RC23261 (96754) March 19, 1999
Electrical Engineering

IBM Research Report

A Decoupled Fetch-Execute Engine with Static Branch Prediction Support

Arthur A. Bright, Jason Fritts, Michael Gschwind
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Decoupled Fetch-Execute Engine with Static Branch Prediction Support

Arthur A. Bright, Jason Fritts, Michael Gschwind
IBM T. J. Watson Research Center
Yorktown Heights, New York

Abstract

We describe a method for supporting static branch prediction on a decoupled fetch-execute pipeline. Using instruction buffers to decouple instruction fetch from the execute pipeline is an effective way to minimize instruction cache penalties by allowing instruction fetch and stall miss handling to proceed independent of the execution pipeline. Dynamic branch prediction is typically used with such architectures, but it is not necessary to assume the cost of dynamic branch hardware when static prediction is sufficient. Traditional static branch prediction approaches were designed for lock-step pipelines and do not adapt well to decoupled fetch-execute pipelines, so alternative means of support were required. We describe the requirements for achieving efficient static branch prediction on a decoupled fetch-execute architecture, and presents the design and results for an implementation on an EPIC-style target architecture.

Keywords: decoupled fetch-execute, static branch prediction, prepare-to-branch, EPIC, VLIW

1. Introduction

As architecture and compiler designers continue to strive for greater degrees of parallelism, the effect of pipeline stall penalties on parallelism becomes very significant. For high levels of parallelism, the average number of cycles spent executing an instruction (CPI) must be much less than 1. Such a small CPI is only possible by minimizing the CPI penalties from stalls, thereby reducing their impact upon pipeline throughput. The problem of reducing stall penalties is aggravated by the potentially greater frequency of stalls due to higher instruction issue rates. It becomes necessary to find more capable methods for decreasing these penalties. Two common methods for reducing stall penalties include decoupled architectures and branch prediction.

Decoupled architectures use buffering and control mechanisms to dissociate memory accesses from the rest of the pipeline. When a cache miss occurs, the decoupled architecture allows the rest of the pipeline to continue moving forward, only stalling those instructions dependent upon that cache access. Decoupling of cache accesses from the pipeline can be used with either instruction or data caches. Decoupled data caches are not as effective in *Explicitly Parallel Instruction Computing* (EPIC)[1] or *Very Long Instruction Word* (VLIW) architectures, where a single instruction contains multiple operations, so any operation that is dependent upon a data cache miss stalls the entire instruction. Decoupling of the instruction cache access from the execute pipeline, hereafter referred to as *decoupled fetch-execute*, is beneficial for both superscalar and EPIC/VLIW architectures.

Decoupled fetch-execute architectures use instruction buffers and branch prediction for enabling instruction fetch to be independent from the rest of the pipeline. The instruction buffers are organized as a queue that receives instructions as they are fetched from the instruction cache. As instructions enter the queue, a branch prediction mechanism checks for the existence of a branch instruction. When a branch is found, the prediction determines the likely branch target and direction, and if necessary, redirects the instruction fetch to the predicted address. Most general-purpose processors today use dynamic branch prediction mechanisms, which can include

[1] In keeping with current usage, we use the term EPIC to refer to a variable-length VLIW architecture.

tables of prediction counters, history tables, and branch target buffers [1][2]. Many of these schemes add considerable hardware, and may affect the processor frequency.

Static branch prediction provides an alternate prediction method. It does not perform as well as dynamic branch prediction for most general-purpose applications, but does do well in some application markets, so architectures for these markets may be able to forego the cost of dynamic branch prediction. Such markets include media processing [3] and binary translation in software [4], which performs run-time compilation using dynamic profile statistics, enabling accurate static branch prediction.

Numerous static branch prediction schemes have been presented in the literature [5][6][7], but the majority have been designed for lock-step pipelines. These designs do not adapt well to decoupled fetch-execute architectures, so a new method was necessary to enable their use of static branch prediction. The remainder of this report examines the requirements for static branch prediction on decoupled fetch-execute architectures, and an implementation for an EPIC processor. Section 2 examines traditional static branch prediction methods, explaining the difficulties adapting these methods to decoupled instruction fetch architectures. Section 3 proposes a new method for static branch prediction and delineates the requirements for implementing this scheme and enabling the predicted branch target to be available for execution immediately after the branch operation. Section 4 describes implementation of the decoupled fetch-execute engine on an EPIC architecture. The performance is examined in section 5, and section 6 finishes with the conclusions.

2. Static Branch Prediction

Static branch prediction is the process of selecting at compile time the direction a conditional branch is expected to take. Conditional branches that are predicted as *not taken*, i.e. those expected to fall through to the sequential path, are easily supported since instruction fetch logic automatically continues sequentially. Unconditional branches and conditional branches that are predicted as *taken*, i.e. those expected to continue execution at a non-sequential target instruction, require support for redirecting the instruction fetch unit to begin prefetching the expected branch target prior to execution of the branch. It is desired that this prefetching begin immediately after the fetch of the branch instruction to enable execution of the expected branch target right after the branch executes.

One method for accomplishing this uses a prediction bit in the branch operation. After fetching the branch operation, the expected branch target address is sent to the instruction fetch unit if the prediction bit indicates *taken*. A problem with this method is that determination of the prediction direction and target address requires access to the contents of the branch operation. The expected branch target can only be fetched once the branch operation is returned by the instruction cache, the direction set by the prediction bit is determined, and the expected branch target address has been computed. As shown in Figure 1, in an instruction cache with *f* stages, the earliest the contents of the branch operation become available is *f* cycles after the fetch was initiated. However, the desired time to begin fetch of the expected branch target is only 1 cycle after the branch begins being fetched. Consequently, the use of a prediction bit for performing static branch prediction will usually not allow ideal timing for fetching the predicted branch target, but will insert at least *f*-1 delay cycles between the branch and predicted target.

An alternative technique is to issue a fetch hint operation, a *Prepare-to-Branch* (PBR) operation, for the expected branch target. The PBR operation typically has one main field that indicates the address or displacement of the predicted branch target. Additional fields may include a register destination for the address of the expected branch target, or a predicate condition which indicates whether to execute the PBR operation. Such a predicate is particularly useful for implementing more intelligent static branch prediction methods, such as branch correlation. Performing static branch correlation without using predication can require substantial code duplication [8][9].

A critical aspect of the Prepare-to-Branch operation is timing. The PBR operation should be scheduled to begin fetch of the expected branch target immediately after initiating fetch of the corresponding branch operation, as indicated by option 1 in Figure 1. It cannot redirect fetch earlier as that will prevent the branch operation from being fetched, and it should not redirect fetch later to avoid extra delay between the branch and the predicted target. Achieving this timing requires two mechanisms. First, a means is necessary for associating the PBR operation with the branch it is predicting. This association is hereafter referred to as *pairing*, and the corresponding branch is called the *paired branch*. Second, there must exist a mechanism for recognizing that the paired branch fetch has started and that fetch of the expected branch target may begin.

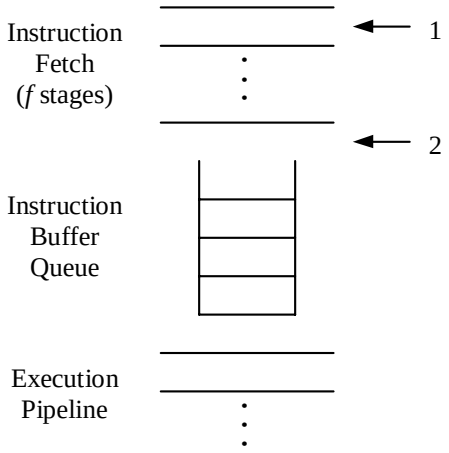

Figure 1 – Predicted branch target fetch timing: 1) desired timing, right after beginning fetch of paired branch, 2) fetch timing requiring contents of branch operation

There are two principal approaches for implementing the Prepare-to-Branch operation. One method, commonly used in in-order lock-step pipelines, is to schedule the PBR operation a fixed number of instructions before the branch [5]. The fixed-position of the branch with respect to the PBR operation serves both as the means for uniquely defining the paired branch as well as indicating when the fetch of the expected branch target begins. The dependent nature of all pipeline stages in a lock-step pipeline ensures correct fetch timing in the fixed-position method. However, the fixed-position timing model is only effective on lock-step pipelines and cannot be used in decoupled fetch-execute architectures, which eliminate the dependency between the execution pipeline and instruction fetch pipeline.

Another scheme for implementing the Prepare-to-Branch operation uses a register destination in the PBR operation for pairing with the branch operation [6]. The branch operation uses the same register destination to provide its target address. The register name provides a means for pairing without necessitating a fixed position for the PBR operation, and allows greater scheduling freedom. Implementing timing for this technique requires first determining if the branch operation is available before starting fetch of the predicted branch target. Availability of the branch operation can be determined by searching the newly fetched instructions, the instruction buffers, and the pipeline, for a branch operation using the same register as the PBR operation. Once the paired branch is found, fetch of the expected branch target may begin. Like the prediction bit scheme, this scheme also requires access to the contents of the branch

operation before enabling fetch of the expected branch target, so it too forces a minimum delay of *f*-1 cycles between fetch of the branch and its predicted target.

3. Proposed Method

The main problem with the existing static branch prediction methods is that the contents of the paired branch operation are needed prior to fetching the predicted branch target. In one approach, the prediction bit is necessary to determine the predicted direction, while the second scheme requires the name of branch register source to determine if pairing exists with a pending PBR operation. Instead, proposed is a Prepare-to-Branch prediction method that does not require the contents of the branch operation.

The proposed solution uses a field in the PBR operation specifying the last few bits of the address of the branch operation. The number of bits used determines the size of the scheduling window (preceding the paired branch) over which the PBR operation can be scheduled. The only restriction upon scheduling PBR operations within this scheduling window is that PBR operations occur in the same order as their paired branches. This ensures that each PBR operation looks in the appropriate scheduling window when searching for its paired branch.

An alternative approach to pairing that does not require access to the branch contents is to indicate the number of instructions after the PBR operation that the paired branch occurs [7]. However, this approach is expected to require greater complexity for decoupled fetch-execute pipelines, particularly for implementations in explicitly parallel architectures with compressed instruction formats, where the size of an instruction is unknown prior to decoding. The remainder of this section discusses the requirements for the proposed static branch prediction scheme using the least significant bits of the branch operation address for pairing.

3.1. Prepare-to-Branch Operation

The Prepare-to-Branch operation must meet two requirements. First, it must contain a field containing the last few address bits of its paired branch. This address may either be to the paired branch operation itself, or to a group of operations which contains only one branch. Typically, 3-5 bits will be sufficient for the size of this field.

Secondly, the PBR operation needs to be scheduled a certain number of cycles prior to the paired branch to enable the proper timing for fetching the expected branch target. This latency is defined by the processor implementation. Assuming an implementation with *e* execute

stages for the PBR operation, a wire delay of *w* cycles, *f* instruction fetch stages, and *d* decode and register fetch stages, the necessary distance, in the absence of stalls, is $e + w + f + d - 1$ cycles, as shown in Figure 2. Current processors typically have 1-2 fetches stages, 1 PBR execute stage, 2-3 decode stages, and 0 cycles for wire delay, achieving a latency of 3-5 cycles. However, as clock frequencies continue to increase, wire delay becomes more prominent, and this latency is expected to increase to 5-7 cycles.

Scheduling for this latency depends upon the type of architecture being used. In explicitly parallel architectures, only one parallel instruction may be issued per cycle, so the latency can be achieved by scheduling the PBR operation the necessary number of parallel instructions before the paired branch. In superscalar processors, instructions may be reordered at run-time so it can be difficult to predict the latency between the execution times of any two instructions. The latency can only be assured by scheduling the PBR operation early enough such that the dependency chains and/or resource dependencies of all operations between the PBR and branch operation enforce the desired latency between the PBR operation and paired branch.

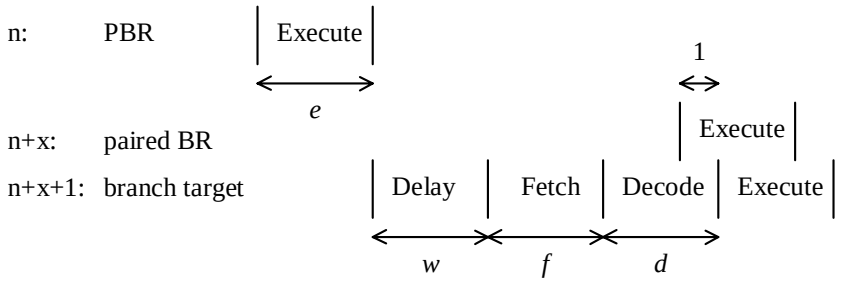

Figure 2 – Minimum latency between a PBR operation and its paired branch is $x = e + w + f + d - 1$ cycles

3.2. Searching for Paired Branch

When a Prepare-to-Branch operation occurs, it is necessary to determine whether the paired branch operation is available before redirecting instruction fetch to the expected branch target. Depending upon when the paired branch was scheduled with respect to the PBR operation, and whether any stalls have occurred, the processing of the paired branch operation may be in a number of possible stages, including:

- in the execution pipeline

- in the instruction buffers
- being fetched by the instruction cache
- not yet being fetched

The process of determining the location of a paired branch therefore requires a means to search for the paired branch in each pipeline stage (prior to the PBR execute stage), instruction buffer, and instruction cache fetch stage. Furthermore, since only a few of the least significant address bits of the paired branch are being used, enough information must be available to determine which instruction is the actual paired branch, in the event that multiple matches are found. Once the location of the paired branch is determined, the appropriate actions must be taken.

If the paired branch is in any of the first three locations, then the paired branch operation is available and the fetch of the expected branch target may begin. However, additional sequential operations following the paired branch may also be available in the pipeline, instruction buffers, and instruction cache. As instruction fetch is being redirected by the PBR operation, all sequential operations after the paired branch in the pipeline, instruction buffers, and instruction cache must be squashed. In the case of the instruction cache, this requires some mechanism to invalidate the offending sequential operations after they have been fetched.

If the paired branch is not yet being fetched, then the paired branch is not available and the PBR operation must wait until the fetch request for the paired branch has been issued. Only after the instruction cache begins fetching at the address of the paired branch may the PBR operation redirect the fetch unit to fetch the expected branch target. While waiting, the PBR operation and all information associated with it must be held in a state register.

3.3. Mispredicted Branches

Support for mispredicted branches is similar though less complicated than the Prepare-to-Branch support. Like the Prepare-to-Branch operation, occurrence of a mispredicted branch must also redirect instruction fetch and invalidate all operations in the pipeline. However, there is no need to search or wait for a paired branch. A misprediction simply invalidates all instruction buffers, all instruction cache accesses, and all sequential operations after the mispredicted branch in the execution pipeline, and immediately redirects fetch to the correct branch target. Again, in the case of the instruction cache, it will be necessary to invalidate the fetched instructions after they have been fetched.

For mispredicted branches, two types of misprediction can occur. The first is a branch that was predicted not taken, but was actually taken. In this case, the address of the actual branch target is specified in the branch operation, so it can be provided to the fetch unit for fetching the correct branch target. The other type of mispredicted branch is a branch that was predicted taken, but was actually not taken. In this case, the address of the actual branch target is not specified in the branch operation, but is the address of the operation sequentially following the branch. Obtaining the address for fetching the correct branch target in this case requires either: a) generating the sequential address of the correct branch target from the mispredicted branch address, or b) storing the sequential branch target address in a state register after the branch operation is decoded earlier in the pipeline. While the former option seems like the better method, the latter approach may be more efficient for explicitly parallel architectures that use a compressed instruction format, as will be seen in section 4.

3.4. Fetch Control

The fetch control unit directs the instruction cache when to perform an instruction fetch and from what location. Essentially, the fetch unit arbitrates between the various events which request instruction fetches and then issues the appropriate fetch requests to the instruction cache when it is not busy. There are five events that can request an instruction. Listed in order of increasing priority, they are:

- instruction buffers request sequential instruction fetch
- prepare-to-branch operation requests expected branch target
- branch predicted not taken is mispredicted and requests non-sequential branch target
- branch predicted taken is mispredicted and requests sequential branch target
- exception or reset requests exception handler

The most common and lowest priority event is the request for a sequential instruction fetch. In this case, the instruction buffers have emptied to the point that they can accept more instruction data and so request the next group of sequential instructions.

When implementing fetch control, if the instruction cache is busy when a fetch request occurs, the fetch request must be stored in a state register which indicates the instruction address for the next fetch. If multiple fetch requests are logged to this register before the instruction cache becomes available, only the highest priority fetch request needs to be saved.

3.5. Instruction Buffers

The purpose of the queue of instruction buffers is to decouple instruction fetch from the execution pipeline. The number of instruction buffers necessary for accomplishing this is dependent upon the architecture. Any implementation of the queue should support standard queue attributes, advancing the queue whenever the head of the queue empties, and writing instruction cache data only to the first empty instruction buffer at the tail of the queue.

With respect to the static branch prediction method described herein, the instruction buffer queue's only responsibility is to issue a fetch request for the next sequential group of instructions when a sufficient number of entries becomes available. The amount of room necessary before a fetch request is made must also account for the space required for any currently outstanding fetch requests. For example, in a 2-cycle instruction cache, up to two instruction fetches may currently be outstanding in the cache, so a request should only be made if at least three buffers are empty.

4. Implementation

The proposed static branch prediction method for decoupled fetch-execute architectures was implemented on an EPIC architecture. The architecture explicitly describes the instructions that are to be issued in parallel as a single long instruction word. The parallel instructions are defined by a compressed instruction format, which uses stop bits after every operation to delineate the parallel operations. A stop bit asserted by an operation indicates that all the instructions between it and the previous operation with an asserted stop bit are to execute in parallel. To allow additional space in the operation format both for the stop bit, and predication as well, three operations are packed into a *bundle* of 128 bits, instead of only 32 bits per operation.

Our implementation assumes an EPIC architecture capable of issuing a long instruction of up to six operations per cycle. Each group of explicitly parallel instructions is called a *package*, and each package may contain between zero and six operations, where the case of zero instructions is defined as a null operation (NOP) followed by a stop bit.

For simplicity in the initial implementation, a limitation was placed upon which operations could be used as branch targets. Defining a pair of bundles aligned on a 32-byte boundary as a *double-bundle*, branching to non-sequential targets is only allowed to the beginning of a double-bundle. This restriction is not a necessary condition for the static branch

prediction method, or even this implementation. Branching to the beginning of every bundle could easily be allowed with only minor logic changes.

One interesting aspect of the target instruction format with respect to this implementation is that not all operations are directly addressable by memory addresses. Instead the address of an operation is characterized by the address of its bundle and its position (either position 0, 1, or 2) within the bundle. Therefore, correlating the PBR operation and the branch operation via the last few bits of the branch operation will not work. Instead, only the address of the double-bundle containing the paired branch is used for this correlation, with the restriction that only one branch operation be placed in a double-bundle containing a paired branch. Again, this restriction could be relaxed to restrict the number of branches to one per in bundles containing a paired branch bundle (instead of one per double-bundle), with a minimum of extra logic.

4.1. Decoupled Fetch-Execute Engine

Figure 3 gives an overview of the decoupled fetch-execute architecture.. This architecture embodies a typical instruction buffer architecture with a few additions to guarantee proper support and fetch timing for the Prepare-to-Branch operation. As seen in the diagram, the design assumes a queue of six instruction buffers, each containing a double-bundle of six operations. The fetch control logic uses a *Fetch Address Table* and *Next Fetch Register* to keep track of all outstanding instruction fetches and the next fetch request, respectively. *The Branch Sequential Table* stores the addresses of the next sequential package after every paired branch. The *PBR Pending Table* maintains a listing of pending PBR operations, and the *Paired Branch Found?* circuit is used to search for the paired branch operation and invalidate all sequential operations following the branch. It works in conjunction with the *Pipeline Address Table, Fetch Address Table*, and instruction buffer addresses to search in the execution pipeline, instruction cache, and instruction buffers, respectively. Finally, the *Shift Amount Control* and *Package Selection* logic are circuits specific to the target architecture for extracting the next package from the first two instruction buffers. The *Decoding Paired Branch?* circuit is required for checking for the existence of the paired branch in the package currently being decoded, and determining whether to store the next sequential package address in the Branch Sequential Table register. Each of these units will be discussed in more detail below.

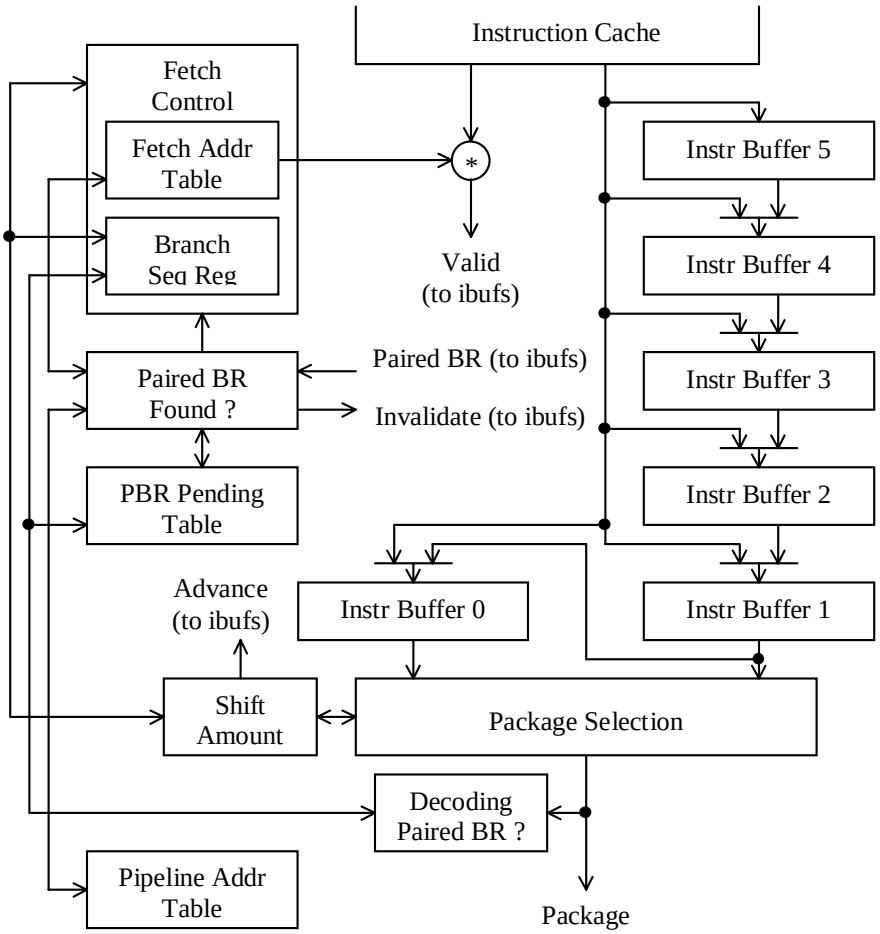

Figure 3 - Decoupled fetch-execute architecture with static branch prediction support

4.2. Instruction Buffer

Each instruction buffer contains four fields, as shown in Figure 4: the contents of a double-bundle, the address of that double-bundle, a valid bit, and an *on-path* bit. To maintain proper queue implementation, the instruction buffer is organized as a FIFO. The *Instruction Buffer Contains Paired Branch?* circuit compares the last few double-bundle address bits against the address of the paired branch for a pending PBR operation. The valid bit indicates if the current contents are valid.

The on-path bit indicates that a previous pending PBR operation checked this double-bundle for a paired branch and found its paired branch later in the instruction buffer or instruction cache, so this double-bundle lies on the correct path of execution. Consequently, when the on-path bit is set (it is initially reset when fetch is initiated in the instruction cache), the

search for a paired branch will always return negative. This ensures that multiple matches will not be found for the paired branch operation.

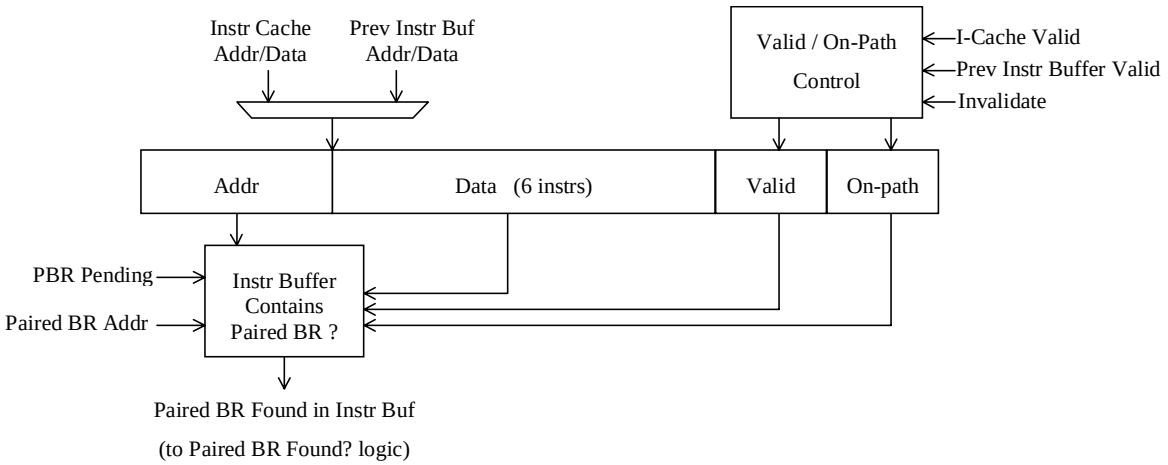

Figure 4 – Instruction Buffer

4.2. PBR Pending Table

The PBR Pending Table implements a queue that maintains the status of all pending Prepare-to-Branch operations. As seen in Figure 5, the two primary fields in the PBR Pending Table contain the address of the expected branch target and the last few bits of the address of the double-bundle containing the paired branch. In addition, there are two bits which indicate the status of the pending PBRs. The first status bit, *pending* is set when a PBR operation executes and is added to the PBR Pending Table, and remains set until the paired branch leaves the instruction buffers. As the paired branch leaves the buffers via the Package Selection logic, it is recognized by the *Decoding Paired Branch?* circuit, which clears the pending bit, advances the PBR Pending Table, and writes the next sequential package address to the *Branch Sequential Table*.

As mentioned in section 3.4, a pending PBR operation may be in one of two states. It is either searching for its paired branch operation, or has found the paired branch. This is indicated by the *found* bit. The found bit is reset when a new PBR operation executes and is added to the PBR Pending Table. While the found bit is deasserted, the pending PBR sends the address of its paired branch to the *Paired Branch Found?* circuit, which searches the pipeline, instruction buffers, and instruction cache each cycle until the paired branch is found. Once found, the found bit is asserted, and the Paired Branch Found? circuit invalidates any sequential operations following the paired branch, sets the on-path bit in the remaining valid operations prior to and

including the paired branch, and a fetch request for the expected branch target to the Fetch Control unit.

The PBR Pending Table is organized as a queue to avoid incorrectly searching for paired branch operations. The predicted direction of the branches defines a predicted path of execution. PBR operations must be occur in the same order as their paired branches so that each PBR operation will only search for its paired branch in the appropriate section of the predicted path. The queue arrangement accommodates this requirement, handling PBR operations one at a time so paired branches are searched for in order, ignoring operations already recognized as on-path.

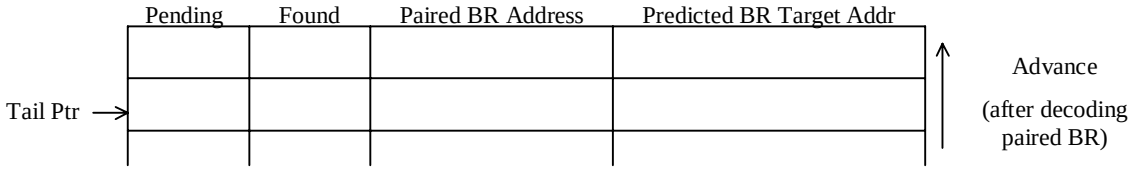

Figure 5 – PBR Pending Table (implemented as a queue)

4.3. Fetch Control

The Fetch Control unit, as shown in Figure 6, arbitrates among the various fetch requests, keeps track of the addresses and validity for all outstanding fetches and stores the sequential address after a paired branch. *The Fetch Address Table* serves the dual purpose of indicating whether the instruction cache is available to receive another fetch request, as well as providing a history of all outstanding instruction fetches. The Fetch Address Table also has a valid bit and an on-path bit for each entry which indicates whether the fetched data will be valid, and whether its operations may be searched for paired branches. This valid bit determines the validity of data arriving from the instruction cache. It is initially set, but may be deasserted by a mispredicted branch or by a pending PBR operation after a paired branch is found. The on-path bit is initially reset, and is only set if it contains operations along the path prior to a paired branch.

The fetch control unit also contains a *Next Fetch Register*, which holds the address and priority of the next instruction to be fetched when the instruction cache becomes available. The Next Fetch Register contains three bits which indicate the priority level of the fetch request, as described in section 3.4. One bit is used for each request type: PBR operation requests, mispredicted branch requests, and exception/reset requests. When no bits are set, a sequential fetch request is indicated.

The fetch control unit also contains the *Branch Sequential Table*, which is a queue that stores the addresses (double-bundle address and Shift Amount) of the next sequential package after each paired branch. If a paired branch is mispredicted, it must return to its sequential branch target, which is available here. Because a sequential branch target is need not be at the beginning of a double-bundle, the Shift Amount (described in section 4.5), indicating the beginning of the package in the double-bundle, is also contained in the Branch Sequential Table. The number of entries in the Branch Sequential Table must equal the number of possible pending PBR operations (the size of the PBR Pending Table).

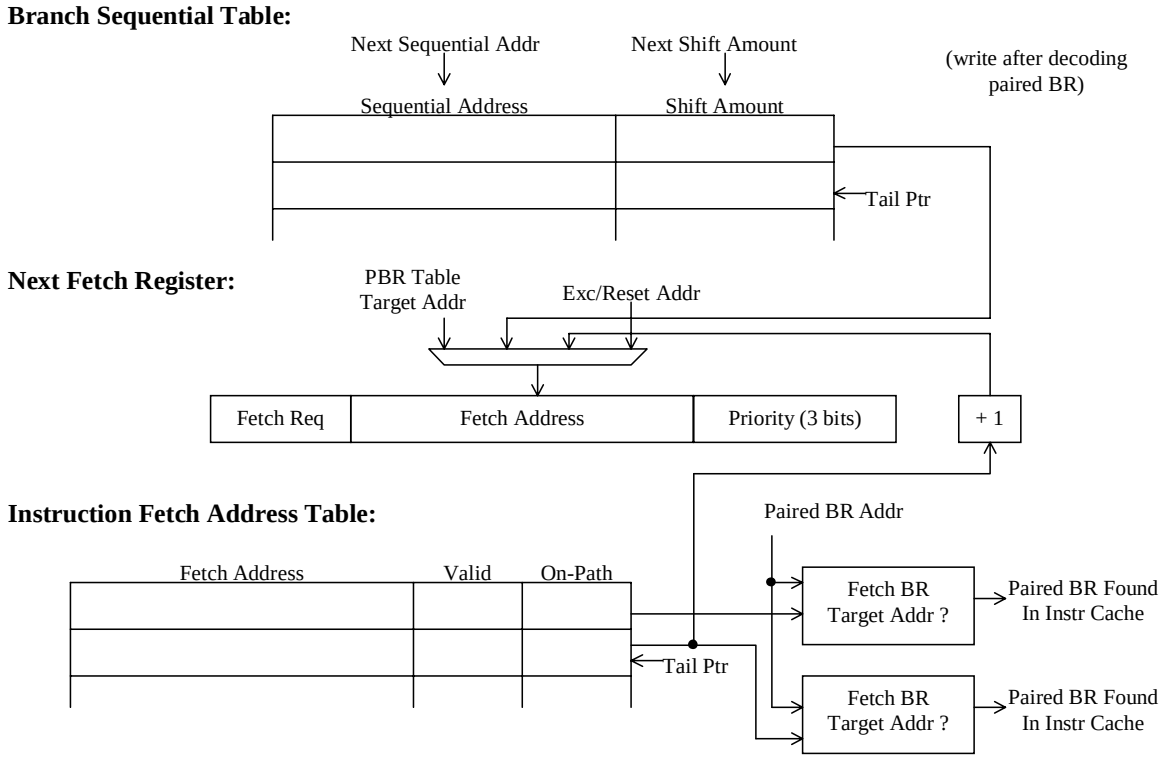

Figure 6 – Fetch Control Unit

4.4. Paired Branch Found?

It is necessary to search the instruction cache, instruction buffers, and pipeline for the paired branch corresponding to a pending PBR operation. For instruction cache, the *Fetch Address Table* keeps an ordered listing of the outstanding fetches currently being performed. For the instruction buffers, each instruction buffer has its own address field. And in the pipeline, the *Pipeline Address Table* (or similar hardware that provides the addresses and validity of packages

in the pipeline) maintains the addresses for packages in the pipeline[2]. The last few address bits of the paired branch corresponding to the first pending PBR operation (whose found bit is not set) is compared against each of these addresses. If the addresses match, and the on-path bit for that operation or package is not set, then the paired branch is found. The Paired Branch Found? circuit communicates this to the fetch unit, instruction buffers, and pipeline, invalidating all sequential operations after the branch and setting the on-path bit for all operations prior to and including the paired branch. Additionally, a request is sent for fetching the expected branch target address, whose address is provided by the PBR Pending Table.

4.5. Package Selection

Because the target architecture does not provide unique addresses for each operation, it is necessary to maintain a pointer for extracting packages from the double-bundles. The pointer, referred to as *Shift Amount*, points to the first operation in the package currently being extracted. The Shift Amount pointer is sent to six multiplexers which select the next six operations in the first two double-bundles. The stop bits of these six instructions are then evaluated to determine which instructions actually belong to the package. All instructions up to and including the instruction with the first asserted stop bit belong to the package. The Shift Amount pointer is then adjusted to point to the next operation after the last instruction in this package. In the event that the first operation of the next package is in the second instruction buffer, not the first, an *Advance* signal is asserted which advances the entire instruction buffer queue.

[2] In in-order processors, Pipeline Address Table is unnecessary if scheduling ensures the paired branch is not in the execution pipeline when the PBR operation executes.

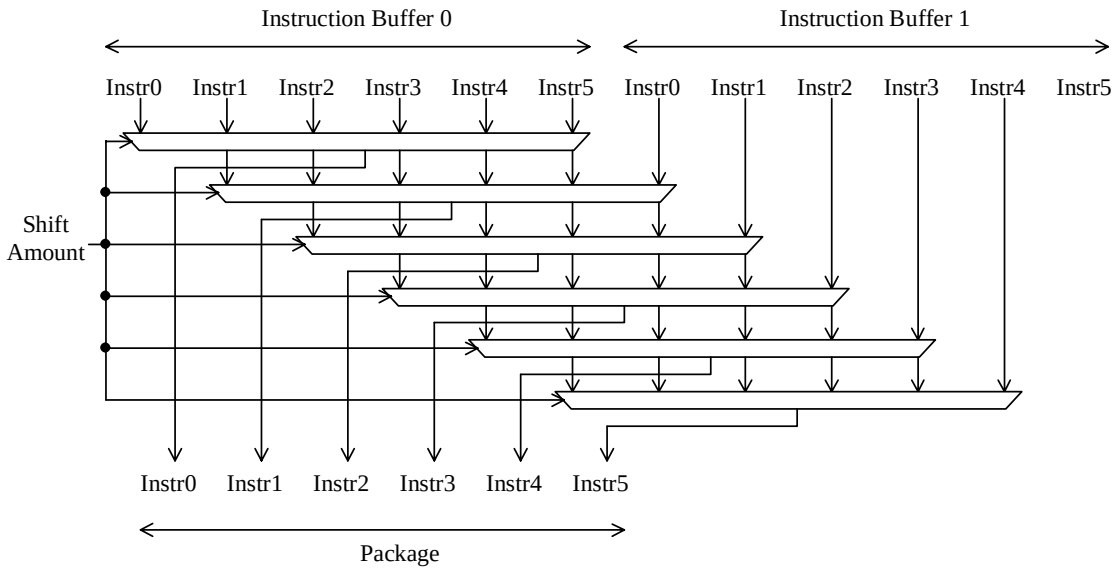

Figure 7 – Package Selection

4.6. Summary of PBR Execution

To summarize, when a PBR operation executes, the PBR operation is written into the first empty spot in the *PBR Pending Table*, setting the pending bit and resetting the found bit. Then it sends the (last few bits of the) address of its paired branch to the Paired Branch Found? circuit, which searches the pipeline (via the *Pipeline Address Table*), instruction buffers (via the *Instruction Buffer Contains Paired Branch?* circuit in each buffer), and instruction cache (via the *Fetch Address Table*). Once the paired branch is found, all sequential operations following the branch are invalidated, all operations prior to and including the branch have their on-path bits set, and a fetch request for the expected branch target is sent to the fetch control unit. The fetch control unit writes the address of the expected branch target into the *Next Fetch Register*, and the instruction cache begins fetching it, assuming no higher priority fetch requests are pending. When the instruction data is returned by the cache, the paired branch and expected branch target flow through the instruction buffers until the paired branch is decoded during *Package Selection* and clears the pending PBR operation from the PBR Pending Table.

For mispredicted branches, the process is similar, though all entries in the instruction cache, instruction buffers, and pipeline are invalidated. The *Next Fetch Register* is then set with the actual branch target. For branches that were initially predicted not taken, the correct branch

target address is provided in the branch operation. Otherwise, for branches that were initially predicted taken, the sequential branch target address is provided by the *Branch Sequential Table*.

5. Results

To determine correctness and feasibility of this design in high performance processors, the implementation of the decoupled fetch-execute engine described in section 4 was modeled in VHDL. The VHDL code was simulated for correctness and verified that the proposed method efficiently supports static branch prediction for architectures with decoupled fetch and execute pipelines. While the initial implementation only allowed for one pending PBR operation, the additional logic for multiple pending PBR operations should be minimal.

Synthesis of the VHDL model was used to examine the relative feasibility of this method in existing and future processors. The target process for synthesis was IBM's SA-12 CMOS standard cell ASIC technology. This technology features a 0.25μm lithography with a 0.18μm effective channel length, 2.5V supply voltage, and 5 levels of metal wiring. Performance for a 2-way NAND gate with fanout of 2 and 0.5mm wire is nominally 67ps.

Preliminary synthesis results give a total of less than 13,000 logic gates, including 1830 registers. Static timing analysis reveals a maximum path length of 3.85ns (260 MHz). With further tuning and logic optimization, the timing can be expected to improve significantly. Furthermore, it should be noted that full custom design, as would be employed in an actual implementation of a processor incorporating this design, is typically capable of factors of 2 to 3 times greater speed than an ASIC implementation. It is therefore expected that operation in the gigahertz range is feasible with the current technology. The National Semiconductor Roadmap [10] forecasts clock frequencies for high performance processors, currently up to 1.25 GHz, as reaching 2.1 GHz by 2003. The implementation described here is consistent with such performance levels.

6. Conclusions

We have presented a method for supporting static branch prediction on architectures that decouple the instruction fetch from the execution pipeline. Use of decoupled fetch-execute architectures represents an effective way to minimize the penalties due to instruction cache misses. However, existing static branch prediction techniques do not work well with such architectures. A new method is proposed for static branch prediction using the Prepare-to-

Branch (PBR) method, which adds a field in the PBR operation specifying the last few address bits of the corresponding branch operation. Use of the branch operation's address defines a method for pairing the two operations without requiring access to the contents of the branch operation. It is therefore possible to determine whether the paired branch operation is in the process of being fetched before the branch is even available. This enables the architecture to start fetching the predicted branch target immediately after fetch begins for the paired branch.

Using this PBR prediction scheme, the report describes the requirements for implementation on a decoupled fetch-execute architecture and enabling proper timing between the branch operation and its predicted target. An implementation for an EPIC architecture is presented, simulated for correctness, and synthesized to a 0.25μm ASIC standard cell technology. Initial cycle time estimations indicate a 3.85 ns clock cycle, corresponding to a 260 MHz design. These early results indicate excellent performance. As the technology for this new method matures, operating frequencies in full custom processor implementations are expected to reach well into the GigaHertz range.

Bibliography

[1] B. Calder, D. Grunwald, and J. Emer, "A System Level Perspective on Branch Architecture Performance," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989, pp. 199-206.

[2] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, May 1989, pp. 224-233.

[3] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors," *SPIE Photonics West, Media Processors '99*, San Jose, CA, Jan. 1999.

[4] K. Ebcioglu, E. R. Altman, S. Sathaye, and M. Gschwind, "Execution-based Scheduling for VLIW Architectures," *Euro-Par '99*, Toulouse, France, Sept. 1999.

[5] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of the 8th Annual Symposium on Computer Architecture*, April 1981.

[6] V. Kathail, M. Schlansker, B. R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," *HPL-93-80*, February 1994.

[7] H. C. Young and J. R. Goodman, "A simulation study of architectural data queues and prepare-to-branch instruction," *Proceedings of the IEEE International Conference on Computer Design '84*, Port Chester, NY, 1984, pp. 544-549.

[8] C. Young and M. Smith, "Improving the accuracy of static branch prediction using branch correlation," Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[9] N. Gloy, M. D. Smith, and C. Young, "Performance issues in correlated branch prediction schemes," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, Michigan, November, 1995, pp. 3-14.

[10] "*The National Technology Roadmap for Semiconductors*", 1997 Edition, Semiconductor Industry Association, 1997.