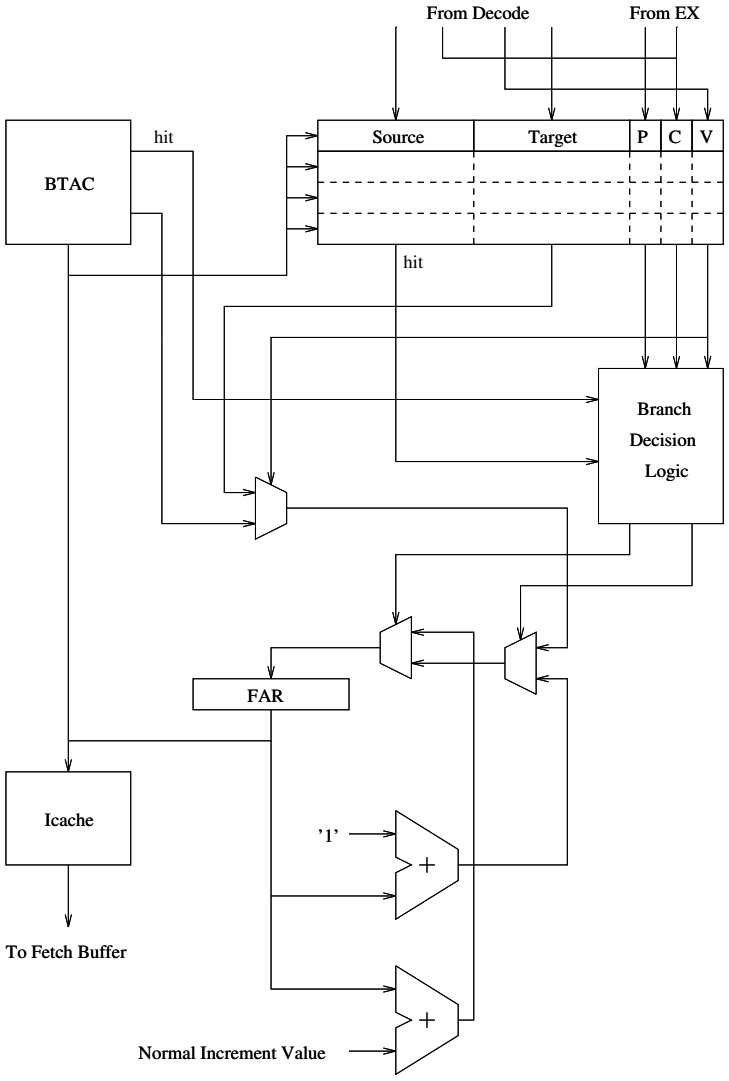

Fig. 5. Future Branch Fetch

to determine whether the branch for which the entry applies has been fetched. Figure 5 illustrates the hardware required for future branch fetching.

The entries in the PBT are checked every time an instruction block is fetched from the I-cache.

If there is a hit, the Target, Predicted bit (P), Calculated bit (C), and Valid bit (V) are presented as output from the cache. The three bit values (P, C, and V) are presented to the Branch Decision Logic, which ultimately decides the outcome of the branch for the next instruction fetch cycle. At the same time,

the Branch Target Address Cache (BTAC) is being accessed, and the indication of a hit or miss is sent to the Branch Decision Logic. If the C bit and V bit are set, this means that the future branch instruction has completed the predicate calculation, and the fetch unit should fetch next at either the Target or the next sequential instruction. The decision about whether to take the Target path or the next instruction is decided based on the P bit. In either case, the instruction fetch hardware can continue fetching and the instructions fetched are not speculative.

If the V bit is not set than this means that the fetched block may not contain a branch instruction (no valid PBT Entry). In this case the hit/miss indication of the BTAC is used to make this determination (as in the case of a branch with no corresponding future branch instruction). If the BTAC indicates a taken branch, then the BTAC target address is used rather than the one in the PBT as that entry is invalid (the V bit selects which target address to use).

If the V bit is set, but the C bit is not, this means that the future branch instruction (fbc) has not yet finished the predicate calculation (D-cache miss delay, etc.). In this case, in order to keep up with the processing rate, the next instruction block will have to be fetched speculatively using the BTAC. If the BTAC hits, the branch is taken, and if the BTAC misses, the next sequential instruction is taken.

We have throughly tested a similar implementation of future branches in the DLX micro-architecture simulator. The only modification we found necessary was allowing for up to four conditional branches to be pending for the same target address. PBT entries have four (Target, P, C, and V) fields for each Source, and two 2-bit counters indicating the current and next available Target entry.

4.4 Hardware Cost

The hardware costs presented here are based on the PowerPC 604 implementation, and they are compared with the existing hardware of the PowerPC 604 microprocessor.

We estimate a transistor count for our implementation to be 30,000. This includes all the additional hardware added to the existing 604, but it does not include hardware that is used in future branching but already exists in the 604 (e.g., BTAC). We estimate that these transistors would occupy 6.5 mm^2 using a .5 micron process. Using data on the 604 die [9] size (approximately 196 mm^2, our future branch implementation would use an additional 3.3% of the existing 604 die area. We believe that these estimates are conservative, and that the actual future branch implementation would require even less real estate than calculated above.

5 Conclusion and Future Work

Our analysis has shown that the traditional approaches to avoiding branch delays, speculative execution, does not scale with increasingly superscalar architectures. Future branches represent a simple but radical departure from traditional

approaches to improving performance of branch instructions. The concept is compatible with almost any instruction set architecture, and is simpler, and with optimizing compiler technology offers higher performance than speculative execution.

Although this paper has demonstrated the feasibility of future branches from architecture, hardware, and compiler viewpoints, there are many open questions and unresolved issues that we are actively addressing at present, including:

- How do oracle, speculative, and future branch execution compare across larger benchmark suites?
- What is the optimum number of entries in a future branch cache?
- Should future branch targets be prefetched from the instruction cache, and how do future branches affect the design of instruction caches, etc.
- What hardware support is needed for saving future branch state, and what is its cost in cycles?
- What is the interplay between future branch instructions, register allocation, and instruction scheduling in a compiler?

References

1. Bill Appelbe, Reid Harmon, Phil May, Scott Wills, and Maurizio Vitale. Hoisting branch conditions – improving super-scalar processor performance. In *Eighth Workshop on Languages and Compilers for Parallel Computing*, 1995.
2. M. Bekerman and A. Mendelson. A Performance Analysis of Pentium Processor Systems. *IEEE Micro*, 15(5):72–83, October 1995.
3. Sreeram Duvvuru and Siamak Arya. Evaluation of a Branch Target Address Cache.
4. J. R. Goodman, J. T. Hseih, K. Liou, A. R. Pleszkun, P.B. Schechter, and H. C. Young. Pipe: A vlsi decoupled architecture. In *Proceedings of The Twelfth Annual Symposium on Computer Architecture*, pages 20–27, 1989.
5. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
6. Stuart Sechrest, Chih-Chieh Lee, and Trevor Mudge. Correlation and aliasing in dynamic branch predictors. In *Proceedings of The 23nd Annual Symposium on Computer Architecture*, pages 22–31, 1996.
7. Motorola Semiconductor Products Sector. *MC88110 Second Generation RISC Microprocessor Users Manual.* Motorola, Inc., Phoenix, Arizona, 1991.
8. Motorola Semiconductor Products Sector. *PowerPC 601 RISC Processor Users Manual.* Motorola, Inc., Phoenix, Arizona, 1993.
9. S. Peter Song and Marvin Denman. The PowerPC 604 Microprocessor. *IEEE Micro*, pages 8–17, October 1994.
10. Tom Thompson and Bob Ryan. PowerPC 620 Soars. *BYTE*, pages 113–120, November 1994.
11. Peter Wayner. SPARC Strikes Back. *BYTE*, pages 105–112, November 1994.
12. Shlomo Weiss and James E. Smith. *POWER and PowerPC.* Morgan Kauffman, San Francisco, California, 1994.

This article was processed using the LaTeX macro package with LLNCS style