Contents

I This is a Part **7**

1 Branch Prediction **9**

Philip G. Emma IBM T.J. Watson Research Laboratory

1.1 What Are Branch Instructions and Why Are They Important? Some History . . . 9
1.2 Why are Branches Important to Performance? . . . 15
1.3 Three Dimensions of a Branch Instruction, and When to Predict Branches . . . 21
1.4 How to Predict Whether a Branch is Taken . . . 26
1.5 Predicting Branch Target Addresses . . . 31
1.6 The Subroutine Return - A Branch That Changes Its Target Address . . . 36
1.7 Putting it All Together . . . 39
1.8 High ILP Environments . . . 45
1.9 Summary . . . 46

References **49**

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

List of Tables

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

List of Figures

1.1 Von Neumann's IAS Machine 12
1.2 The Pipelining Concept 17
1.3 Instruction Prefetching 20
1.4 An RS-Style Pipeline with Various Approaches to Branch Prediction 23
1.5 Simple State-Machines for Branch Prediction 27
1.6 Decode-Time Branch Predictors 29
1.7 Branch Target Buffer 34
1.8 Subroutine Call and Return 36

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

Part I

This is a Part

Chapter 1

Branch Prediction

Philip G. Emma
IBM T.J. Watson Research Laboratory

1.1 What Are Branch Instructions and Why Are They Important? Some History 9
1.2 Why are Branches Important to Performance? 15
1.3 Three Dimensions of a Branch Instruction, and When to Predict Branches 21
1.4 How to Predict Whether a Branch is Taken 26
1.5 Predicting Branch Target Addresses 31
1.6 The Subroutine Return - A Branch That Changes Its Target Address . 35
1.7 Putting it All Together 39
1.8 High ILP Environments 45
1.9 Summary 46

1.1 What Are Branch Instructions and Why Are They Important? Some History

The branch instruction is the workhorse of modern computing. As we will see, resolving branches is the essence of computing.

According to the von Neumann model of computing, instructions are fetched, and then executed in the order in which they appear in a program. The machine that he built in Princeton at The Institute for Advance Studies during World War 2 (which we will call the "IAS machine") did exactly this [1] [2]. Modern processors might not actually do this (fetch and execute instructions in program order), but when they don't, they generally ensure that no outside observer (other processors, I/O devices, etc.) be able to tell the difference.

A program is a *static* sequence of instructions produced by a programmer or by a compiler. (A compiler translates static sequences of high-level instructions (written by a programmer or by another compiler) into static sequences of lower-level instructions. A machine (processor) executes a *dynamic* sequence of events based on the static sequence of instructions in the program.

Programs comprise four primary types of instructions:

1. Data *movement* instructions (called "Loads" and "Stores"),

2. Data *processing* instructions (e.g., add, subtract, multiply, AND, OR,

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

etc.),

3. *Branch* instructions, and
4. *Environmental* instructions.

The first two categories (movement and processing) is what the average person conceives of as "computing." It is what we typically do with a pocket calculator when we balance our checkbooks. In fact, Echert and Mauchley built ENIAC (slightly before IAS) using only these instructions. ENIAC comprised Arithmetic-Logic Units (ALUs), which operated on data using ALU control information (i.e., Add, Subtract, etc.) [3]. The calculations to be done were configured ahead of time by a human operator who connected the right units together at a patch-panel to create the desired dataflow.

ENIAC was used to generate "firing tables" for field artillery being produced for the War. Specifically, it was used to generate and print solutions to simple (linear) polynomials for specific sequences of inputs covering relevant operational ranges for the artillery (each cannon barrel is slightly different because of the process variations inherent to the manufacturing of things this large).

The kind of computation done by ENIAC is still done today, but today we don't normally think of these applications as "computing." For example, digital filters do exactly this - for a sequence of input data, they produce a corresponding sequence of output data. A hardwired (or configurable) filter doesn't execute "instructions." It merely has a dataflow which performs operations on a data stream as the stream runs "through" the filter. Although this is very useful, today we would not call this a "computer."

What von Neumann featured in his programing model was the ability for a program to change what it was doing based on the results of what it was calculating, i.e., to create a dynamic sequence of events that differed from the static sequence of instructions in the program. Prior to von Neumann, a "computer" (like ENIAC) merely had *dataflow*. There had been no working notion of a "program."

Clearly, Alan Turing had had a similar notion, since without the ability to make conditional decisions, a Turing machine would only run in one (virtual) direction. Godel is famous for a remarkable proof that there are meaningful functions that cannot be computed in a finite number of steps on a Turing machine, i.e., whether the machine will halt is not deterministic [4]. This is called "The Halting Problem." Any finite-length program without conditional decision points certainly would be deterministic (i.e., whether it halts could be determined).

Many claim that others - such as Charles Babbage - had had similar notions (conditional control) in the mechanical age of calculation [5].

Von Neumann conceived a "program" as comprising two orthogonal sets of operation that worked in conjunction:

1. A *dataflow* which did the physical manipulation of the data values, and

2. A *control flow*, which dynamically determined the sequence(s) in which these manipulations were done.

Together, these two things allowed a much more complex (today, we might say "capricious") set of calculations. In fact, von Neumann conceived this kind of computing with the goal of solving a set of nonlinear differential equations (from Fluid Dynamics) for the purpose of designing the detonation means for atom bombs. The War ended before the machine was fully operational.

The control flow of a program is implemented by instructions called *branch* instructions.

Recall that a (static) program is a sequence of instructions. When the program is loaded into memory so that it can be executed, each instruction in the sequence is physically located at a specific address. (In the vernacular of Computer Science, "address" means "location in memory.") Since the program is a static sequence, the addresses of its instructions are sequential as well. For example, a program that is loaded at address 100 will look like this in memory:

```
LOCATION        INSTRUCTION
    100         Instruction #1
    101         Instruction #2
    102         Instruction #3
```

and so on. If no instruction was a branch instruction, the execution of the program (dynamic flow of instructions) would be exactly the static sequence: Instruction #1, Instruction #2, Instruction #3, and so on.

A branch instruction causes the flow of the program to divert from the next sequential instruction in the program (following the branch instruction) to a different instruction in the program. Specifically, it changes the flow of addresses that are used to fetch instructions, hence it changes the instruction flow.

Recall that von Neumann conceived two essential (and orthogonal) aspects of a computation: a dataflow, and a control flow. The IAS machine embodied these concepts in separate "engines," which today are called the "Instruction Unit" (IU) and the "Execution Unit" (EU). Machines today can have a plurality of each of these; both real (for high instruction-level parallelism) and virtual (for multiple threads).

Conceptually and physically, the IAS machine looks like the diagram in Figure 1.1. In addition to the IU and EU, there is a memory (M) which holds the data to be operated on by the IU and EU, an Input/Output device (I/O) for loading data into M, and an operator's Console (C) that allows the operator to initiate I/O and subsequent execution.

By today's standards, this is an extremely primitive system, although it is considered to be the first "*stored program computer*", so called because it is the first machine having an actual program that was stored into memory where it was operated upon. This machine is primitive in two important ways.

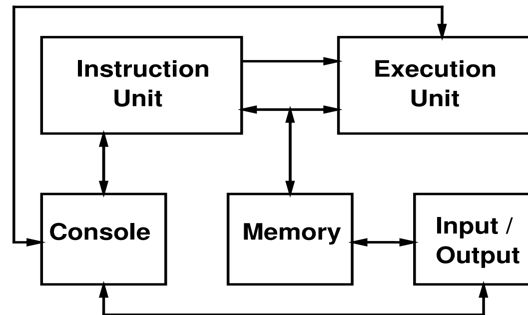

FIGURE 1.1: Von Neumann's IAS Machine.

First, the machine and its *programming model* (the way in which a programmer conceives of its operation) draws no distinction between an *instruction* and an *operand* (a datum to be operated on). To this machine, they are both just "data." Whether the content of a memory location is interpreted as an instruction or as an operand depends entirely on the context in which it is fetched.

When the IU issues an *instruction fetch* to the memory, the data that is returned is considered to be an instruction. The IU decodes that data, and initiates actions based on the values of the bits in the (predetermined) positions containing the *operation code* (called the "opcode field" of the instruction). When the IU issues an *operand fetch* to the memory, the data that is returned is sent to the EU, where it is subsequently processed. Since the machine draws no distinction between instruction data and operand data, instructions in a program can be (and were) operated on by the EU, and changed by the program as the program ran. Indeed, in those days, this was the programmer's concept of how to implement control flow (which would be anathema to any programmer today - in fact, this idea would probably not even occur to most students of Computer Science today).

The second important way in which the IAS machine is primitive is that there is no notion of virtual anything. While today, we have virtual systems, virtual processors, virtual processes, virtual I/O, and virtual everything else (all outside the scope of this chapter), a very basic concept appearing in

commercial processors (well after IAS) was virtual memory. The concept of virtual memory was first used in the Atlas computer [6]. The basic notion of virtual memory is that the logical addresses (as used logically by a program) did not need to be in direct correspondence with specific physical locations in the memory. Decoupling the logical from the physical enables three things:

1. The amount of logical memory (as conceived of by the program) can be much larger than the amount of real storage in the machine - providing that there is a dynamic way of mapping between the two as the program runs.

2. Programs can be written without specific knowledge of where they will reside in memory when they are run. This allows multiple coresident programs to use the same logical addresses although they are referring to different parts of the physical storage, i.e., it allows every program to "think" that it begins at address 0, and that it has all of memory to itself.

3. It allows parts of programs to be moved into and out of the memory (perhaps coming back to different physical locations) as they run, so that an operating system can share the available real memory between different users as multiple programs run concurrently.

Without virtual memory, programs written for the IAS machine could only refer to real locations in memory, i.e., a program was tightly bound to a specific physical entity in those days. (In modern programming, the logical and the physical are deliberately decoupled so as to allow a program to run on many different machines.)

In the IAS machine, this means that the program and its data is constrained to not exceed the size of the real memory. The IAS memory system has 1024 words in total. Can you imaging solving nonlinear differential equations using 1024 words total? Programs written to do this were exceedingly abstruse, with instructions changing each other's contents as the program ran. This is mind-boggling by today's standards.

A branch instruction has the form "If <condition> Then GoTo <address," where <condition> specifies some bits that can be set by other (temporally prior) instructions, and <address is the address of a place in the program to which instruction sequencing is diverted if <condition> is true. This allows the sequencing of instructions in a program to be dynamically altered by data values that are computed as the program runs (e.g., "If <end of record> Then GoTo <program exit>").

If the specified <condition> is the constant value "true," then the branch is always taken. This is called an *unconditional branch*, and it has the abbreviated semantics "GoTo <address." Branches for which <condition> is a variable (i.e., for which <condition> may or may not be true - depending on the results of certain calculations) are called *conditional branches.* As we will

see later, there is a big difference in the ability to predict conditional branches. (Hint: unconditional branches are always taken.)

As should be evident, adding branch instructions to a program increases the flexibility of the program (literally) infinitely. It enables a completely new dimension (called control flow) to be part of a calculation. More than any other thing (except perhaps virtualization), this simple concept has enabled computing machines to undertake tasks of previously unimaginable complexity - for example, financial modeling, engineering simulation and analysis (including the design of computers), weather prediction, and even beating the world's best chess player.

Without the branch instruction, computing would probably not have advanced beyond mundane calculations, like simple digital filters. With the branch instruction, we (humans) have enabled an infinitely more complex system of filters to evolve - called *data mining* - which is a proxy for "intuiting" (i.e., it has a nearly psychic overtone).

Today, user programs comprise (primarily) movement instructions, processing instructions, and branch instructions. Above, we had also mentioned "*environmental instructions*" as a fourth category. Today, this last category is primarily limited to privileged operations performed by the *Operating System*, and not usually allowed by user programs. These are mainly used to facilitate aspects of virtualization - to manage and administer physical resources via the manipulation of architected state that is "invisible" to the computer user. This state includes page tables and other kinds of system status, interrupt masks, system timers and time-of-day clocks, etc.

As applications become increasingly complex, some of this state is becoming subject to manipulation by certain applications in a very controlled way. Today, this "state" corresponds to tangible entities. On the horizon, some of this state will become genuinely abstract. Four such examples of abstract state are:

1. What is the parallelism in a sub-sequence of instructions? What can be done out of order, and what must be rigorously in order? What are the predictable future actions of the program at this point? This sort of information can be used by hardware (if it exists) to get more performance out of a program.

2. What level of reliability is required for this calculation? Do we need this to be right, or do we just need it to be fast? This can influence the hardware and algorithms brought to bear on an abstract semantic construct.

3. How hot (temperature) is it? Should we run certain units differently and/or shift part of the load around the processor so as to maintain a lower ambient temperature? This targets energy efficiency and long-term product reliability.

4. Will this code cause power transients as it is run? Is there a better set of controls, or other switching that can be brought to bear to keep the voltage stable?

These last categories can be thought of as ”green” instructions because they target efficient energy usage and product reliability.

For the most part, this ”green” category does not yet exist in any commercial instruction set, but as systems are becoming more power limited, these things will emerge. We will not discuss environmental instructions further; instead, we will focus on the branch instructions. We will argue that branch instructions are the most important instructions from a performance perspective, because they are the primary limiters to instruction-level parallelism [7] [8] [9] [10].

1.2 Why are Branches Important to Performance?

Computation is all about changing *state*, where state comprises the observable data within a system. A computer operates on state by fetching it (so that the computer can observe the state), and by storing it (so that the computer can alter the state). Therefore, at first blush, computing is all about fetching and storing, and there is an important duality between these two operations.

While the following illustrations sound trivial, understanding them is vital to the concept of coherency among (and within) processes.

First, imagine a computer that can only fetch, but cannot store. Such a computer can only observe state; it cannot alter it. Hence the state is static, and no ”computation” is ever manifest - which is not distinguishable from no computation ever being done. Hence, this is not really a computer.

Next, imagine a computer that can only store, but cannot fetch. (This is the dual of the above.) This computer continually changes the state, but is unable to observe any of these changes. Since the state is not observable, the state need not change - or even exist. This is also not distinguishable from no computation being done, hence this is not really a computer either.

For a computer to do anything meaningful, it must be able to observe state (fetch) and to change state (store). What makes any particular computation unique and meaningful is the sequence in which the fetches and stores are done. In the absence of branch instructions, the order of the fetches and stores would be statically predetermined for any program - hence the result of any calculation would be predetermined, and there would be no philosophical point to doing a computation.

The branch instructions create unique and dynamic orderings of fetch and store instructions, and those orderings are based on the observation of state

(via the "If <condition>" clause) as the state is changed by running processes. Ergo, computation is essentially all about resolving branches.

Then, from first principles, the maximum rate of instruction flow is gated by the ability to resolve branches. *Minsky's Conjecture* is that the maximum parallelism in a program is O(log N) where N is the number of processing elements [11]. My fancied interpretation of this (which was not necessarily Minsky's, although valid nonetheless) is that each unresolved branch splits the processing into two speculative threads, which creates a speculation tree that can be no more than log N deep, if N is the number of processing elements.

In addition to branches being the essence of computation, their frequency (static or dynamic) is dominant among all instructions [12]. For typical applications, branches comprise between one-third and one-fifth of all instructions, more-or-less independent of the specific instruction-set architecture, and of the program. In the ensuing examples, we will assume that on average, one in every four instructions is a branch. By frequency alone, it is clear that branches are crucial to performance.

In addition, branches gate performance in several other fundamental dimensions. These dimensions have to do with common performance-enhancing techniques used in microarchitecture to improve *Instruction-Level Parallelism* (ILP). ILP is usually measured (in the literature) in *Instructions Per Cycle* (IPC), although we find it much more convenient to work with its inverse: *Cycles Per Instruction* (CPI) [13].

The prevalent techniques that improve ILP are called pipelining, superscalar processing, and multithreading. Each of these is discussed below so as to make the relevance of branches clearer.

Figure 1.2 progressively illustrates the concept of pipelining. Figure 1.2.a shows the phases of instruction processing as previously discussed for a von Neumann computer. Simply, an instruction is fetched, decoded and dispatched, and executed. The machine that von Neumann built did these three things sequentially for each instruction prior to fetching a next instruction.

Pipelining is basically an assembly-line approach to instruction processing in which multiple instructions can simultaneously be in different phases of processing. For example, suppose that the instruction fetch process fetches instructions continually (without waiting for resolution signals from the other end of the processor), and that the three major blocks shown are run autonomously (with respect to each other) with queues in between them (to decouple them) as shown in Figure 1.2.b. In this figure, at least three instructions can be active simultaneously. Hence the arrangement in Figure 1.2.b has the potential of running a program three times faster (assuming that the logical phases are all equal in time).

The phases in Figure 1.2.b can be partitioned further, as shown in Figures 1.2.c and 1.2.d. The finer the partitioning, the greater the rate at which instructions can enter the pipeline, hence the greater the ILP. Note that partitioning the pipeline more finely does not alter the latency for any given instruction, but it increases the instruction throughput. (In fact, partitioning

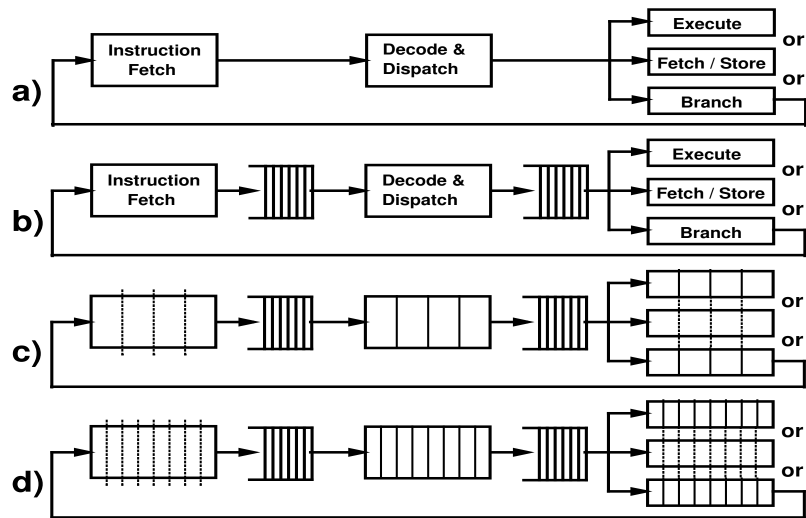

FIGURE 1.2: The Pipelining Concept: a) Phases of instruction processing; b) Decoupling the phases by the insertion of queues; c) Pipelining the phases; and d) Using a deeper pipeline structure.

the logic requires the insertion of latches between each new partition, hence the latency actually increases with the degree of pipelining.) Therefore, the pipeline in Figure 1.2.d has a longer latency, but potentially higher throughput than the pipeline in Figure 1.2.c.

Note that combinational logic (the Decode, Execute, and Branch blocks) can be finely partitioned by inserting latches between smaller numbers of logic levels. The number of resulting partitions is called the number of *pipeline stages*, or the *length* (also called the *depth*) of the pipeline. The memory blocks in the pipeline (Instruction Fetch, and Operand Fetch and Store) cannot have latches inserted into them because they are (basically) analog circuits. The pipeline "stages" in a memory path are conceptual, which is why they are denoted by dotted lines in Figures 1.2.c and 1.2.d. Hence, the memory must have enough banks (which enable concurrent accesses to be in progress) to support the flow implied by the degree of pipelining.

Given enough memory banks, and given no disruptions to the instruction flow, the pipeline can be made very deep, and can achieve very high throughput (hence, ILP). The largest inhibitor of pipeline flow is the branch instruction - recall that we had said that roughly one in four instructions is a branch. As shown in Figure 1.2.a., the branch is (nominally) resolved at the end of the pipeline, and if taken, will redirect instruction fetching. When this occurs, the entire contents of the pipeline must be invalidated, and instruction flow must be reinitiated. If there are N pipeline stages, this costs (approximately) N cycles.

In fact, all pipeline disruptions revolve around the major functional phases of instruction processing shown in Figure 1.2.a, and are temporal - independent of the degree of pipelining. Hence, the average number of cycles required to process an instruction (called "Cycles Per Instruction," or CPI) is linear in N, since each disrupting event causes a fixed number of stages to be invalidated [13].

In commercial programs, roughly two out of three branches is taken, hence there is a taken branch every six instructions. Therefore, there is a nominal delay (due to branches alone - before including any of the actual execution) of N/6 CPI for an N-stage pipeline. All high-performance processors today are pipelined. Up until this year, the trend had been to make pipelines deeper and deeper [14] to achieve high frequency (which is related to 1/N). It appears that this trend is reversing as it is becoming much more important to make processing more energy efficient [15].

Superscalar processing is another technique used to achieve ILP. "Superscalar" merely refers to processing multiple simultaneous instructions per stage in the pipeline. That is, a superscalar processor decodes (and executes) at least two instructions per cycle. Many machines try to do four per cycle, and there are advocates of as many as sixteen or even thirty-two instructions per cycle [16] [17]. With an N-stage pipeline running M-way superscalar, there can be as many as MN active instructions in the processor.

What superscalar processing also does is that it puts interlocked events

(like branches and dependencies) closer together in the pipeline so that their resulting delays appear to be longer. And with a branch occurring every four instructions, a four-way superscalar processor should expect to encounter a branch instruction every cycle. That is, using the model described by Figure 1.2, a four-way superscalar processor should expect an N-cycle delay after every decode cycle.

A sixteen-way superscalar processor should expect to see four branches per cycle. This is philosophically disturbing, since the relevance of each of these branches (i.e., their very presence) depends on the outcomes of the predecessor branches - with which it is coincident in time.

It should be obvious that branch instructions severely inhibit the potential benefits of superscalar processing.

The last pervasive ILP technique is multithreading. In multithreaded processors, multiple independent instruction streams are processed concurrently [18]. There are many variations on the specifics of how this is done, but the general idea is to increase the utilization of the processor (especially given the ample delays present in any stream), and the aggregate processing rate of instructions, although each stream may run slower. Multithreading is mentioned in this context because it is the dual of superscalar - its general effect is to spread out the distance (in pipeline stages) between interlocked events.

In pipelined processors, instruction fetching can be made autonomous with respect to the rest of the machine. That is, instruction fetching can be made to drive itself, and to stage what is fetched into instruction buffers - which are (autonomously) consumed by the instruction decoder(s). This is sometimes called "instruction prefetching." (Note that the word "prefetching" is used in many different contexts in a computer. Be sure to understand what it does and does not connote in each context.) As we will see later, enabling the autonomy of instruction prefetching is a precondition of achieving high performance [19].

If there were no branch instructions, then instruction prefetching would be exceedingly simple. It would merely involve fetching sequential blocks of instructions starting at the beginning of a program. This can be done by loading a "Prefetch Address" register with a starting address, and repeatedly fetching blocks of instructions, putting them into an instruction buffer, and incrementing the Prefetch Address register. This is shown in Figure 1.3.a. (A simple interlock is also needed that stops prefetching when the instruction buffer is full, and resumes prefetching when it is not.)

Because there are branch instructions, this autonomy breaks down. When any instruction in the instruction buffer is found to be a taken branch instruction, then generally: 1) the instructions that were prefetched into the buffer following this instruction are useless, and 2) the instructions that are actually needed (the branch target instruction and its successors) have not been prefetched, and are not available for decoding following the branch. As already discussed, this causes a penalty proportional to the length of the pipeline. Thus, branches inhibit the ability to prefetch instructions.

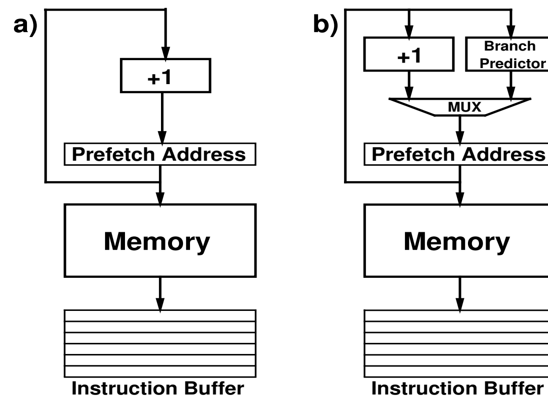

FIGURE 1.3: Instruction Prefetching: a) Sequential prefetching in the absence of branches; and b) Augmenting the prefetching with a branch predictor.

The key to achieving high ILP is to preserve the autonomy of instruction prefetching. This requires that the branch instructions in the running program be anticipated, and be predicted correctly so that instruction prefetching can fetch the correct dynamic sequence of instructions to be executed [20] [21]. Figure 1.3.b shows the prefetching process augmented with a branch predictor which attempts to redirect the prefetching as appropriate.

In the remainder of this chapter, we will explore the dimensions and consequences of various branch prediction methods.

1.3 Three Dimensions of a Branch Instruction, and When to Predict Branches

As explained in the introduction, a branch instruction causes the instruction flow to be diverted to a non-sequential address in the program depending on the program and the system state. There are three dimensions to this diversion of flow:

1. Does the diversion occur, i.e., is the branch taken or not?

2. If the branch is taken, where does it go? I.e., what is the target address?

3. If the branch is taken, what is the point of departure? I.e., what should be the last instruction executed prior to taking the branch?

Most of the literature focuses on the first dimension - whether the branch is taken. In real machines, and depending on the *Instruction-Set Architecture* (ISA), it may be necessary to deal with the second dimension (target address) as well. This depends on *when* (in the pipeline flow) the branch is to be predicted. The predictor in Figure 1.3.b implicitly handles this dimension, since it shows the branch being predicted prior to its actually having been seen. (The branch is predicted at the same time that it is being prefetched.)

The third dimension is seldom discussed, since in all machines today, the point of departure is the branch instruction itself. Historically, there have been variations on branches in which the branch instruction is not the last instruction in the sequential sequence in which it acts. For example, the 801 processor - the first *Reduced Instruction Set Computer* (RISC) - has a "Delayed Branch" instruction [22] [12].

The delayed branch instruction causes instruction flow to be diverted after the instruction that sequentially follows the delayed branch instruction. I.e., if the delayed branch is taken, then the next-sequential instruction (following the delayed branch) is executed prior to diverting the flow to the branch target instruction.

The 801 has a two-stage pipeline: Instruction Decode; and Execute. If an instruction is found that can be migrated (by the compiler) past the branch instruction, the delayed branch completely hides the delay associated with diverting the instruction fetching. What would have been an empty stage in the pipeline (following a normal branch) is filled by the execution of the migrated instruction as the branch target instruction is fetched.

In practice, the delayed branch has limited application, since statistically, most branch groups (the set of instructions following a branch and ending with the very next branch) are very short, and the ability to migrate instructions is limited. While this still provides some benefit in a two-stage pipeline, the delayed branch has fallen out of use as pipelines have become longer.

Before discussing *how* to predict branches, it is important to understand *when* to predict them. Consider the pipeline abstraction shown in Figure 1.4.a. This is an RS-style pipeline, which is typical of IBM 360 machines [23]. In this pipeline, instructions are fetched sequentially, and put into an instruction buffer. Then they are decoded, and an address is generated by adding the contents of general-purpose registers. For a branch instruction, the address generated is a branch-target address which is sent back to the beginning of the pipeline. For a load or store instruction, the address generated is an operand address, and an operand fetch is done. Next, the instruction is executed. For branch instructions, execution is when the branch outcome (taken or not taken) is determined.

In this pipeline, the decode stage is the first stage to actually "see" the branch instruction. Prior to decode, instructions are merely unidentified data being moved through buffers. When the decoder first "sees" a branch instruction, what should it do? These are the various options and their consequences:

1. Assume that the branch is not taken. Leave the target address in an address buffer, but keep fetching and decoding down the sequential path. If the branch is not taken, no cycles are lost. If the branch is taken, this is discovered at execution time, and the target instruction can be fetched subsequently, using the buffered target address. The penalty for being wrong is the entire latency of the pipeline.

2. Assume that the branch is taken. Save the sequential instructions that have already been fetched into the instruction buffer, but redirect fetching to the branch target address. If the branch is taken, then redirecting is the right thing to do, and the cost of the branch is the inherent latency of instruction fetch, decode, and address generation (i.e., the beginning of the pipeline). If the branch is not taken (which is discovered in execution), then it was a mistake to redirect the stream, and the prior sequential stream (which was saved in the instruction buffer) can resume decoding. The cost of being wrong is the latency of the pipeline not including the instruction fetch.

3. Do one of the above with *hedge fetching*. Specifically, follow one of the

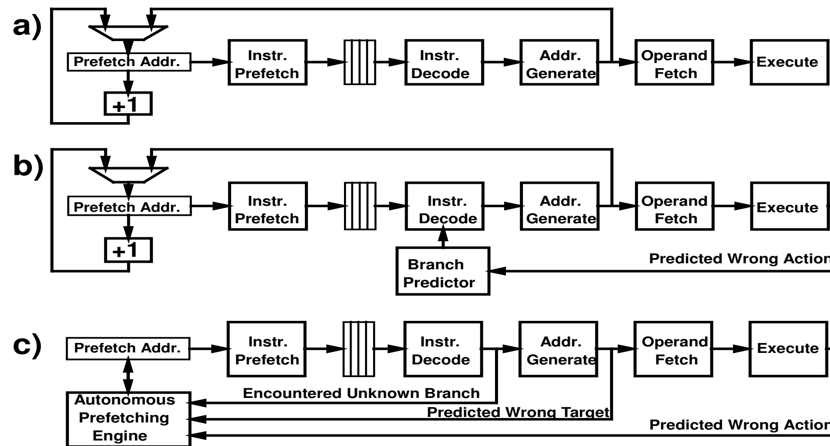

FIGURE 1.4: An RS-Style Pipeline with Various Approaches to Branch Prediction: a) Without branch prediction; b) With decode-time branch prediction; and c) With prefetch-time branch prediction.

paths, but do some instruction-fetching down the other path to enable decoding to resume immediately if the guess is wrong [24] [25] [26] [27] [28] [29] [30]. In this case, we lose less when the guess is wrong, but we risk losing some cycles (due to sharing the fetch-bandwidth between the two paths) when the guess is right.

4. Fetch and execute down both paths [31] [32]. Again, we lose less when the guess is wrong, but we also lose performance unnecessarily when the guess is right.

5. Stop decoding. This is the simplest thing to do (because it does not require pipeline state to be subsequently invalidated). In this strategy, we concede a startup penalty (essentially, a delay until the branch executes) to every branch instruction, but the downside (the penalty for being wrong) can be smaller, and the design is simpler, since there cannot be wrong guesses.

Which of these strategies is best? It depends on:

1. The various penalties for being right, wrong, or partially right/wrong as described (e.g., for hedge fetching, following both paths, or stopping), and

2. The probability of being right or wrong in each case.

The penalties depend on the specific details of the pipeline - with larger penalties for deeper pipelines [33]. The interesting variable is that the probability of being right or wrong is not the same for all branches [34] [35] [36] [37] [38]. Many real machines have used combinations of all of the above strategies depending on the individual branch probabilities [39].

Figure 1.4.b shows a branch predictor that works in conjunction with the decoder. In this case, the "when" is at decode-time. These predictors can be very simple. A big advantage of predicting at decode time is that we know that the instruction is a branch because the decoding hardware is looking at it [40] [41] [42]. (This will not be the case for certain other predictors.) Also, generally these predictors do not have to predict a target address, since the processor will calculate the address in concert with the prediction.

All that these predictors need to do is to examine the branch instructions and guess whether they are taken. Optionally in addition, the predictors can make estimates as to the confidence in their guess for each given branch, and select one of the five courses of action listed above (calculated as a function of the penalties and the certainties of the guesses). The processor need not follow the same course of action for every predicted branch. After the branch instructions execute, their outcomes can be used to update information in the predictor.

Note that when predicting at decode time, there is always a penalty for taken branches, even when the prediction is right. This is because the target

instruction has not yet been prefetched. The predictor does not know that a branch instruction is in the sequence until it is seen by the decoder. For this configuration, this is the earliest point at which there can be an inkling that instruction fetching needs to be redirected.

Therefore, when there is a taken branch, there will always be a penalty at least as large as the time required to fetch an instruction. Since the latency of a fixed-size SRAM array (where the instructions are stored) is fixed, as the pipeline granularity increases, the number of cycles for a fetch (hence the penalty for a taken branch) grows. As we said in the previous section, typical programs have taken branches every six instructions. Thus, branch delays fundamentally remain significant for decode-time predictors, even if the predictors are very accurate.

To completely eliminate branch penalty requires that branch instructions be anticipated (i.e., before they are "seen" by decoding logic), and that branch target instructions be prefetched early enough to enable a seamless decoding of the instruction stream. Specifically, to eliminate all penalty requires that branches be predicted at the time that they are prefetched [20] [21]. This arrangement is shown in Figure 1.4.c.

Predicting branches at prefetch-time is much more difficult to do for two reasons. First, the instruction stream is not yet visible to the hardware, so there is no certainty that branches do or do not exist in the stream being prefetched. The prefetch mechanism must intuit their presence (or lack thereof) blindly. Second, it is not sufficient to predict the branch action (taken or not taken). To eliminate all penalty requires that the target addresses be predicted for branches that are taken. The target address needs to be provided by the predictor so that prefetching can be redirected immediately, before the branch has actually been seen.

The increased complexity and difficulty of predicting at prefetch-time is evident in Figure 1.4.c. Note that there are now three ways in which the predictor can be wrong (shown as three feedback paths to the predictor in Figure 1.4.c) as opposed to just one way for decode-time predictors (shown as a single feedback path in Figure 1.4.b). These three ways are:

1. The predictor can be wrong about the presence of a branch, which will be discovered at decode time. That is, the predictor will not be aware of a branch that is in the stream, and the error will be discovered when the decoder encounters a branch instruction. Or, the predictor could indicate the presence of a taken branch in the stream, and the decoder could find that the instruction that was predicted to be a branch is not a branch instruction. (This will happen because of hashing collisions if full address tags are not used, and because of page overlays if the predictor uses virtual addresses.)

2. The predictor can be wrong about the branch target address for correctly predicted taken branches. This will be discovered at address-generation time if the processor generates a target address that is different than the

predicted target address [43]. When this happens, it is usually because the branch instruction changes its target - as for a subroutine return. It can also be caused by hashing collisions and page overlays (as in the previous case) when the collision or overlay happens to have a taken branch in the same location as what it replaces.

3. The predictor can be wrong about whether the branch was taken - as for a decode-time predictor.

Thus for prefetch-time predictors, the number of ways of being wrong, and the number of recovery points further fragment the various penalties involved. Note that these predictors can be wrong in more than one way, which can cause more complexity in the recovery processes. For example, the predictor could predict a branch to be taken, but predict the wrong target address. The processor can redirect the fetching following address-generation, and shortly thereafter (when the branch executes) discover that the branch isn't taken, which must trigger a second recovery action for the same branch.

The various complexities of these elements of prediction are discussed in the following sections.

1.4 How to Predict Whether a Branch is Taken

The very first predictors made "static guesses" based on the branch opcode, the condition specified, the registers used in the address calculation, and other static state [44] [45]. For example, unconditional branches are certainly taken. Looping branches (like "Branch on Count") are usually taken, and can be predicted as such with high certainty. These predictions were eventually augmented by static predictions made by compilers [46] [47] [48] [49]. This required more branch opcodes (to denote "likely taken" or "likely not-taken" branches). The compiler's guess was used in conjunction with a guess made by the hardware to determine the overall best guess and pipeline strategy to use for each branch instruction.

Not long thereafter, branches were predicted using dynamic information of various sorts. Specifically, the past execution of branches was used in several different ways to predict the future actions of branches. The basic variations are:

1. Any branch is assumed to behave (be taken or not) in accordance with the behaviors of other recently executed branches, i.e., a "temporal average behavior" is assumed for branches.

2. Each specific branch (identified by its address) is assumed to do (to be taken or not) the same thing that it did the last time.

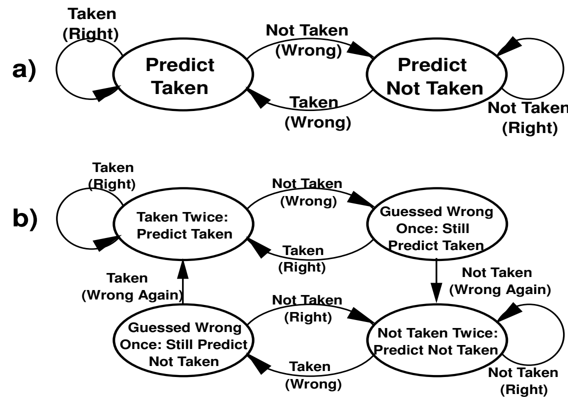

FIGURE 1.5: Simple State-Machines for Branch Prediction: a) A 1-bit state-machine; and b) A 2-bit state-machine.

3. Each specific branch (identified by its address) is assumed to be taken or not based on some pattern of its own behavior exhibited in the past. This is an evolution of variation 2.

4. Each specific branch (identified by its address) is assumed to be based on its past actions, and on the actions of other recently executed branches. This is a mixture of all of the above variations.

The first variation is the simplest, since it does not require much state. It merely requires a few bits to record the actions of the last several branches via a state machine that generates predictions. The other three variations involve predicting actions for specific (by address) branches, which require tables to store past actions by branch address. The last variation requires both of these mechanisms.

Figure 1.5 shows state-transition diagrams for two very simple state-machines that track recent branch behavior in a program. The first (Figure 1.5.a.), is just a saturating one-bit counter. It records the action of the last branch ("Taken" or "Not Taken"), and it predicts that the next branch will do the same thing. The second (Figure 1.5.b) is only slightly more complex, and requires a two-bit up/down counter that saturates both high and low.

The state-transition diagrams for both figures are symmetric. Each state records the past history of branches (e.g., "Taken Twice"), and what the next

prediction will be (e.g., "Predict Taken"). The state transitions (shown as arcs) are labeled with the next branch outcomes ("Taken" or "Not Taken") and whether the prediction (from the originating state) is "Right" or "Wrong."

What the state machine in Figure 1.5.b does is to stick with a particular guess ("Taken" or "Not Taken") until it has been wrong twice in succession. This particular predictor first appeared in a CDC patent [50]. The CDC environment was predominantly a scientific computing environment in which many of the branches are used for loops. The basic idea is to not change the "Taken" prediction for a looping branch when it first falls through [51] [52] [53].

In addition, in that time period, flip-flops were very precious. Distinct branch instructions were allowed to alias to the same state-machine so as to provide an "average group behavior" of the branches in a program. As mentioned, recording state transitions for individual branches requires tables (i.e., a record of state for each branch instruction).

There were a number of papers in the literature at that time that made obvious extensions to this (three-bits, four-bits, etc.). Instead of counting (which has intuitive basis), many of these devolved into what is essentially profiling [54] [55] [56] [57] [58] [59] [60] [61] [62] [63]. For example, if the state comprises ten bits, then there are 1024 possible sequences of branch actions, and the appropriate prediction for each sequence can be predetermined by profiling a trace tape. (E.g., when the sequence "1001101100" occurs in the trace, measure whether "1" or "0" occurs most often as the next branch action in the trace, and use this as the prediction.)

Each such prediction defies intuitive rationalization - it is just based on measurement. In some of these studies, the prediction was run (again) on the very same tape that was used to generate the profiles. While (quite unsurprisingly) the "guesses" get better as the number of bits is increased because some of the unusual patterns essentially identify specific branch occurrences (which certainly happens when the sequence only occurs once) in the profiling trace. What was not shown in these studies was whether the profiling statistics result in robust prediction. I.e., does it work well on a different execution trace?

Therefore going beyond two or three bits of state is probably not that useful in this particular context. More recently, it has become useful in a different context to be discussed later. There are even more exotic finite-state machines (using more bits) - including neural networks and Fourier analysis - that have been used to predict branch outcomes [64] [65] [66].

Getting high predictive accuracy requires predicting the branches individually, and relying on the behavior of each individual branch rather than a group average. This is easy to understand, since some of the individual branches in a program tend to be taken almost always (e.g., looping branches), or almost never (e.g., exception handling), and the "group average" filters out this information.

Note that the worst possible predictive accuracy is 50%, because 50% in-

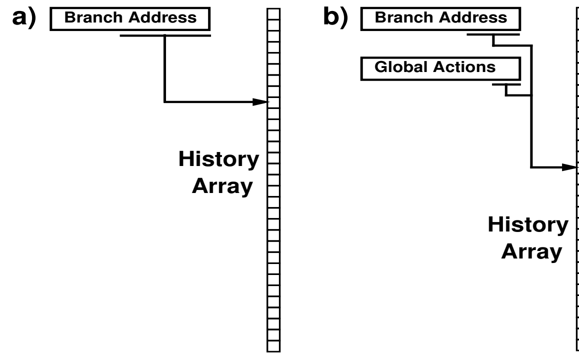

FIGURE 1.6: Decode-Time Branch Predictors: a) Using the branch address; and b) Using the branch address together with the last global actions.

dicates that there is no correlation between the prediction and the branch action. If the accuracy is <50%, and this is monitored, the predictor can be made to work at >50% accuracy by continuing to use the same algorithm, but inverting the prediction. Therefore, 50% accuracy should be a lower bound.

Maintaining state for each individual branch requires a table - with an entry for each branch. Since the state stored for a branch is small (e.g., one bit), it is inefficient to make the table rigorously discriminate addresses by storing tags (the addresses). In practice, the table is implemented by performing a hash function on the branch address, and using the hash to address a bit vector. The most straightforward hash (requiring no circuitry) is truncation - the elimination of high-order bits.

Figure 1.6.a shows a simple predictor that is used in many real machines. This method was first disclosed as a "Decode History Table" (called a "DHT" in parts of the industry) because it worked at decode time [67] [68] [69], but has become to be known as a "Branch History Table" (BHT) in most of the literature today. In Figure 1.6.a, the low-order log(n) bits of the branch address are used to access the n-entry table that contains state information for the branches. Branches whose addresses share these low-order bits will hash to the same entry.

The smaller the table, the more hashing collisions will result, and the more

potential for error because of this. Therefore, when there are a fixed number of bits to be used for prediction, it is always useful to assess whether the predictor benefits more from making the state more complex, or from having more entries. (E.g., would you rather have n one-bit predictors, or n/2 two-bit predictors?) In commercial code where the working-sets are large, it is almost always better to have more entries [70] [71].

The branch address is a *discriminator* which uniquely identifies a particular branch instruction. A hash of that address is also a discriminator - with some aliasing. As the hashing becomes cruder, the discriminator does a poorer job of uniquely identifying branches [72] [73] [74] [75] [76]. (In the first predictor discussed, there was no hash function - all branches aliased to the same state machine.)

Recall that when we were discussing profiling using n-bit sequences, we had said that long unique strings were (essentially) uniquely identifying particular branch occurrences on a profiling tape. In this sense, a string of branch outcomes is also a discriminator for the next branch in the sequence, since it correlates to the path that was traversed (through the code) in reaching this branch [59]. The longer the discriminating string, the better it does at discriminating.

The fourth prevalent prediction technique (described in the opening paragraphs of this section) uses both discriminator types in tandem. That is, for each particular branch address (the first discriminator), the unique sequences of branch outcomes preceding this branch (the second discriminator) are recorded, and a prediction is made based on past history for this branch when it was preceded by the same sequence of global branch outcomes [77] [78] [79] [80] [81] [82] [83].

Conceptually, the rationale behind this is that some of the unique branches (by address) are not very predictable (e.g., they have accuracies closer to 50% than to 100%), although when those branches are reached by taking particular paths through the program, they are relatively predictable for each particular path. The first discriminator (branch address) identifies the branch, and the second discriminator (current sequence of global branch actions) correlates to the path being followed. When used in conjunction, the two discriminators achieve a higher accuracy than either could by itself.

In principle, this could be implemented as a table of tables involving a sequence of two lookups. In real machines, the problem with cascades of lookups is that they take too much time. A branch predictor needs to provide a prediction in the cycle in which it must predict [84]. It does the machine little good to have a correct prediction that is later than this.

Thus, realizations of this kind of predictor are best made by hashing both discriminators together into a single lookup as shown in Figure 1.6.b. In this figure, the "Global Branch Actions" register records the actions of the last n branches as a sequence of ones and zeros. Whenever a branch executes, its action (taken or not) is shifted into one end of the register as a zero or one. The lookup into the history array is done by merging the low-order bits of the

branch address with a portion of the Global Branch Actions register.

A caveat with algorithms that use the "last branch actions" in real machines is that the "last" actions might not yet have occurred at the time that the prediction needs to be done. For example, if there is a branch every four instructions, and the pipeline between decode and execution is longer than this, then at decode-time for each branch, its immediately preceding branch can not have executed yet, hence the "last branch outcome" is unknown. A state register that tracks "last outcomes" (as in Figure 1.6.b) will be out of phase by one or more branches.

Therefore, a predictor that is designed based on studying statistics from an execution trace might not behave as well as anticipated in a real machine. In this case, the trace statistics will be based on the actual "last actions" of the branches, but the real machine will be using a different (phase-shifted) set of "last actions." One way of dealing with this is to use the "Global Branch Predictions" (not shown, but obvious to conceive of) instead of the "Global Branch Actions." This will align the phases correctly, although some of the bits may be wrong - which is likely academic.

Finally, there are compound (hybrid) prediction schemes. We had previously mentioned that hardware could monitor the predictive accuracy for a given branch, and change the prediction if appropriate. In that example, we had said that if the accuracy fell below 50%, we would invert the predictions from that point on.

This basic idea can be used in a more general way. Specifically, we can build multiple independent prediction mechanisms, and dynamically monitor the accuracies of each of them for each branch. We can achieve the best of all of them by using the monitored data to dynamically select the best predictor for each individual branch [35] [85] [86] [87] [88] [37] [89].

In a hybrid scenario, some of the branches are predicted by using one-bit counters, while others are predicted using two-bit counters, and others using global branch information. While this is not the most efficient use of bits, it achieves the highest accuracy - assuming that the working-set is adequately contained.

1.5 Predicting Branch Target Addresses

Branch target addresses are predicted at prefetch time so as to facilitate the redirection of instruction prefetching so as to eliminate all delay for correctly predicted taken branches. As mentioned previously, this drives additional complexity.

The simple part of target prediction is that the future target address of a branch is usually whatever the historical target address was. This must be

recorded for each branch instruction.

In addition to whatever state is required to predict branch action, we now need to store a target address (e.g., thirty-two bits for a thirty-two-bit address space). Therefore, the amount of state for a branch prediction entry is much larger in this environment than what we discussed in the previous section. The point is that branch-target prediction done at prefetch-time is a much more costly proposition than branch-action prediction done at decode-time, although doing it well is essential to achieving high performance [20] [21].

Since we predict target addresses to facilitate the redirection of instruction prefetching, it is necessary to understand what we mean by ”instruction prefetching.” In the first place, the quanta being fetched are not instructions. They are generally some larger fixed data-width that can be accommodated by the busses and buffers, and that provide ample instruction prefetching bandwidth. In many machines, ”instruction prefetching” is the act of prefetching quadwords (sixteen bytes) or double-quadwords (thirty-two bytes) aligned on integral boundaries (sixteen-byte boundaries for quadwords, and thirty-two-byte boundaries for double-quadwords) [90] [91] [92] [93]. For the remainder of this discussion, we will use quadwords as the unit of fetching, although the discussion easily generalizes to other widths.

The quadwords that are fetched contain multiple instructions, one or more of which can be a branch. For example, if the length of an instruction is a word (four bytes), then a quadword contains four instructions, and (as many as) four of them can be branches. For variable-length instruction-sets, a quadword need not contain an integer number of instructions, and some instructions (including branches) will span quadword boundaries.

Therefore, two halves of a single branch instruction can reside in consecutive quadwords. A taken branch can leave the middle of a quadword, and can branch into the middle or end of a different quadword. Or a branch can branch into the quadword containing itself. These realities drive further complexity.

Basically, a prefetch-time predictor is a table of entries, where each entry corresponds to a specific branch instruction, and contains the address of the branch-target instruction and some other state information. The table must be organized to be compatible with the instruction prefetch width (e.g., a quadword) so that it can be searched on behalf of each instruction prefetch. The other state information in an entry includes state to predict whether the branch is taken, state having to do with the organization of the table itself, and state that enables the hit-logic to find the right branches within the quadwords.

Since we are now putting more at stake by redirecting instruction prefetching, it is more important to discriminate branches correctly. Therefore instead of merely hashing entries, branch tags are generally used. (But since the entry already needs to hold a full target address, the overhead for a tag is relatively less than it was in the previous section.) Since a quadword can contain multiple branches, the table should be set-associative so as to avoid thrashing between branches contained in the same quadword.

When originally disclosed, this was called a "Branch History Table" (BHT) [94] [95] [96] [97] [91] [98] [99] [100] [101], but has come to be called a "Branch Target Buffer" (BTB) by many [102] [103] [104]. Recall that the recent literature uses "Branch History Table" (BHT) to denote the table of prediction bits described in the previous section - which was originally called a Decode History Table (DHT). This can sometimes lead to confusion in discussions, so one must always take pains to make the context clear.

A BTB is sketched in Figure 1.7.a, and the fields within a BTB entry are depicted in Figure 1.7.b. The fields shown are:

1. Valid - A bit that indicates that this is a real entry, and not uninitiated state.

2. Action - The state bits that are used to predict whether the branch is taken. This corresponds to the state described in the previous section. (In some implementations, the Valid bit is interpreted directly as an indication that the branch is taken, and there is no explicit Action field.)

3. Tag - The high-order address bits of the branch instruction (used by the hit-logic to match an entry to the correct quadword).

4. Offset - The low-order address bits of the branch instruction (the byte address within the quadword) that is used by the hit-logic to locate the branch within the quadword.

5. Target Address - The address of the branch-target instruction.

6. Length - A bit used to indicate whether the branch instruction spills into the next quadword. If it does, then the next-sequential quadword needs to be fetched prior to redirecting fetching to the target address, or the processor will not be able to decode the branch instruction.

7. Special T - A field to indicate how the Target Address field is to interpreted for this entry. In some implementations, the penalty (in complexity) can be severe for predicting the target address incorrectly. Special T can simply be used to alert the predictor not to trust the Target Address field, and to tell the processor to stop decoding when it encounters this branch, pending verification of the target address by the address-generation stage in the pipeline. Or it can be used more elaborately to create hints about branches that change their target address. This is discussed in the next section.

8. Check bits - Parity or other check bits to detect array errors - these would generally be in any real product.

The BTB works as follows. The Prefetch Address (shown in Figures 1.3 and 1.4) is used to search the BTB at the same time that the corresponding quadword of instructions is prefetched. In particular, the BTB can be the

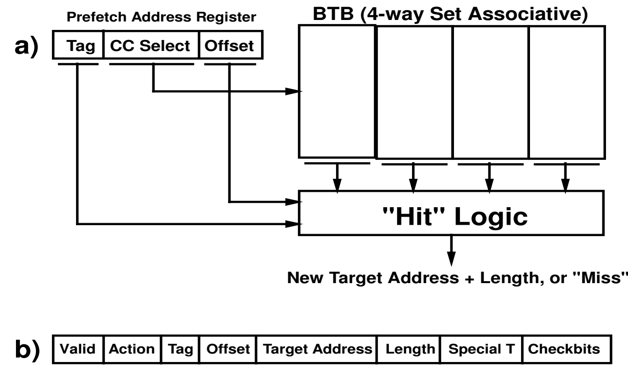

FIGURE 1.7: The Branch Target Buffer (nee, "Branch History Table"): a) The BTB structure and lookup; and b) The field within a BTB entry.

"Autonomous Prefetch Engine" shown in Figure 1.4.c. Referring now to Figure 1.7.a, the low order four bits of the prefetch address (the byte within quadword) are sent to the hit-logic. The next set of bits (the log of the number of rows in the BTB) is used to select a congruence class within the BTB. In this case, the BTB is shown as four-way set-associative table, so four candidate entries are read out. The remaining high-order bits of the prefetch address are sent to the hit-logic for a tag comparison with each of the four candidate entries.

The hit logic determines the presence of a branch within a quadword by doing the following four evaluations on the entries within the selected congruence class of the BTB:

1. If the entry does not have the Valid bit set, then it is removed from consideration.
2. If the Tag field in the entry does not match the high-order bits of the prefetch address (which comprise the Tag portion of the address shown in Figure 1.7.a), then the entry denotes a branch in a different quadword, and it is removed from consideration.
3. If the Offset field in the entry is less than the low order bits (byte within quadword) of the prefetch address (see Figure 1.7.a), then the

entry denotes a branch that resides in a lower part of the quadword than the point at which instruction flow enters the quadword, and it is removed from consideration. For example, if flow enters this quadword at the end of the quadword (due to a branch), then the processor will not encounter the branches that are at the beginning of the quadword.

4. If more than one entry passes the first three tests, then we choose the first entry (the one with the smallest Offset field) that will be predicted as taken (by the Action field). The taken branch with the smallest Offset (greater than the Offset in the Prefetch Address Register - as determined in Step 3) is the first taken branch that will be encountered in the flow. This is the "hit" that is sought.

If a "hit" is encountered, then instruction prefetching will be redirected to the address specified in the Target Address field of the entry that caused the hit. Note that if the corresponding Length field indicates that the branch spills into the next quadword, then the next-sequential quadword is prefetched prior to redirecting the prefetching to the target address. (Care must be taken to suppress BTB results during the prefetching of the next-sequential quadword in this case.)

If no "hit" is encountered, then prefetching (and BTB searching) continues with the next-sequential quadword.

We have stated that the BTB search must be done on the same quantum boundaries as the instruction prefetching. This is not an absolute requirement, but the BTB should (at least) keep up with the prefetching, or the prefetching cannot be redirected in a timely manner. It is equally acceptable to organize the BTB on double-quadwords if the instruction fetching is on quadwords. This may actually be preferable, because it allows sufficient BTB bandwidth to enable timely updates. This is discussed later.

As mentioned in a previous section, we can predict the target incorrectly for three reasons:

1. If full address-tags are not used, we can hit on a coincident entry from an aliased quadword which generally will have a different target address.

2. Since we are not doing address translation prior to the BTB search, we are searching with a virtual address. When the operating system performs page overlays, we may (coincidentally) have an entry for a branch at a location that also contains a branch in the new page, but the target address will likely be different.

3. Some branches really do change their target addresses. This is discussed in the next section.

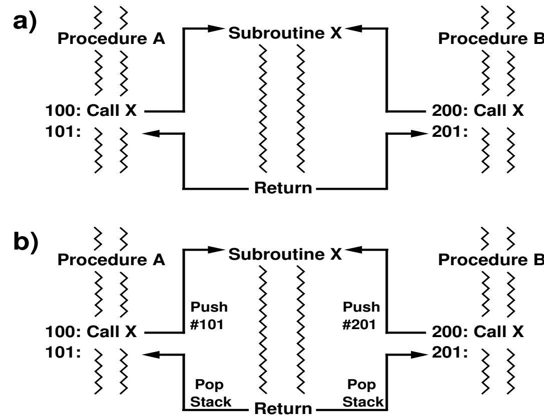

FIGURE 1.8: The Subroutine Call and Return: a) The call-and-return structure: and b) The same structure annotated with stack operations.

1.6 The Subroutine Return - A Branch That Changes Its Target Address

Most branches in a program always branch to the same target instruction. A canonical exception to this is the branch used for returning from a subroutine. A subroutine is a generic procedure that can be called from different places in programs. When a subroutine is called from procedure A, upon completion it should return to procedure A (typically, at the instruction immediately following the calling instruction). When it is called from procedure B, it should return to procedure B. Figure 1.8.a shows this structure.

A problem arises in prefetch-time predictors (BTBs) that treat the subroutine return as any other branch. While some *Instruction-Set Architectures* (ISAs) have explicit Call and Return instructions - and the subroutine return can be recognized by the decoding hardware (and tagged as such in the Special T field of the BTB), other ISAs implement these primitives with standard branch instructions, and the decoding hardware is oblivious to the call-return primitive.

When the BTB is unaware that a particular branch is a subroutine return branch, it always predicts the target address to be whatever it was last time

(the historical return point). When a subroutine last returns to procedure A, it remembers the return point in A. When the subroutine is next called from procedure B, the BTB predicts the returning branch as returning to procedure A. This is an incorrect prediction, since we know that the correct return point is in B.

In this section, we explain how the branch predictor is made sensitive to the call-return primitive so as to correctly predict subroutine return branches at prefetch time [105] [106] [107] [90] [108] [109] [110] [111] [112]. First, so as to make the explanation as simple as possible, we explain how to do this in the context of an ISA having explicit Call and Return instructions, and we augment the prefetch-time predictor to include a stack.

In this scenario, whenever the decoder encounters a Call instruction, the address of the instruction that sequentially follows the Call instruction (i.e., the return point) is pushed onto the stack in the predictor hardware. When a Return instruction is first encountered, it is unknown to the BTB, and it is mispredicted. When a BTB entry is created for the Return instruction, the "Special T" field is set to indicate that this is a Return instruction.

Now in future predictions, whenever the BTB encounters a hit in which the "Special T" field is set to indicate a subroutine return, the Target Address from the BTB entry is not used. Instead, the address on the top of the stack is used as the return point. Figure 1.8.b is annotated with the corresponding stack actions to illustrate this behavior.

Now, consider ISAs that do not have explicit Call or Return instructions. The question is how to recognize that a particular branch instruction is implementing a subroutine return? Although ISAs may not have an explicit Call instruction, they generally have a particular kind of branch that tends to be used for subroutine calls.

For example, the Branch And Link (BAL) is a branch that stores the address of its successor instruction (the return point) into a general purpose register as it branches. While being used for subroutine calls, it can also be used for other things (e.g., it performs the "load context" primitive, which enables a program to know its current location in storage).

If there is no explicit Return instruction, then we need to use clues that a branch instruction is a subroutine return instruction. There are a few such clues:

1. If the BTB predicts the target address incorrectly, the corresponding branch could be a subroutine return.

2. If the target address of a branch is specified as being the contents of the same register used by a preceding BAL instruction, then this branch could be a subroutine return.

3. If the target address of the branch happens to be an address that immediately follows the address in which a BAL is stored (which should be on top of the call-stack), then the branch could be a subroutine return.

A predictor can be made sensitive to one or all of these things. The first clue is already provided for us in the existing structure - we just need to observe that the BTB makes a wrong target prediction. The second clue requires more hardware to keep track of specific register usage. The third clue is the most interesting, because it allows us to use a trick that is an artifact of how the BTB is organized.

Recall that instruction prefetching is done on a quadword or double-quadword basis, and that the BTB is organized around this width (or around an even larger width). When a branch target is mispredicted, and the processor recovers and redirects prefetching to the correct target address (clue 1), the first thing that is done is that the BTB is searched to see if there are any branches at the new target address.

While the hit logic is looking for branches at or after the new target address (the entry point into the target quadword), it sees entries for *all* branches within the target quadword, i.e., it sees the branches that are located before the entry point into the quadword (unless the entry point is 0 - the beginning of the quadword).

Within this context, we can do the following things:

1. Whenever we encounter a branch that could be a subroutine call (e.g., a BAL), in addition to pushing the next instruction address onto the stack, when we create the BTB entry for the branch, we indicate "Probable Calling Branch" in the Special T field for the entry.

2. Whenever we have a target address misprediction, and redirect prefetching to a new address, then when we do the BTB lookup in the new quadword, we notice whether there is a "Probable Calling Branch" immediately prior to our entry point into the quadword. If there is, we assume that the branch that was just mispredicted is a subroutine return, and we indicate "Subroutine Return" in its Special T field.

3. Whenever we get a BTB hit with "Subroutine Return" indicated in the Special T field, we use the address on the top of the stack instead of the address in the Target Address field of the BTB.

Finally, the algorithm described above can be implemented directly in the BTB without the use of a stack [90]. This requires a new Special T state that indicates that an entry is a subroutine entry point. This works as follows:

1. Whenever we encounter a branch that could be a subroutine call (e.g., a BAL), then when we create the BTB entry for the branch, we indicate "Probable Calling Branch" in the Special T field for the entry.

2. Whenever we have a target address misprediction, and we redirect prefetching to a new address, then when we do the BTB lookup in the new quadword, we notice whether there is a "Probable Calling Branch" immediately prior to the entry point into the quadword. If there is, we

assume that the branch that was just mispredicted is a subroutine return, and we indicate "Subroutine Return" in its Special T field.

(a) In addition, the Target Address field of the Probable Calling Branch in the target quadword specifies the address of the subroutine entry point (i.e., the beginning of the subroutine), and we know the address of the returning branch (it is the address of the branch that was just mispredicted - for which we are currently doing the BTB search).

(b) Create a new entry in the BTB at the subroutine entry point. We set its Special T field to indicate "Subroutine Entry Point," and we set its Target Address field equal to the address of the returning branch. I.e., a Subroutine Entry Point entry does not indicate the presence of a branch instruction. It is merely a pointer to the returning branch - the branch for which we are doing the search.

3. Whenever we get a BTB hit with "Subroutine Entry Point" indicated in the Special T field, we do not redirect prefetching, since this is not a branch. However, we do a BTB update. Specifically, we take the Target Address field (which points to the returning branch), and we update the entry corresponding to the returning branch. Specifically, we change its Target Address field to the address of the instruction that sequentially succeeds the branch that generated the current BTB hit, i.e., we set it to point to the sequential successor of the calling branch. (Remember that the branch that generated this hit is the calling branch.)

4. Whenever we get a BTB hit with "Subroutine Return" indicated in the Special T field, its Target Address should be correct, since it was updated to the correct return point in Step 3.

This algorithm will work for subroutines having one or more entry points, and a single return point. If there are multiple return points, this algorithm can have problems unless each entry point correlates to a specific return point.

1.7 Putting it All Together

A number of points are made in this section about the operating environment (the context) of a predictor relative to the size of the instruction working-set, and of the statistical accuracy of predicting branches. Considering these things motivates (or not) the use of even more exotic mechanisms, and it relates branch prediction to instruction prefetching.

We have said that (roughly) one in four instructions is a branch. While this was a statement about the dynamic frequency of branches, it is also roughly

true of their static frequency. Since the job of the BTB is to remember the branches in a program, the BTB is attempting to remember the same context as the instruction cache [113]. Therefore, it is interesting to consider which of these (the BTB or the instruction cache) remembers more context, and whether one of them can prefetch for the other.

If the average instruction size is one word (four bytes), and there is one branch per four instructions, then on the average, each quadword contains one branch. (This is primarily why we have discussed instruction prefetching in units of quadwords.) Therefore, a BTB that contains N branch entries holds (roughly) the same context as an instruction cache having N quadwords. To compare these two (in bytes) requires an estimate of the size of a BTB entry.

For the sake of putting down some approximate numbers, let's assume that we are working with 32-bit addresses, and that we are using full address tags. Next, we approximate the size of an entry in a 4K-entry, four-way set-associative BTB. Recall that an entry can have eight fields. With these assumptions, the size of each field is:

1. Valid - one bit.
2. Action - one or two bits.
3. Tag - If there are 4K entries and four sets, then ten bits are used for the congruence class selection (i.e., not needed in the tag). Since the tag discriminates at the quadword granularity, the low-order four bits of the address are not part of the tag. Thus, for a branch address, the tag is 32-10-4 = eighteen bits.
4. Offset - The low-order branch-address bits (the byte address within the quadword). For a quadword, this is four bits.
5. Target Address - The address of the target instruction - thirty-two bits.
6. Length - one bit.
7. Special T - one or two bits.
8. Check bits - between zero and eight bits.

The total is in the range of 58-68 bits. So that we can work with round numbers, we'll call this eight bytes, which is half of a quadword.

Therefore, for an N-entry BTB to hold the same context as an N-quadword cache, it requires half the capacity of the cache in bytes. In practice, it would be unusual to make a BTB this large. In the first place, it is desirable to be able to cycle the BTB quickly - so that on a hit, the very next lookup can be done on the target address provided by the hit. (When the BTB cannot cycle this fast, one can then explore whether it is worthwhile to do "hedge-BTB lookups" in the event that a hit turns out to have been wrong.) Therefore, BTBs typically do not remember as much context as the instruction cache.

However, if a BTB remembered *more* than the instruction cache, then running the BTB well ahead of the instruction stream would provide a vehicle for prefetching for the instruction cache [114] [115] [116] [117]. Since implementations usually have these relative sizes reversed, it is possible to consider multilevel BTBs (i.e., a hierarchy of pageable BTBs), and to use the cache miss process as a trigger to prefetch BTB information down the (posited) BTB hierarchy [118] [119] [120].

We should mention that it is possible to economize on the size of BTB entries by assuming that target addresses are physically proximate to branch addresses, which is usually true. If we assume that a target instruction will fall within one Kilobyte of the branch instruction, then we can shorten the Target Address field to eleven bits. We can also eliminate many of the high-order bits from the Tag without losing much [121] [122] [123]. There have been a number of proposals like this, which can cut the size of an entry appreciably without losing much accuracy.

Since we have demonstrated that a BTB entry already requires lots of storage (eight bytes), it is then reasonable (in a relative sense) to add even more information for some of the branches if that will improve the predictive accuracy further. Most of the branches in a program are very predictable (e.g., 99+% accurate), and the BTB will do just fine by remembering the last action and target address for these. In real commercial code, BTBs will be found to have accuracies in the 75-90% range.

Suppose that we achieve an accuracy of 90%. It is not the case that each branch in the program is 90% predictable. More likely, we will find that 80% of the branches in the program are nearly 100% predictable, and 20% (or so) are predicted with very poor accuracy (50% or so). This tells us that the way in which we are trying to predict these 20% does not work for these particular branches. Then increasing our aggregate accuracy from 90% to a much higher number requires doing something even more exotic for this (relatively small) subset of the branches.

On a side note, since it is a relatively small percentage of the branches that are apparently intractable, a very small cache can be used to store the alternate paths for only these branches. Since the cache is not used to store the alternate paths of the predictable branches, it can be very small, hence fast. Such a cache enables quicker recoveries when these intractable branches are mispredicted [124] [125] [126].

On the other hand, a branch outcome is not a random event. It is the result of a calculation or a test on the state of an operand [127] [128] [129] [130] [131]. Many of the least-predictable branches depend of the test of a single byte, and frequently the state of that byte is set (by a store) well ahead of the test (e.g., hundreds of cycles). In addition, it is sometimes possible to predict the values of the operands themselves [132] [133]. There have been proposals for additional tables that maintain entries (indexed by the addresses of the relevant test-operands) that contain a pointer to the branch instruction affected by the test-operand, the manner in which that test-operand is tested to determine

the outcome of the branch, and the previous outcome of that branch [127] [128] [134].

When the address of a store operand hits in this table, the operand being stored is tested (by the prediction hardware) in the manner described in the table, and if the new (determined) branch outcome differs from the historical outcome, an update is sent to the affected branch entry in the BTB. If the store occurs enough ahead of the instruction prefetch (which is typically the case), then the BTB will make a correct prediction.

There are another subset of unpredictable branches that cannot be predicted in this manner because the operand that is tested is not always fetched from the same address. The test instruction that determines the branch outcome may test different operands at different times. For many of these instructions, there is a strong correlation between the operand that is tested and the branch outcome. (Generally, the operands being tested are statically defined constants.)

When this is the case, the branch is very predictable once it is known which of the operands is to be tested. Since the branch outcome (taken or not taken) depends on the branch itself (discriminated by the branch address) and on the operand that is tested (discriminated by the operand address), this method is called an "Instruction cross Data" (IXD) method.

Unfortunately, the correct operand address is not generally known until the test instruction (which immediately precedes the branch instruction) performs address generation (i.e., this instruction generates the operand address that we need to know). This is too late to influence the instruction prefetching, so this technique is more applicable at decode-time [135]. Since this table is used at decode time to predict branch action, its entry is small (one bit). The table is indexed by a hashing of the instruction address with the operand address. It looks very much like predictor shown in Figure 1.6.b, except that an operand address is used instead of the "Global Actions" that are shown in the figure.

In real machines, since prefetch-time mechanisms are costly, and (therefore) cannot remember much context, and since decode-time mechanisms are simple, and can remember lots of context, and since the two operate at different points in the pipeline, we generally use both mechanisms in tandem [136] [137] [138] [139] [70]. For example, with eight bytes per entry, an eight Kilobyte BTB remembers one Kilobranches, which corresponds to a sixteen Kilobyte working set. With one bit per entry, an eight Kilobyte BHT remembers sixty-four Kilobranches, which corresponds to a one Megabyte working set.

It will frequently be the case that the instruction decoder encounters branches in the instruction stream that the BTB did not know were there (this is not a distinguishable event if the Valid bit is used as a Taken indicator - hence the use of these distinct fields). Then, although the instruction prefetching had not been redirected by the BTB, it is at least still possible to predict the branch as "taken" (if it is) at decode time if we are also using a BHT (a.k.a.,

DHT). This allows us to mitigate some of the penalty associated with the BTB miss (assuming that the branch is taken).

Thus, in a real machine, there is nothing as simple as "the branch predictor." There will typically be several mechanisms at work (both independently and in tandem) at different points in the pipeline. There will also be different points at which recovery is done (based on branch presence or target or action becoming known, or based on a more reliable predictor supplanting the guess made by a less reliable predictor). There will also be different opportunities to hedge (do BTB searches and instruction prefetches) down alternate paths, depending on the bandwidth available, and the complexity of the buffering that we will want to deal with.

Predicting a branch is not a single event. It is an ongoing process as branch information makes its way through a pipeline. Although quoting some theoretical accuracy (derived from an execution trace), and some nominal penalty for a wrong guess is helpful in getting a rough idea about branch performance, simulation in a cycle-accurate timer (which includes modeling all table lookups and updates explicitly) is needed for an accurate picture of processor performance [140].

As we have already mentioned, "accuracy" as derived from an execution trace can be wrong when prediction mechanisms depend on "last" branch action, because in a real machine, the "last" action has not yet happened at the time that the prediction is made.

Another crucial aspect of prediction that we have not yet mentioned is whether the required updates (to our various tables) have been able to occur by the time that they are needed. When doing analysis from trace tapes, it is generally assumed that the updates magically get done, and that if the information in the table is correct, that the branch will be predicted correctly in a real machine. This is not always the case [141].

For example, we have seen comparative simulations in which higher performance is achieved by making the caches smaller. Why this occurs is as follows. If the BTB search has priority over the BTB update, then in branch-laden code, the BTB is constantly being searched, and the updates to it do not get done. Because the BTB does not get updated, the branches are repeatedly guessed wrong (although in an execution trace analysis, they would be counted as "right," since the updates should have occurred). When the cache size is reduced, cache misses cause the processor to stall, and when the instruction prefetching stops, the BTB updates get made. Then, although the processor takes an occasional cache miss, the branches get predicted correctly, and the overall effect is that the processor runs faster.

Therefore, it is important to consider how updates to the BTB are prioritized relative to searches. It is also important to make sure that there is adequate update bandwidth. (This is why we had mentioned that it can be advantageous to organize a BTB on double-quadwords when the instruction prefetch is a quadword.)

Another phenomenon worth mentioning is that because of the dynamics of

searching and updating a BTB simultaneously, in some configurations (meaning the read-write priorities on a finite number of ports) it is possible for a running program to create multiple entries for the same branch instruction with predictions that disagree. After this happens, the hit-logic may encounter coincident hits with opposite predictions for a given branch.

Some prediction algorithms require much more update bandwidth than others, and this needs to be taken into account when comparing them. For example, an algorithm that requires knowing the last N actions for a specific branch requires updating the table each time the branch executes. A simpler algorithm may only require an update when the action changes (from taken to not taken). An execution-trace analysis may show that the first algorithm achieves higher accuracy. In a running machine, the update bandwidth may interfere with the ability to search the table in a timely way, and the simpler algorithm - although less accurate - may achieve higher performance. Therefore, better accuracy does not necessarily imply better performance.

In the section on predicting branch action, we had mentioned that predictors that take more than a cycle (such as those involving two-stage lookups, or for large BTBs) are problematic. At that time, we had not yet gotten into the details of instruction prefetching, and had just said that it was not useful to get a correct prediction too late. In fact, this is particularly damaging for prefetch-time predictors.

When a predictor takes more than a single cycle to predict, the prediction must be done out of phase (and ahead of) instruction prefetching, otherwise it cannot drive the prefetching correctly. When the first wrong prediction occurs, prefetching is redirected to the correct target address (as determined by the processor), and a quadword of instructions are prefetched. If no prediction is available by the end of the next cycle (and none will be, because we are assuming that the predictor takes more than a single cycle), then instruction prefetching has no choice but to fetch the next sequential quadword blindly.

It is very likely that each quadword of instructions contains a branch. Since the prefetch mechanism cannot respond within a cycle, instruction prefetching is always done blindly following a wrong prediction (and if at a quadword per cycle rate, usually incorrectly) until a branch prediction (pertaining to a branch in the first quadword that was prefetched) becomes available. Since the prediction comes later than when its associated prefetch should have been initiated, the prediction is (de facto) ”wrong” even when it is factually correct, because it came too late to redirect the prefetching to the correct target address.

This will trigger another recovery action, and for the same reason, the next branch prediction is also (likely) missed. Hence, because of timing, a single wrong prediction can snowball into a sequence of ”wrong” predictions, even when the predictor has the correct information about all branches subsequent to one that is guessed wrong.

It is very important to keep the timing of the prediction mechanism to one cycle. In high-frequency designs where the pipeline stages are very lean,

this can be problematic, since tables may have to be kept very small, hence inherently inaccurate.

1.8 High ILP Environments

Throughout the 1990s, processor complexity was increased in an effort to achieve very high levels of *Instruction Level Parallelism* (ILP). Doing this requires decoding and executing multiple instructions per cycle.

In the late 1970s, the first "*Reduced Instruction Set Computer*" (RISC) machine, the 801 [12], decoded and executed a very specific four instructions per cycle. By doing a fixed-point operation (to manipulate loop indices), a load (or a store), a floating-point operation, and a branch (e.g., a loop), the 801 could do vector processing in superscalar mode. When running this kind of program at full speed, the 801 encounters a branch every cycle.

By the mid 1990s, people were talking about eight-at-a-time superscalar machines, and by the end of the 1990s the ante was up to sixteen-at-a-time, and even more [16]. With a branch occurring every four instructions, decoding more than four instructions per cycle requires predicting more than one branch per cycle [142] [143] [144] [145] [146] [147] [148] [17], as well as doing multiple disjoint instruction prefetches per cycle [142] [143] [149]. The complexity of this is considerable.

Basically, this requires that BTB entries contain monolithic information about sequences of branches (including multiple <branch address; target address pairs), and that it generate multiple instruction prefetches (at each of the predicted target addresses) each cycle. If the amount of information in a BTB grows too large (say it becomes a quadword - the same size as the instruction sequence it is predicting), then one needs to question the efficacy of using a BTB (and very high ILP techniques) against simply making the caches larger. In this regime, ILP techniques like trace-caches start to make a lot more sense [150].

Currently, the quest for higher ILP has abated as the efficient use of power has become paramount [15], and modern designs are becoming much more energy-concious. At the same time, multithreading is a growing trend, and this requires multiple disjoint instruction streams in flight simultaneously.

Multithreading has some of the same aspects as very wide-issue superscalar. It requires doing BTB (if it is used) lookups on behalf of multiple instruction streams. While these lookups might be simultaneous (requiring multiple banks or ports) or not, there is another aspect to this that is problematic. If the instruction streams are not disjoint, they nonetheless can operate on different data, and the same branch (in two streams, say) may have different outcomes. Thus, each stream will damage the predictive accuracy of the other streams.

If the instruction streams are disjoint, since BTB lookups are done with virtual addresses, the BTB (as it is) cannot distinguish the streams, so it can provide one stream with (erroneous) branch predictions that are generated by different streams. (E.g., the BTB will find branches that don't exist.) This can be fixed by adding "stream tags" to the BTB. However, since the required number of branch entries per virtual quadword is roughly equal to the number of streams, the set-associativity needs to be increased accordingly, or thrashing will greatly reduce the accuracy of predictions - even with stream tags.

Although a highly set-associate (hence more complicated) BTB would be able to discriminate between streams because of the tagging, the overhead of doing this is less reasonable in a BHT (a.k.a., DHT), where an entry is 1 bit. Hence the likely use of a BHT is to allow all of the streams to update and use the BHT in mutual collision. This will tend to randomize the predictions, and reduce their accuracy. The alternative is to use stream tags (essentially, to have a separate BHT for each stream). While this works, remember that for a fixed number of bits, dividing them among the streams means that each stream will remember much less context - hence have a lower predictive accuracy.

It costs a lot to run fast. And many of the mechanisms required to do this start breaking down as certain thresholds are exceeded. We had made the case in Section 1.2 that ILP is very fundamentally limited by the branches. Many of those limitations in high ILP environments and in highly pipelined machines are due to complicated second-order effects which go well beyond the scope of Minsky's conjecture.

Running fast requires resolving sequences of branches as quickly as possible so as to facilitate the correct supply of instructions to the execution end of the pipeline at a high rate. There have been (and continue to be) proposals for "runahead threads" which attempt to run the branch portion of the instruction stream speculatively ahead of the canonical instruction processing so as to resolve the branches early [7] [8] [151] [152] [10] [153].

1.9 Summary

The branch instruction is the workhorse of modern computing. Processor performance is fundamentally limited by the behaviors of the branch instructions in programs. Achieving high performance requires preserving the autonomy of instruction prefetching [19], which requires that the branch instructions be accurately anticipated, and predicted correctly.

Conceptually, predicting branches is simple. The vast majority of branches usually do exactly the same thing each time they are executed. Predictors

based on this principle are fairly straightforward to implement, and do a reasonably good job. To do an even better job of predicting branches requires much more complexity. Also, the details of implementation in a real machine can make the realities of branch prediction more complex still. We touched on some of those details in this chapter.

Because of the complexity of operations within a real pipeline, we emphasized that getting a true picture of the efficacy of branch prediction requires simulating it in a cycle-accurate timer model in which the details of the prediction mechanism (including updating it) are modeled explicitly [140]. We also tried to make clear that in a real machine, there is no such thing as "the prediction mechanism." Branch prediction involves interaction between the states of several mechanisms (including the processor itself) at different stages in the pipeline.

Since there are several ways in which a prediction can be wrong, and these become manifest in different stages of the pipeline, there are many different penalties involved. For this reason, it is not particularly meaningful to talk about "the accuracy" of an algorithm. "The accuracy" usually refers to some abstract analysis performed on an execution trace. This number does not necessarily transfer to a real machine in a direct way.

While the complexity and the details of branch prediction can appear daunting, there are two saving graces. First, simple algorithms work pretty well. Many complex algorithms can outsmart themselves when actually implemented.

Second, a prediction is just that: a prediction. It does not *have* to be right. In this sense, predicting branches is simpler than operating the rest of the machine. In our algorithms, we can make approximations, take shortcuts, and shave-off lots of bits, and still do pretty well. The things that we are predicting (branches) are capricious - there is no point in obsessing excessively over predicting something that has inherent unpredictability.

It can be very interesting to ponder evermore clever prediction strategies, and to extrapolate their theoretical limits. It is equally interesting and challenging to ponder the complexities of a real environment (threads, ILP, tag aliasing, virtual/real aliasing, overlays, pipelining, changing operand state, register pointers, etc.), and the ways in which the vagaries and vicissitudes of that environment can foil a prediction mechanism. When all is said and done, our predictors will still generate predictions that can be wrong for multiple reasons, or that can be right by fortuitous accident.

While achieving high predictive accuracy is necessary for running very fast, adding too much complexity (to achieve that accuracy) can (ironically) hurt the accuracy because of second-order effects. If adequate performance suffices, the complexity should be kept small. If truly high performance is required, the added complexity needs to be modeled in detail to be sure that it works in a real machine.

While doing good branch prediction is essential in modern computers, many of the more exotic schemes conceived, while boons to deipnosophists, may yet

be a step or two away from implementation. When designing a predictor for a real machine, the most important thing to practice is practicability.

References

[1] A.W. Burks, H.H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In *in Aspray and Burks [AB87]*, volume 1, pages 97–142. The Institute of Advanced Study, Princeton, Report to the U.S. Army Ordinance Department.

[2] J. von Neumann. *Collected Works*, volume 5. MacMillan, New York, 1963.

[3] A. W. Burks and A. R. Burks. The eniac: First general purpose electronic computer. *Annals of the History of Computing*, 3(4):310–399, 1981.

[4] K. Godel. Uber formal unentscheidbare satze der principia mathematica und verwandter systeme i. *Mh. Math. Phys.*, (38):173–198, 1931. (English translation in M. Davis, The Undecidable, pp. 4-38, Raven Press, Hewlett, N.Y., 1965.).

[5] E. Morrison and P. Morrison. *Charles Babbage and his Calculating Engines.* New York, 1961.

[6] Fotheringham J. Dynamic storage allocation in the atlas computer including an automatic use of backing store. *Communications of the ACM*, 4(10):435–436, 1961.

[7] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Pipeline processing apparatus for executing instructions in three streams, including branch stream, pre-execution processor for pre-executing conditional branch instructions. U.S. Patent #4991080, assigned to IBM Corporation, Filed Mar. 13, 1986, Issued Feb. 5, 1991.

[8] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Multiple sequence processor system. U.S. Patent #5297281, assigned to IBM Corporation, Filed Feb. 13, 1992, Issued Mar. 22, 1994.

[9] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, Nov. 1999.

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

[10] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 19-24 2001.

[11] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill Book Company, 1984. p. 14.

[12] G. Radin. The 801 minicomputer. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, Palo Alto, California, March 1982.

[13] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research & Development*, pages 215–232, Feb. 1997.

[14] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–34, May 25-29 2002.

[15] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma. Optimizing pipelines for power and performance. In *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 333–344, November 2002.

[16] Y.N. Patt, S.J. Patel, M. Evers, D.H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, Sept. 1997.

[17] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, May 25-29 2002.

[18] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept./Oct. 1997.

[19] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 16–27, Nov. 16-18 1999.

[20] T.Y. Yeh and Y.N. Patt. Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. In *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 164–175, Dec. 1-3 1993.

[21] A. Seznec and A. Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 241–252, June 9-11 2003.

[22] J. Cocke, B. Randell, H. Schorr, and E.H. Sussenguth. Apparatus and method in a digital computer for allowing improved program branching with branch anticipation, reduction of the number of branches, and reduction of branch delays. U.S. Patent #3577189, assigned to IBM Corporation, Filed Jan. 15, 1965, Issued May 4, 1971.

[23] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, Jan. 1967.

[24] W.F. Bruckert, T. Fossum, J.A. DeRosa, R.E. Glackenmeyer, A.E. Helenius, and J.C. Manton. Instruction prefetch system for conditional branch instruction for central processor unit. U.S. Patent #4742451, assigned to Digital Equiptment Corporation, Filed May 21, 1984, Issued May 3, 1988.

[25] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Using a small cache to hedge for a bht. *IBM Technical Disclosure Bulletin*, page 1737, Sept. 1985.

[26] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Hedge fetch history table. *IBM Technical Disclosure Bulletin*, pages 101–102, Feb. 1989.

[27] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 165–175, Dec. 2-4 1996.

[28] T.C. Mowry and E.A. Killian. Method and apparatus for reducing delays following the execution of a branch instruction in an instruction pipeline. U.S. Patent #5696958, assigned to Silicon Graphics, Inc., Filed Mar. 15, 1995, Issued Dec. 9, 1997.

[29] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 38–47, Nov. 16-18 1999.

[30] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *Ninth International Symposium on High-Performance Computer Architecture*, pages 53–64, Feb. 8-12 2003.

[31] A.K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: an optimal form of speculative execution. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 313–325, Nov. 29- Dec. 1 1995.

[32] T.F. Chen. Supporting highly speculative execution via adaptive branch trees. In *Fourth International Symposium on High-Performance Computer Architecture*, pages 185–194, Feb. 1-4 1998.

[33] D.J. Lilja. Reducing the branch penalty in pipelined processors. *Computer*, 21(7):47–55, July 1988.

[34] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Group approach to branch prediction. *IBM Technical Disclosure Bulletin*, page 4339, Dec. 1984.

[35] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Comprehensive branch prediction mechanism for bc. *IBM Technical Disclosure Bulletin*, pages 2255–2262, Oct. 1985.

[36] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 142–152, Dec. 2-4 1996.

[37] C.C. Lee, I.C.K. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 4–13, Dec. 1-3 1997.

[38] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 241–250, Jan. 8-12 2000.

[39] P.K. Dubey and M.J. Flynn. Branch strategies: Modeling and optimization. *IEEE Transactions on Computers*, 40(10):1159–1167, Oct. 1991.

[40] J.P. Linde. Microprogrammed control system capable of pipelining even when executing a conditional branch instruction. U.S. Patent #4373180, assigned to Sperry Corporation, Filed July 9, 1980, Issued Feb. 8, 1983.

[41] D.R. Ditzel and H.R. McLellan. Arrangement and method for speeding the operation of branch instructions. U.S. Patent #4853889, assigned to AT&T Bell Laboratories, Filed May 11, 1987, Issued Aug. 1, 1989.

[42] S. Krishnan and S.H. Ziesler. Branch prediction and target instruction control for processor. U.S. Patent #6324643, assigned to Hitachi, Ltd., Filed Oct. 4, 2000, Issued Nov. 27, 2001.

[43] K. Wada. Branch advanced control apparatus for advanced control of a branch instruction in a data processing system. U.S. Patent #4827402, assigned to Hitachi, Ltd., Filed Apr. 22, 1986, Issued May 2, 1989.

[44] H. Potash. Branch predicting computer. U.S. Patent #4435756, assigned to Burroughs Corporation, Filed Dec. 3, 1981, Issued Mar. 6, 1984.

[45] D.R. Beard, A.E. Phelps, M.A. Woodmansee, R.G. Blewett, J.A. Lohman, A.A. Silbey, G.A. Spix, F.J. Simmons, and D.A. Van Dyke.

Method of processing conditional branch instructions i scalar/vector processor. U.S. Patent #5706490, assigned to Cray Research, Inc., Filed June 7, 1995, Issued Jan. 6, 1998.

[46] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–164, Dec. 2-4 1996.

[47] D.I. August, D.A. Connors, J.C. Gyllenhaal, and W.M.W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Third International Symposium on High-Performance Computer Architecture*, pages 84–93, Feb. 1-5, 1997.

[48] T.Y. Yeh, M.A. Poplingher, and M. Rahman. Optimized branch predictions for strongly predicted compiler branches. U.S. Patent #6427206, assigned to Intel Corporation, Filed May 3, 1999, Issued July 30, 2002.

[49] M. Tremblay. Software branch prediction filtering for a microprocessor. U.S. Patent #6374351, assigned to Sun Microsystems, Inc., Filed Apr. 10, 2001, Issued Apr. 16, 2002.

[50] J. Smith. Branch predictor using random access memory. U.S. Patent #4370711, assigned to Control Data Corporation, Filed Oct. 21, 1980, Issued Jan. 25, 1983.

[51] R. Nair. Optimal 2-bit branch predictors. *IEEE Transactions on Computers*, 44(5):698–702, May 1995.

[52] P.M. Paritosh, R. Reeve, and N.R. Saxena. Method and apparatus for a single history register based branch predictor in a superscalar microprocessor. U.S. Patent #5742905, assigned to Fujitsu, Ltd., Filed Feb. 15, 1996, Issued Apr.21, 1998.

[53] Y. Totsuka and Y. Miki. Branch prediction apparatus. U.S. Patent #6640298, assigned to Hitachi, Ltd., Filed Apr. 7, 2000, Issued Oct. 28, 2003.

[54] A.J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[55] J.F. Brown III, S. Persels, and J. Meyer. Branch prediction unit for high-performance processor. U.S. Patent #5394529, assigned to Digital Equiptment Corporation, Filed July 1, 1993, Issued Feb. 28, 1995.

[56] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 15–23, Nov. 29 - Dec. 1 1995.

[57] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, June 22-24 1995.

[58] B. Fagin and A. Mital. The performance of counter- and correlation-based schemes for branch target buffers. *IEEE Transactions of Computers*, 44(12):1383–1393, Dec. 1995.

[59] S. Reches and S. Weiss. Implementation and analysis of path history in dynamic branch prediction schemes. *IEEE Transactions on Computers*, 47(8):907–912, Aug. 1998.

[60] S.C. Steely and D.J.Sagar. Past-history filtered branch prediction. U.S. Patent #5828874, assigned to Digital Equiptment Corporation, Filed June 5, 1996, Issued Oct. 27, 1998.

[61] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 86–97, June 30 - July 4 2001.

[62] G.D. Zuraski, J.S. Roberts, and R.S. Tupuri. Dynamic classification of conditional branches in global history branch prediction. U.S. Patent #6502188, assigned to Advanced Micro Devices, Inc., Filed Nov. 16, 1999, Issued Dec. 31, 2002.

[63] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving branch prediction by dynamic data flow-based identification of correlated branches from a large global history. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 314–323, June 9-11 2003.

[64] D.A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan. 19-24 2001.

[65] M. Kampe, P. Stenstrom, and M. Dubois. The fab predictor: Using fourier analysis to predict the outcome of conditional branches. In *Eighth International Symposium on High-Performance Computer Architecture*, pages 223–232, Feb. 2-6, 2002.

[66] D.A. Jimenez. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 243–252, Dec. 3-5 2003.

[67] D.B. Sand. Branch prediction apparatus and method for a data processing system. U.S. Patent #4430706, assigned to Burroughs Corporation, Filed Oct. 27, 1980, Issued Feb. 7, 1984.

[68] J. J. Losq, G. S. Rao, and H. E. Sachar. Decode history table for conditional branch instructions. U.S. Patent #4477872, assigned to IBM Corporation, Filed Jan. 15, 1982, Issued Oct. 16, 1984.

[69] S.G. Tucker. The ibm 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.

[70] D.R. Kaeli and P.G. Emma. Improving the accuracy of history-based branch prediction. *IEEE Transactions on Computers*, 46(4):469–472, Apr. 1997.

[71] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 249–258, Nov. 30 - Dec. 2 1998.

[72] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Improved decode history table hashing. *IBM Technical Disclosure Bulletin*, page 202, Oct. 1989.

[73] P.G. Emma, D.R. Kaeli, J.W. Knight, J.H. Pomerene, and T.R. Puzak. Aliasing reduction in the decoding history tables. *IBM Technical Disclosure Bulletin*, pages 237–238, June 1993.

[74] A.N. Eden and T.N. Mudge. The yags branch prediction scheme. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 69–77, Nov. 30 - Dec. 2 1998.

[75] C.M. Chen and C.T. King. Walk-time address adjustment for improving the accuracy of dynamic branch prediction. *IEEE Transactions on Computers*, 48(5):457–469, May 1999.

[76] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 251–262, Jan. 8-12, 2000.

[77] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 19-21 1992.

[78] T.Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[79] S. Sechrest, C.C. Lee, and T. Mudge. The role of adaptivity in two-level adaptive branch prediction. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 264–269, Nov. 29- Dec. 1 1995.

[80] M. Evers, S.J. Patel, R.S. Chappell, and Y.N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, June 27 - July 1 1998.

[81] T. Juan, S. Sanjeevan, and J.J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 155–166, June 27 - July 1 1998.

[82] A.R. Talcott. Methods and apparatus for branch prediction using hybrid history with index sharing. U.S. Patent #6510511, assigned to Sun Microsystems, Inc., Filed June 26, 2001, Issued Jan 21, 2003.

[83] G.P. Giacalone and J.H. Edmondson. Method and apparatus for predicting multiple conditional branches. U.S. Patent #6272624, assigned to Compaq Computer Corporation, Filed Apr. 4, 2002, Issued Aug. 7, 2001.

[84] D.A. Jimenez, S.W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 67–76, Dec. 10-13, 2000.

[85] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, and J.R. Robinson. Multi-prediction branch prediction mechanism. U.S. Patent #5353421, assigned to IBM Corporation, Filed Aug. 13, 1993, Issued Oct. 4, 1994.

[86] P.H. Chang. Branch predictor using multiple prediction heuristics and a heuristic identifier in the branch instruction. U.S. Patent #5687360, assigned to Intel Corporation, Filed Apr. 28, 1995, Issued Nov. 11, 1997.

[87] P.Y. Chang, E. Hao, and Y.N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 252–257, Nov. 29 - Dec. 1 1995.

[88] S. Mallick and A.J. Loper. Automatic selection of branch prediction methodology for subsequent branch instruction based on outcome of previous branch prediction. U.S. Patent #5752014, assigned to IBM Corporation, Filed Apr. 29, 1996, Issued May 12, 1998.

[89] A. Falcon, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet / critic hybrid branch prediction. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 250–261, June 19-23, 2004.

[90] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, F.J. Sparacio, and C.F. Webb. Subroutine return through the branch history

table. U.S. Patent #5276882, assigned to IBM Corporation, Filed July 27, 1990, Issued Jan. 4, 1994.

[91] N. Suzuki. Microprocessor having branch prediction function. U.S. Patent #5327536, assigned to NEC Corporation, Filed May 22, 1991, Issued July 5, 1994.

[92] C.N. Tran and W.K. Lewchuk. Branch prediction unit which approximates a larger number of branch predictions using a smaller number of branch predictions and an alternate target indication. U.S. Patent #5974542, assigned to Advanced Micro Devices, Inc., Filed Oct. 30, 1997, Issued Oct. 26, 1999.

[93] T.M. Tran. Branch prediction mechanism employing branch selectors to select a branch prediction. U.S. Patent #5995749, assigned to Advanced Micro Devices, Inc., Filed Nov. 19, 1996, Issued Nov. 30, 1999.

[94] E.H. Sussenguth. Instruction sequence control. U.S. Patent #3559183, assigned to IBM Corporation, Filed Feb. 29, 1968, Issued Jan. 26, 1971.

[95] S. Hanatani, M. Akagi, K. Nigo, R. Sugaya, and T. Shibuya. Instruction prefetching device with prediction of a branch destination address. U.S. Patent #4984154, assigned to NEC Corporation, Filed Dec. 19, 1988, Issued Jan. 8, 1991.

[96] J.S. Liptay. Design of the ibm enterprise system/9000 high-end processor. *IBM Journal of Research and Development*, 36(4):713–731, July 1992.

[97] D.B. Fite, J.E. Murray, D.P. Manley, M.M. McKeon, E.H. Fite, R.M. Salett, and T. Fossum. Branch prediction. U.S. Patent #5142634, assigned to Digital Equiptment Corporation, Filed Feb. 3, 1989, Issued Aug. 25, 1992.

[98] T. Morisada. System for controlling branch history table. U.S. Patent #5345571, assigned to NEC Corporation, Filed Aug. 5, 1993, Issued Sept. 6, 1994.

[99] M. Kitta. Arrangement for predicting a branch target address in the second iteration of a short loop. U.S. Patent #5394530, assigned to NEC Corporation, Filed Feb. 22, 1994, Issued Feb. 28, 1995.

[100] K. Shimada, M. Hanawa, K. Yamamoto, and K. Kaneko. System with reservation instruction execution to store branch target address for use upon reaching the branch point. U.S. Patent #5790845, assigned to Hitachi, Ltd., Filed Feb. 21, 1996, Issued Aug. 4, 1998.

[101] C. Joshi, P. Rodman, P. Hsu, and M.R. Nofal. Invalidating instructions in fetched instruction blocks upon predicted two-step branch operations with second operation relative target address. U.S. Patent #5954815,

assigned to Silicon Graphics, Inc., Filed Jan. 10, 1997, Issued Sept. 21, 1999.

[102] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Branch target table. *IBM Technical Disclosure Bulletin*, page 5043, Apr. 1986.

[103] C.H. Perleberg and A.J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, Apr. 1993.

[104] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, April 18-21 1994.

[105] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Highly accurate subroutine stack prediction mechanism. *IBM Technical Disclosure Bulletin*, pages 4635–4637, Mar. 1986.

[106] C.F. Webb. Subroutine call and return stack. *IBM Technical Disclosure Bulletin*, page 221, April 1988.

[107] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Indirect targets in the branch history table. *IBM Technical Disclosure Bulletin*, page 265, Dec. 1989.

[108] D.R. Kaeli and P.G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 27-30 1991.

[109] S. Narita, F. Arakawa, K. Uchiyama, and H. Aoki. Branching system for return from subroutine using target address in return buffer accessed based on branch type information in bht. U.S. Patent #5454087, assigned to Hitachi, Ltd., Filed Oct. 23, 1992, Issued Sept. 26, 1995.

[110] B.D. Hoyt, G.J. Hinton, D.B. Papworth, A.K. Gupta, M.A. Fetterman, S. Natarajan, S. Shenoy, and R.V. D'Sa. Method and apparatus for resolving return from subroutine instructions in a computer processor. U.S. Patent #5604877, assigned to Intel Corporation, Filed Jan. 4, 1994, Issued Feb.18, 1997.

[111] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 259–271, Nov. 30 - Dec. 2 1998.

[112] PM. Ukai, K. Tashima, and A. Aiichiro. Predicted return address selection upon matching target in branch history table with entries in return address stack. U.S. Patent #6530016, assigned to Fujitsu, Ltd., Filed Dec. 8, 1999, Issued Mar. 4, 2003.

[113] S.P. Kim and G.S. Tyson. Analyzing the working set characteristics of branch execution. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 49–58, Nov. 30 - Dec. 2 1998.

[114] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Multiple branch analyzer for prefetching cache lines. U.S. Patent #4943908, assigned to IBM Corporation, Filed Dec. 2, 1987, Issued July 24, 1990.

[115] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Prefetch deconfirmation based on bht error. *IBM Technical Disclosure Bulletin*, page 4487, Mar. 1987.

[116] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Next-sequential prefetching using a branch history table. *IBM Technical Disclosure Bulletin*, pages 4502–4503, Mar. 1987.

[117] J.B. Keller, P. Sharma, K.R. Schakel, and F.M. Matus. Training line predictor for branch targets. U.S. Patent #6647490, assigned to Advanced Micro Devices, Inc., Filed Oct. 14, 1999, Issued Nov. 11, 2003.

[118] J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, P.L. Rosenfeld, and F.J. Sparacio. Pageable branch history table. U.S. Patent #4679141, assigned to IBM Corporation, Filed Apr. 29, 1985, Issued July 7, 1987.

[119] T.Y. Yeh and H.P. Sharangpani. Method and apparatus for branch prediction using first and second level branch prediction tables. U.S. Patent #6553488, assigned to Intel Corporation, Filed Sept. 8, 1998, Issued Apr.22, 2003.

[120] D.R. Stiles, J.G. Favor, and K.S. Van Dyke. Branch prediction device with two levels of branch prediction cache. U.S. Patent #6425075, assigned to Advanced Micro Devices, Inc., Filed July 27, 1999, Issued July 23, 2002.

[121] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Dynamic distinction of distinguished segments. *IBM Technical Disclosure Bulletin*, pages 4065–4069, Feb. 1986.

[122] B. Fagin and K. Russell. Partial resolution in branch target buffers. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 193–198, Nov. 29- Dec. 1 1995.

[123] B. Fagin. Partial resolution in branch target buffers. *IEEE Transactions on Computers*, 46(10), Oct. 1997.

[124] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Branch wrong guess cache. *IBM Technical Disclosure Bulletin*, pages 310–311, May 1988.

[125] J.O. Bondi, A.K. Nanda, and S. Dutta. Integrating a misprediction recovery cache (mrc) into a superscalar pipeline. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, Dec. 2-4 1996.

[126] S. Wallace, D.M. Tullsen, and B. Calder. Instruction recycling on a multiple-path processor. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 44–53, Jan. 9-13 1999.

[127] P.G. Emma, J.H. Pomerene, G.S. Rao, R.N. Rechtschaffen, H.E. Sachar, and F.J. Sparacio. "store-time updates to the branch history table. U.S. Patent #4763245, assigned to IBM Corporation, Filed Oct.30, 1985, Issued Aug. 9, 1988.

[128] P.G. Emma, J.H. Pomerene, G.S. Rao, R.N. Rechtschaffen, H.E. Sachar, and F.J. Sparacio. Branch prediction mechanism in which a branch history table is updated using an operand sensitive branch table. U.S. Patent #4763245, assigned to IBM Corporation, Filed Oct. 30, 1985, Issued Aug. 9, 1988.

[129] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 59–68, Nov. 30 - Dec. 2 1998.

[130] T.H. Heil, Z. Smith, and J.E. Smith. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 28–37, Nov. 16-18 1999.

[131] L. Chen, S. Dropsho, and D.H. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Ninth International Symposium on High-Performance Computer Architecture*, pages 65–76, Feb. 8-12 2003.

[132] P.G. Emma, J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, and F.J. Sparacio. Operand history table. *IBM Technical Disclosure Bulletin*, pages 3815–3816, Dec. 1984.

[133] A. Sodani and G.S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 205–215, Nov. 30 - Dec. 2 1998.

[134] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Methods and apparatus for insulating a branch prediction mechanism from data dependent branch table updates that result from variable test operand locations. U.S. Patent #5210831, assigned to IBM Corporation, Filed Oct. 30, 1989, Issues May 11, 1993.

[135] P.G. Emma, J.H. Pomerene, G. Rao, R.N. Rechtschaffen, and F.J. Sparacio. Ixd branch history table. *IBM Technical Disclosure Bulletin*, pages 1723–1724, Sept. 1985.

[136] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Unencountered branch indication. *IBM Technical Disclosure Bulletin*, pages 5036–5037, Apr. 1986.

[137] P.G. Emma, J.W. Knight, J.W. Pomerene, T.R. Puzak, R.N. Rechtschaffen, J. Robinson, and J. Vannorstrand. Redundant branch confirmation for hedge fetch suppression. *IBM Technical Disclosure Bulletin*, page 228, Feb. 1990.

[138] S. McFarling. Combining branch predictors. Technical report, DEC WRL TN-36, June 1993.

[139] R. Yung. Design decisions influencing the ultrasparc's instruction fetch architecture. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, Dec. 2-4 1996.

[140] B. Calder, D. Grunwald, and J. Emer. A system level perspective on branch architecture performance. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–206, Nov 29- Dec. 1 1995.

[141] T.M. Tran and D.B. Witt. Superscalar microprocessor which delays update of branch prediction information in response to branch misprediction until a subsequent idle clock. U.S. Patent #5875324, assigned to Advanced Micro Devices, Inc., Filed Oct. 8, 1997, Issued Feb. 23, 1999.

[142] P.G. Emma, J.W. Knight, J.H. Pomerene, and T.R. Puzak. Simultanous prediction of multiple branches for superscalar processing. U.S. Patent #5434985, assigned to IBM Corporation, Filed Aug. 11, 1992, Issued July 18, 1995.

[143] D.N. Pnevmatikatos, M. Franklin, and G.S. Sohi. Control flow prediction for dynamic ilp processors. In *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–163, Dec. 1-3 1993.

[144] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–263, Nov. 29- Dec. 1 1995.

[145] M. Rahman, T.Y. Yeh, M. Poplingher, C.C. Scafidi, and A. Choubal. Method and apparatus for generating branch predictions for multiple branch instructions indexed by a single instruction pointer. U.S. Patent

#5805878, assigned to Intel Corporation, Filed Jan 31, 1997, Issued Sept. 8, 1998.

[146] S . Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Third International Symposium on High-Performance Computer Architecture*, pages 94–103, Feb. 1-5, 1997.

[147] A.R. Talcott, R.K. Panwar, R. Cherabuddi, and S. Patel. Method and apparatus for performing multiple branch predictions per cycle. U.S. Patent #6289441, assigned to Sun Microsystems Inc., Filed Jan. 9, 1998, Issued Sept. 11, 2001.

[148] R. Rakvic, B. Black, and J.P. Shen. Completion time multiple branch prediction for enhancing trace cache performance. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 47–58, June 10-14 2000.

[149] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–334, June3 22-24 1995.

[150] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 14–23, Dec. 1-3 1997.

[151] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 269–280, Dec. 10-13 2000.

[152] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 30 - July 4 2001.

[153] R.S. Chappell, F. Tseng, A. Yoaz, and Y.N. Patt. Microarchitectural support for precomputation microthreads. In *Proceedings of the 35th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 74–84, Nov. 18-22 2002.