POOR MAN'S TRACE CACHE: A VARIABLE DELAY SLOT ARCHITECTURE

Tino C. Moore
Michigan Technological University, tinom@mtu.edu

POOR MAN'S TRACE CACHE: A VARIABLE DELAY SLOT ARCHITECTURE

By

Tino C. Moore

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2022

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Advisor:	*Dr. Soner Onder*
Committee Member:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Jianhui Yue*
Committee Member:	*Dr. David Whalley*
Department Chair:	*Dr. Andy Duan*

Contents

List of Figures

List of Algorithms

List of Tables

List of Abbreviations

BTAC	Branch Target Address Cache
BTIC	Branch Target Instruction Cache
BTB	Branch Target Buffer
CLB	Cache Line Boundary
CFG	Control Flow Graph
CTI	Control Transfer Instruction
DCTI	Direct Control Transfer Instruction
EDDP	Energy Delay^2 Product
EX	Execute Processor Pipeline Stage
GAg	Two-Level Adaptive Branch Predictor Using a Global History Register and a Global Pattern History Table
GSHARE	Global History with Index Sharing Branch Predictor
I-Cache	Instruction Cache
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
L1	Level 1 Cache
LD	Load Instruction/Operation
MEM	Memory Processor Pipeline Stage

MPKI	Mispredictions Per Thousand Instructions
MUX	Multiplexor
NT	Not Taken Branch
PC	Program Counter
PMTC	Poor Man's Trace Cache
RAS	Return Address Stack
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
SB	Store Buffer
SPEC	Standard Performance Evaluation Corporation
ST	Store Instruction/Operation
STC	Software Trace Cache
TAGE	Partially Tagged Geometric History Length Branch Predictor
TC	Trace Cache
VLIW	Very Long Instruction Word

Abstract

We introduce a novel fetch architecture called Poor Man's Trace Cache (PMTC). PMTC constructs taken-path instruction traces via instruction replication in static code and inserts them after unconditional direct and select conditional direct control transfer instructions. These traces extend to the end of the cache line. Since available space for trace insertion may vary by the position of the control transfer instruction within the line, we refer to these fetch slots as *variable delay slots*. This approach ensures traces are fetched along with the control transfer instruction that initiated the trace. Branch, jump and return instruction semantics as well as the fetch unit are modified to utilize traces in delay slots. PMTC yields the following benefits:

1. Average fetch bandwidth increases as the front end can fetch across taken control transfer instructions in a single cycle.

2. The dynamic number of instruction cache lines fetched by the processor is reduced as multiple non contiguous basic blocks along a given path are encountered in one fetch cycle.

3. Replication of a branch instruction along multiple paths provides path separability for branches, which positively impacts branch prediction accuracy.

PMTC mechanism requires minimal modifications to the processor's fetch unit and the trace insertion algorithm can easily be implemented within the assembler without compiler support.

Chapter 1

Introduction

High performance superscalar processors feature wide issue widths. To reap the benefits of increased issue width and execution resources, the front end must be able to provide comparable instruction fetch width. As the issue width increases, it becomes more challenging for the front end to maintain an average fetch width of correct path instructions near the issue width. For an 8 issue superscalar, an average fetch width of 4 instructions is typical in integer code. The fetch limitation is due to the relatively short average depth of basic blocks in comparison to the issue width. Often the fetch width is limited by control transfer instructions. When a taken branch or jump is encountered, the remaining instructions in the fetch buffer are along the wrong path and not valid for decoding. To increase the average effective fetch width we propose a novel technique, called Poor Man's Trace Cache (PMTC), to rescue the wasted fetch slots when direct control transfer instructions (DCTI) are encountered. Rescuing the fetch slots can be done by attaching a variable sized delay slot to a DCTI where the target is known at compile time. The delay slot extends from the DCTI to the next cache line boundary. The delay slot can be filled with a trace consisting of a block of instructions copied from the target. As a result, whenever the DCTI is located before a cache block boundary, instructions alongside the taken path, including additional branch instructions can be inserted into these unused fetch slots. In some cases the trace can be a combination of instructions across several targets that originally span multiple cache lines. Delay slots can be safely added to every unconditional jump instruction and selectively applied to statically biased taken branches controlling loops. Hence, the proposed technique takes advantage of the static taken bias of these instructions.

Instruction fetch is limited by the accuracy of branch prediction resources. Fetched instructions must be along the correct path or else the processor will incur an additional penalty to remove the wrong path instructions. Instruction fetch techniques that seek to increase fetch bandwidth must also seek to improve branch prediction accuracy to effectively use the increase in bandwidth. Techniques that solely increase fetch bandwidth simply result in more instructions being discarded by the processor.

PMTC has a unique benefit of occasionally compressing the fetch depth for a chain of control transfer instructions. Control transfer chain reduction happens when another control transfer instruction is located within the trace. Chain reduction removes the need to fetch the intermediate target cache line. When the cache line size is greater than the average basic block size this happens frequently. Copying branches into multiple traces also provides path separability for those branches which can be utilized by the branch predictor and branch target buffer (BTB). Some hard to predict control transfer becomes easy to predict as the path separability is made explicit in the static program.

PMTC presents a novel way to address the fetch problem since it improves *both the fetch bandwidth and the branch prediction accuracy.* Relevant micro-architecture background is presented in Chapter 2. Fetch problem characteristics are described in Chapter 3. The following Chapters describe the trace insertion algorithm and its implementation in an 8 issue superscalar. The simple assembly modification algorithm described in Chapter 4 in tandem with very simple micro-architecture modifications discussed in Chapter 5 provide these benefits. Performance and implications that the static traces have on dynamic program execution are examined in Chapter 6. Performance is analyzed using integer and floating point benchmarks from SPEC 06. Earlier dynamic and static techniques dealing with the fetch problem across taken control transfers, and the conclusions, are described in Chapter 7.

Chapter 2

Background

2.1 Superscalar Micro-Architecture

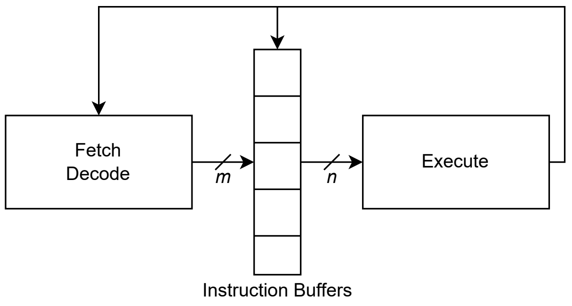


Figure 2.1: Decoupled Execution Model

Superscalar processor micro-architecture is typically utilized in state of the art general purpose applications. Superscalar processors implement a decoupled fetch execute architecture as shown in Figure 2.1 [58].

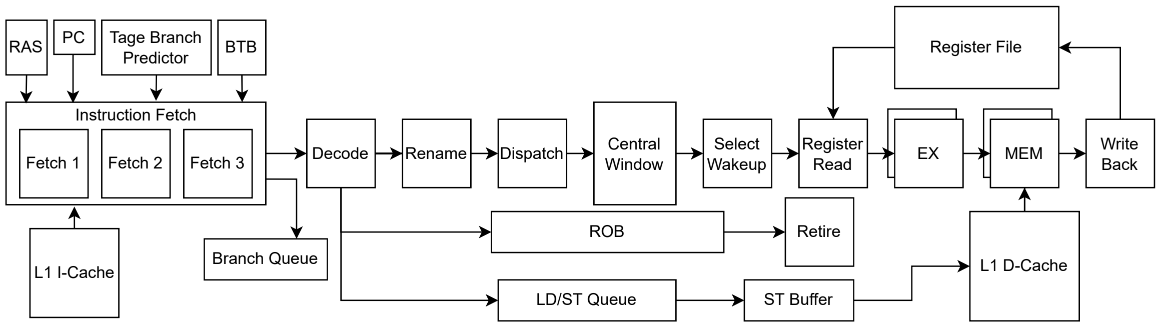


Figure 2.2: Superscalar Processor Pipeline

Instructions are fetched and issued into instruction storage. In a decoupled fashion instructions are then dispatched from storage to the execution hardware. The term superscalar typically implies a set of micro-architectural characteristics including in-order issue, out-of-order execution, speculative execution, wide instruction pipelines and in-order instruction retirement. Typical superscalar processors employ deeper pipelines similar to that shown in Figure 2.2. The objective of a superscalar processor is to dynamically maximize exploitable Instruction Level Parallelism (ILP).

Typical superscalar processor characteristics are described in more detail in the following sections.

2.1.1 Instruction Fetch

Instructions are fetched in-order from the static program text. The fetch width, which varies by design, spans multiple instructions. It is typical for the front end to read two instruction cache lines during the fetch process allowing instruction delivery from a basic block that spans more than one cache line. The fetch process may consist of multiple pipeline stages.

2.1.2 Control Speculation

Superscalar processors employ control speculation to allow instruction fetch and execution to continue while branch outcomes are yet unresolved. Control speculation hardware is accessed during the fetch process. Control speculation is split into two components, namely target prediction and outcome prediction. Target prediction is typically handled by a Branch Target Buffer (BTB). This structure identifies any control transfer instructions among the fetched instructions. The BTB also distinguishes between control transfer type including direct or indirect, conditional or unconditional and if the instruction is a return. The branch predictor is accessed for any conditional control transfer instructions identified by the BTB and provides the outcome prediction. When a return is detected by the BTB, the Return Address Stack (RAS) is popped to obtain the destination address of the return. Figure 2.2 shows the integration of the branch predictor, RAS and BTB with the processor front end.

2.1.3 Parallel Decode and Renaming

Superscalar processor architectures employ parallel decoding of the instructions provided by the fetch unit. The register file is typically larger than what is defined by the Instruction Set Architecture (ISA). Logical registers are mapped to physical registers using a reference table. Using this mapping, it is possible for the processor to dynamically eliminate anti and output dependencies, and expose additional ILP. The rename stage renames the destination registers by allocating a new physical register to each instruction and updates source register names based on the current mapping. Once the fetched instructions have been decoded and renamed, they are dispatched into an instruction buffer.

2.1.4 Out Of Order Dispatch and Execution

Instructions are dispatched from the instruction buffer when their input operands become available. Operands become available once producing instructions complete and the register file is updated with the result or the result is available on a bypass and forwarding network. In this fashion, instructions may be executed out of program order. Superscalar processors feature multiple execution pipelines for parallel execution of independent instructions. Execution units of various types are present.

There may be dedicated integer, floating point and memory address execution units.

2.1.5 Memory Speculation

Load instructions may be issued early, before prior store addresses have been determined, in order to dynamically shorten the load latency. To achieve this, superscalar processors typically employ some form of speculative disambiguation via an algorithm similar to *store sets* [10]. Using this algorithm loads are speculatively issued until conflicts with prior stores arise. Once a conflict arises, it is recorded within a *store set* to prevent future conflicts. Loads wait for all stores in their *store set* to complete before executing. When a memory order violation is detected, miss-speculation recovery is triggered similar to a control miss-speculation.

2.1.6 Preserving Sequential Consistency

When a miss-speculation occurs, the processor must recover the last valid in-order state to preserve sequential consistency and support precise exceptions. Superscalar processors typically utilize additional structures to assist in recovery. The Reorder Buffer (ROB) maintains information about every instruction in program order. When a miss-speculation occurs, older instructions in the ROB may be retired safely. Once retirement reaches the miss-speculated instruction the correct in-order state has been established. Instructions following the miss-speculation must be re-executed or discarded. An alternative to reorder buffer based design is to use checkpointing [21].

Memory contents must also preserve the in-order state to support precise exceptions. Speculative loads do not change the contents of memory. Speculative stores would change memory contents and break the sequential consistency. In order to prevent this, superscalar processors buffer store instruction payloads in a Store Buffer (SB) until the store instruction is at the head of the ROB, known to be non-speculative and ready to retire. Once the store retires, the data payload may leave the SB and be written to memory as it is now part of the in-order state.

2.2 Branch Prediction Mechanisms

Increasing pipeline depth negatively impacts performance in the presence of control transfer. Branch outcome and target address are required to direct the fetch unit without interrupting the instruction stream. However, as the pipeline depth increases, the latency for recognizing a branch, resolving the outcome and computing the target also increases.

Three pieces of information are required in order to correctly handle a transfer of control within the instruction stream: (1) whether an instruction is a control transfer instruction and its type, (2) outcome for a conditional transfer, and (3) the target address.

There are a number of ways control transfers can be handled.

1. The processor can do nothing until the target and outcome are known by stalling when a branch is detected. Stalling will insert a bubble, or useless instructions, into the instruction stream, but it will not trigger recovery.

2. Branch target address and branch outcome can be speculatively predicted. This will not alter the instruction stream, but will trigger recovery and require a subsequent penalty if the prediction is incorrect.

3. Statically remove or reduce the number of branches in the program text. This can be accomplished through predication of control dependent instructions, block structured ISA or by combining basic blocks into super blocks to reduce the total number of branches [35].

4. Multi-path execution. The processor can fetch and execute instructions along both paths of the branch, assuming the target is known, and discard the incorrect path instructions once the branch outcome becomes known. This approach requires branches and their targets to be identified by the front end

and adds complexity managing multiple instruction streams from different addresses. This approach is not always applicable nor optimal.

5. Switch to another thread. If the processor has multi-threading support, it can switch to a dormant thread to prevent wasted processor time. This approach does not hide the latency of the branch resolution with respect to the original thread, but it removes the need for speculation or recovery.

Branch prediction mechanisms seek to exploit repeated behaviour. Means of pattern recognition vary by predictor implementation, but typically exploit the following:

1. Static bias. Some conditional branches frequently produce the same outcome regardless of how this branch was reached.

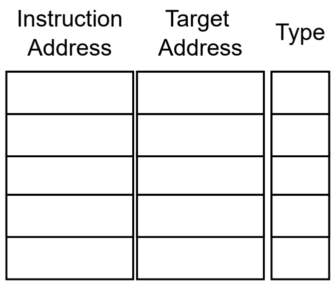
2. Local history. Some conditional branch outcomes are repetitive with respect to prior outcomes of this branch.

3. Global history. Some conditional branch outcomes may be correlated to prior outcomes of some or all control transfer instructions preceding this branch.

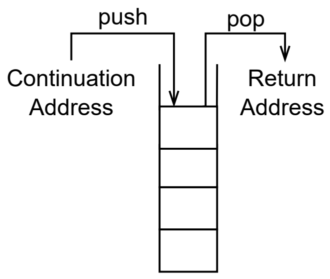
4. Data correlation. Similar to correlation with prior branch outcomes, some conditional branch outcomes can be correlated with a piece of data like a load or register value.

2.2.1 Branch Detection and Target Prediction

Branch detection and target prediction are typically handled and predicted separately from branch outcome. The typical method of target prediction utilizes a Branch Target Address Cache (BTAC) also known as a Branch Target Buffer (BTB) [30] [45] [66].



(a) Branch Target Buffer



(b) Return Address Stack

Figure 2.3: Branch Target Predictors

The BTB is a small cache dedicated to associating branch instruction addresses with target addresses. Figure 2.3a shows the general organization of a BTB. BTB entries may be extended to contain additional information about a branch such as its type. In practice the BTB is implemented in an n-way interleaved organization to allow parallel access of n entries. The interleaving width n typically matches the fetch width of the processor such that the BTB can be accessed for every instruction fetched in a given cycle. The BTB relies only on the instruction addresses and can be accessed as soon as the instruction addresses are known. While the BTB can handle returns, a separate structure is typically employed to increase performance. The Return Address Stack (RAS) as shown in Figure 2.3b is responsible for predicting the target address when a return is detected [28]. The RAS is a first-in-last-out stack structure. Continuation addresses are pushed onto the stack whenever a call is executed. When executing a return, the target address can be obtained by popping the head element of the stack. The stack organization is effective due to the typical ordered nature of return targets. Stack depth is normally determined by the typical maximum call depth.

2.2.2 Predicting Branch Outcomes

Taxonomies of branch prediction mechanisms typically organize predictors based on how they are implemented to exploit pattern repetition. In this work we take a simpler approach to classify prediction mechanisms based on what data is used as input.

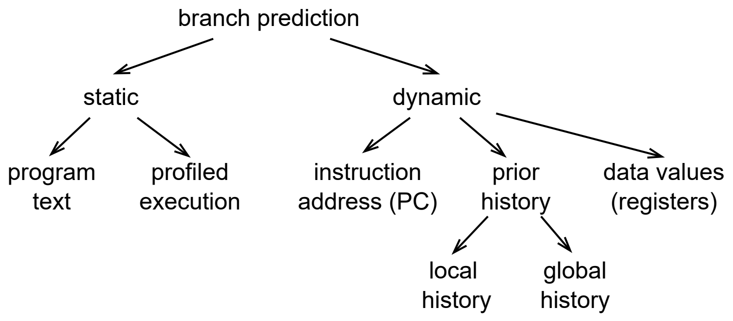


Figure 2.4: Branch Predictor Input Taxonomy

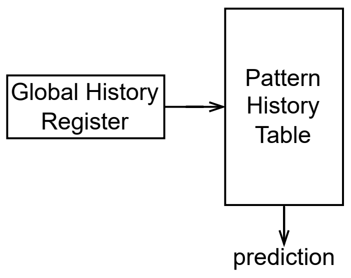
Figure 2.4 shows a taxonomy of inputs used by most branch predictors. The taxonomy is split at the top level between static and dynamic inputs. Static inputs are available at compile time where as dynamic inputs become available as the program is executing. Static mechanisms can exploit static analysis of the program text, or additionally use recorded outputs from profiled execution, at compile time to annotate or modify the program text with suggestions of expected branch outcomes. Dynamic inputs may include the instruction address, or Program Counter (PC), data values currently held within architectural registers, and a history of prior outcomes. Prior history can be further divided depending on whether the history only considers the current branch or extends to cover the outcomes of previous control transfers.

Dynamic prediction mechanisms may utilize a combination of inputs spanning multiple leaves within the dynamic side of the taxonomy. Combination of inputs may be used together to exploit correlation or instead independently used in parallel to provide multiple outcomes from which the predictor may select between. Dynamic predictor implementations vary drastically across the corpus of branch prediction works. Figure 2.5 shows an oversimplified implementation of three select predictors to illustrate classification within the taxonomy proposed in Figure 2.4.

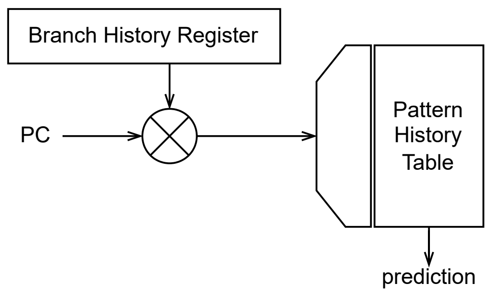
Figure 2.5a shows the organization of the *Two-Level Adaptive Branch Prediction Using a Global History Register and a Global Pattern History Table* (GAg) [72]. GAg implements a global history register and pattern history table. The contents of the history register, updated with the outcome of each branch, are used to directly index the pattern history table and provide a prediction. With respect to the taxonomy, GAg uses only the global history dynamic input.

Figure 2.5b shows the organization of the *Global History with Index Sharing* predictor (GSHARE) [34]. GSHARE is similar to GAg in that it uses the global history and a shared pattern history table. GSHARE deviates from GAg in the indexing method. Rather than directly indexing the shared pattern history table using the contents of the global history register, GSHARE uses a hashed index generated by taking the bit-wise exclusive-or of the global history with the branch's instruction address. Hashing the instruction address with the global history allows a single branch to occupy different pattern history table indices depending on the prior branch outcomes similar to GAg, but tries to better capture and exploit the effect of correlation by including the instruction address. Relative to the input taxonomy, GSHARE uses both the instruction address and global history dynamic inputs.

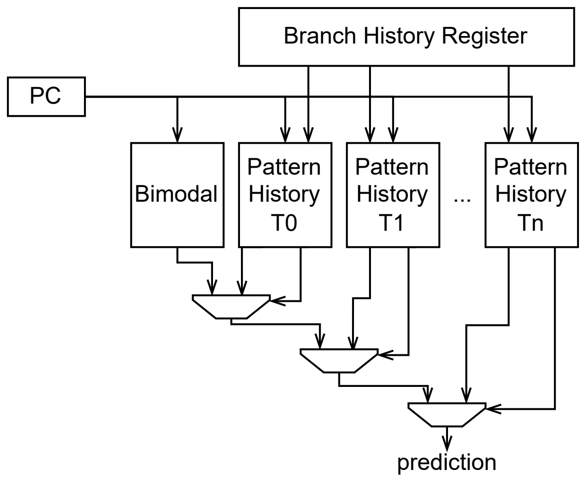
Figure 2.5c shows the organization of the *Partially Tagged Geometric History Length Branch Predictor* (TAGE) [64] [60] [61]. TAGE accesses multiple predictor tables in parallel using different history lengths which form a geometric series. TAGE then uses a selection mechanism to select which table output to use as the final prediction. Pattern history table entries are tagged to ensure the indexed entry corresponds with the branch being predicted to increase confidence and avoid aliasing to a prediction



(a) GAg



(b) GSHARE



(c) TAGE

Figure 2.5: Branch Predictor Implementations

that does not match the current branch. With regard to the input taxonomy, TAGE uses the dynamic inputs instruction address and global history, where the history is subdivided into multiple components. TAGE is regarded as state of the art in single branch prediction.

Work has explored augmenting branch prediction with additional techniques targeting hard to predict low confidence branches using branch prediction inversion [33] and reversal via data correlation [1].

2.2.3 Multiple Prediction

Some superscalar processor front end designs such as the Trace Cache (TC), described in Section 2.3, require multiple branches to be predicted at once. In some cases it is possible to extend the single branch prediction mechanisms described in Section 2.2.2 to provide multiple predictions via structural replication or adding additional ports. In general multiple branch prediction mechanisms are organized and updated differently.

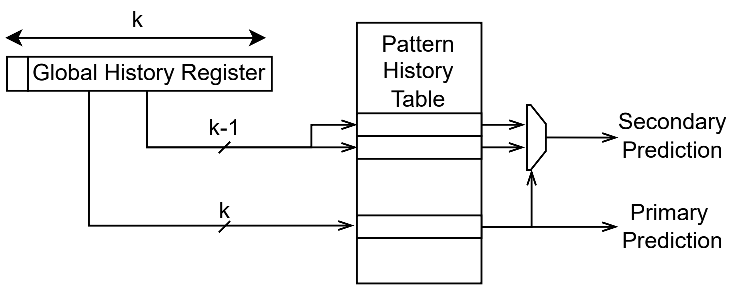


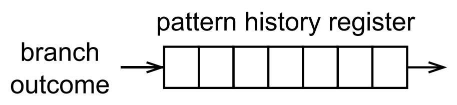
Figure 2.6: Multiple Branch Predictor

Figure 2.6 shows the multiple branch predictor proposed in [71]. In this design the full *k* bits of the global history are used to index the primary prediction. *k-1* bits of the history register are used as the most significant bits to index two entries in the pattern history table. The primary prediction is used as the least significant bit of the secondary indexing and drives a multiplexer (MUX) to select the final value of the secondary prediction. This effectively adjusts the branch history to assume the primary prediction was already contained in the history register. The multiple prediction mechanism shown can be extended further to any number of branches at the cost of accuracy and delay, and alternative designs may utilize multiple predictions stored within a single pattern history table index. In general, multiple predictors can not achieve the same accuracy as single predictor implementations.

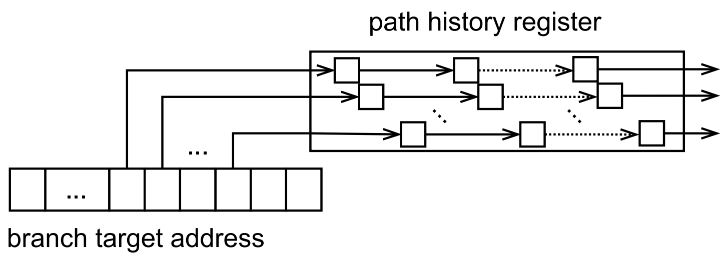
2.2.4 Path Prediction

Sections 2.2.2 and 2.2.3 illustrated prediction mechanisms that utilize a branch history. These mechanisms use pattern history, the history of prior branch outcomes. Path based history, or the sequence of previous basic-block addresses, can be used in place of pattern history [37] [24] [22]. Path based history encodes more information than outcomes alone can provide. It is possible for two distinct paths to produce perceived identical pattern history. While the pattern history is not truly identical, the limitation on the size of the history register prevents the predictor from being able to distinguish between them resulting in undesirable aliasing. Using path based history instead allows for differentiation between distinct paths regardless of prior branch outcomes.

Figure 2.7 shows the difference between pattern and path based history register contents [37]. The path based history register stores a subset of bits taken from the



(a) Pattern History



(b) Path History

Figure 2.7: Branch History Implementations

branch target address. Each insertion to the history shifts the current contents forward and appends the bits from the current target address. When read, the contents of the path history register can be logically divided into adjacent components where each component is a subset of bits from target addresses along the path up to the current point. Contents of the path history register can be used for indexing and hashing in the same fashion that pattern history would be used.

2.3 Dynamic Trace Caching

Static instruction ordering prevents continuous delivery of instructions by the processor's front end. The Trace Cache (TC) was designed to address this problem by adding a supplementary instruction cache that stores instructions in the dynamic order rather than static order [58]. Traces consist of instructions spanning multiple basic blocks. By fetching traces from the TC rather than instructions from the instruction cache, the front end is able to deliver instructions from noncontiguous regions in the same cycle.

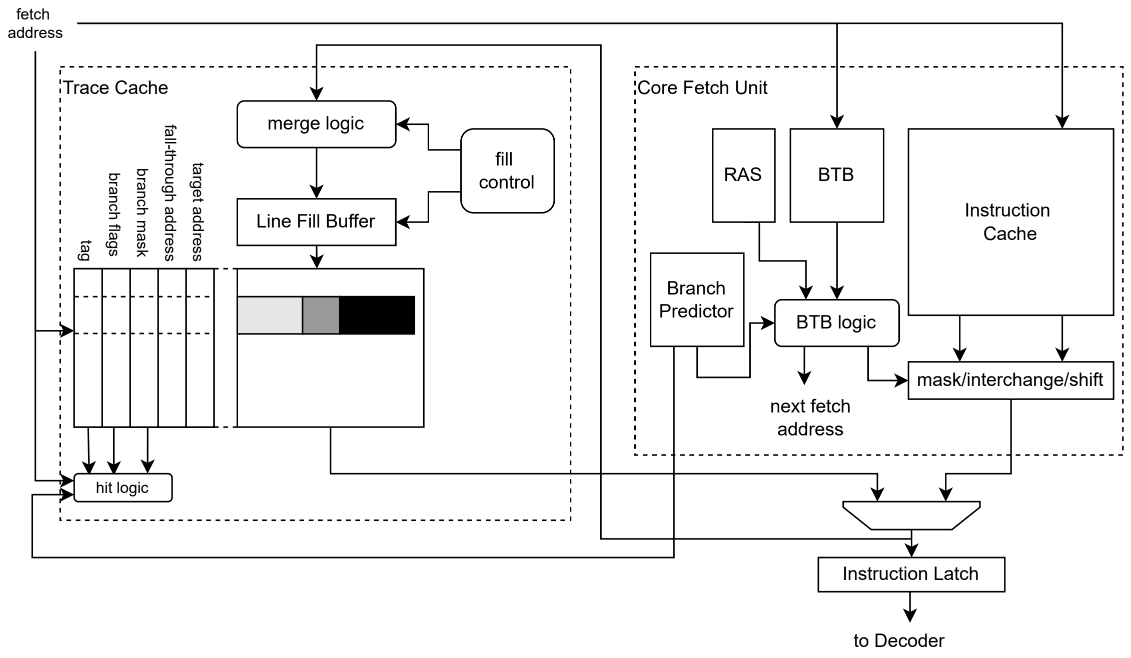


Figure 2.8: Trace Cache Front End

Figure 2.8 shows the organization of the TC and its inclusion with the processor front end [58]. The TC stores instructions that make up the trace as well as some additional information needed to utilize the trace. Branch outcomes corresponding to the trace are kept as well as the target and fall through addresses succeeding the trace. The fetch address is provided to the conventional fetch unit in tandem with the TC. In a given cycle, instructions may be exclusively issued from either the TC or the conventional fetch unit using the i-cache. A hit in the TC will cause selection logic to direct the output of the trace cache to the instruction latch. Otherwise instructions will be latched from the conventional fetch unit. As the dynamic instruction sequence becomes known, it is directed to the fill logic which is responsible for constructing new traces within the TC.

The trace cache can maintain high fetch bandwidth when a high TC hit rate is achieved. The trade off is increased hardware complexity dedicated to trace cache fill, hit and selection logic as well as increased storage cost since the TC storage is separate from the instruction cache. In practice TC hit rates are far lower than i-cache hit rates. The following sections briefly introduce design criteria that can significantly impact trace cache performance.

2.3.1 Trace Issue

Trace cache hit detection consists of two components.

1. The fetch address must match the beginning address of the trace.
2. All branch outcomes corresponding to the trace must match the predicted branch outcomes.

Matching the fetch address is handled by performing a tag comparison similar to how the instruction cache operates. Matching branch outcomes to predictions deviates from the conventional fetch unit. Use of a trace cache requires a multiple branch predictor, as discussed in Section 2.2.3, since all branch outcomes must be simultaneously known. TC hit rates are thus related to branch prediction accuracy.

Trace cache implementations may use either atomic or partial trace issue policies. Atomic policies issue the trace in its entirety or not at all. Therefore, all branch outcomes must match the predictor output to produce a trace hit. Partial trace issue allows a prefix of the trace up to the entire trace to be issued. Partial issue policies allow issue of instructions across correct branch outcomes until the first mismatched branch outcome. Partial issue policies seek to increase hit rate.

2.3.2 Trace Construction

Trace construction consists of buffering instructions issued by the fetch unit along with any branch outcomes and targets until the trace length or max number of branches permitted in a single trace is reached. Once the trace has been constructed, it is written into the trace cache. Instruction sourcing for trace construction can take place at two different points of the pipeline.

1. Speculative trace construction sources instructions as they are issued from the processor front end, where instructions themselves may be speculative.

2. Non-speculative trace construction sources instructions from the retirement stage of the pipeline once instructions are known to be non-speculative.

Speculative trace construction reduces the latency associated with populating the trace cache, but may trigger additional and potentially harmful evictions as a greater number of traces are written across the execution of the program.

2.3.3 Retirement and Recovery

Trace retirement occurs once all branches within the trace are resolved and the trace instructions reach the head of the ROB. Retirement and recovery can be handled by two different policies.

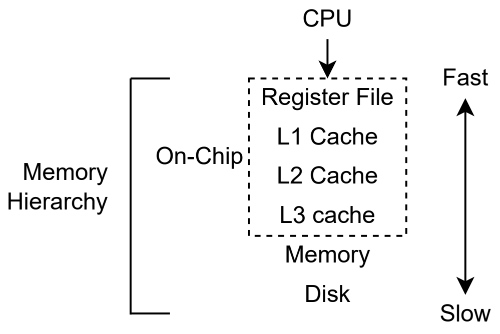
1. Atomic trace retirement requires all contents of the trace to be along the correct path. Either all instructions from the trace are retired or all are discarded. Recovery takes place at the starting instruction address of the trace.

2. Partial trace retirement allows a prefix up to the entire trace to be retired. In this policy instructions across correctly predicted branches are retired and those beyond any miss-speculated branches are discarded. In order to recover, branch targets must be kept for all intermediate branches within the trace. Recovery takes place at the correct target of the first miss-speculated branch. Partial

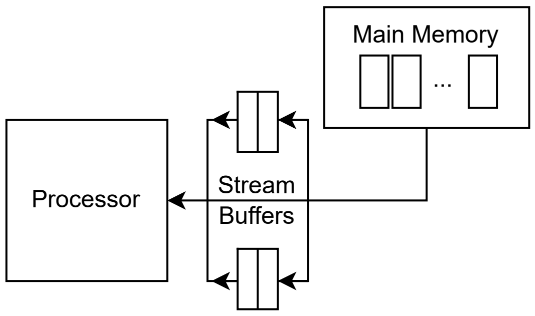
trace retirement prevents redundant execution of correct path instructions.

2.4 Static Trace Caching

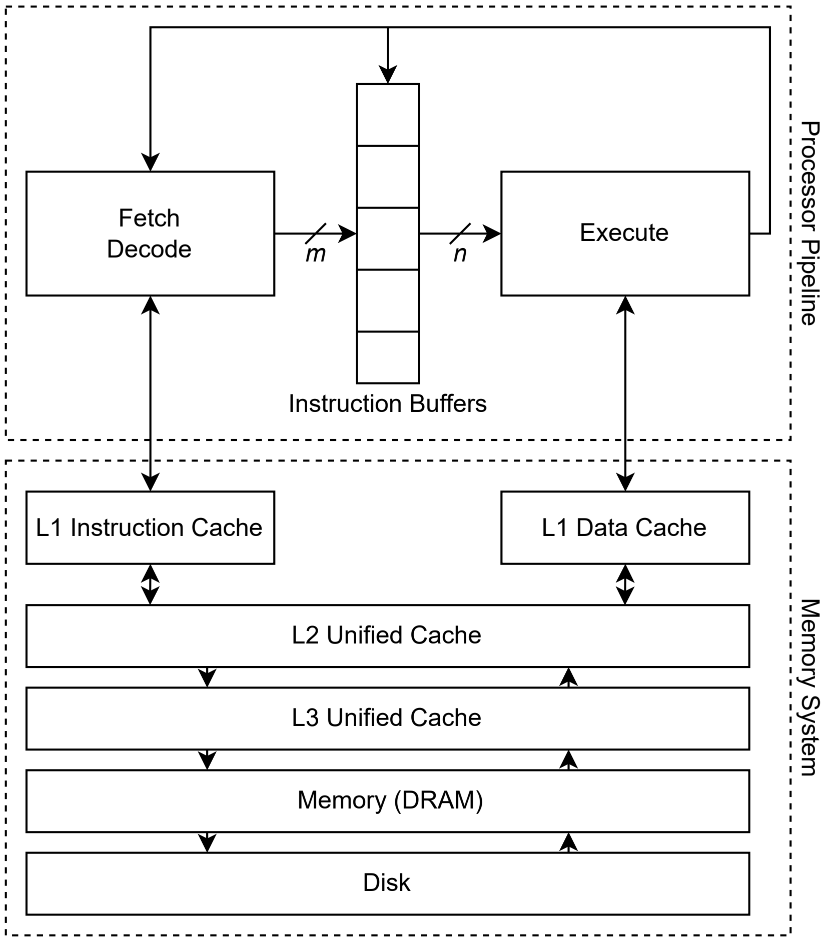
The trace cache is a hardware approach to delivering instructions in dynamic execution order without modifying the static program text. Modifying static program text provides an alternative way to construct and execute traces. The Software Trace Cache (STC) is a profile based code reordering technique that modifies the program text to better reflect the dynamic execution order of instructions [52]. Sequentially executed basic blocks are mapped to physically contiguous locations in memory to make the program more amenable to the natural sequential fetching of the fetch unit. STC can however negatively interact with branch prediction mechanisms. As dynamically executed blocks are placed in contiguous locations, branch outcomes become biased towards not-taken. Biasing branch outcomes has a negative effect on branch history contents exploited by many predictors. Branch history when using STC is primarily composed of *0* bits and begins to saturate thus preventing optimal use and distribution of entries within pattern history tables.



(a) Memory Hierarchy



(b) Stream Buffer Pre-Fetch



(c) Memory System Integration

Figure 2.9: Memory System

2.5 Memory System

Memory Systems for modern processor are designed to balance cost and performance using a hierarchical design as shown in Figure 2.9a [18]. Memory components closer to the processor are typically small and fast. As distance from the processor increases so does size and access latency. The fastest component of the memory hierarchy is the register file, followed by one or several levels of on-chip cache. Each level of the hierarchy is designed to exploit both spatial and temporal locality to provide high hit rates. Temporal locality is exploited by keeping recently used data at higher levels of the hierarchy. Spatial locality is exploited by moving data in larger blocks between levels of the hierarchy such that subsequent accesses to contiguous data will hit in a higher level of the memory.

Basic means of exploiting locality do not cover irregular or sparse workloads effectively. In such cases, inclusion of a pre-fetch mechanism can increase performance by adding a means to learn the access patterns during execution for later exploitation [36]. Figure 2.9b illustrates the basic organization of the Stream Buffer Pre-Fetch mechanism used to exploit sequential fetch [26] [42]. On a cache miss stream buffers begin pre-fetching and storing contiguous cache lines beginning at the target. Lines following the requested target are placed in the buffers rather than the cache itself. As lines are fetched by the processor, they are moved into the cache and stream buffer indexing is adjusted. This mechanism can help hide the latency of cache misses for sequential instruction delivery. Alternative pre-fetch designs buffer or cache noncontiguous lines spanning multiple basic blocks using history of previous control flow and path information.

Figure 2.9c shows a simplified model of memory system integration with the processor. Both the front end and back end of the processor need simultaneous access to the cache. The front end needs to fetch instructions in order to supply the back end with operations to execute. The back end needs to access the cache for loading and storing values used as input operands for execution. The level one (L1) cache is typically split into separate instruction cache and data cache components to exploit the exclusivity and independence of instruction fetch and data access. Memory levels beyond the L1, or nearest, cache are unified.

2.6 Program Representations

Program representations define how a program is represented and determine what information is associated with the program. Different representations may be used when compiling and optimizing code, and for directly executing code.

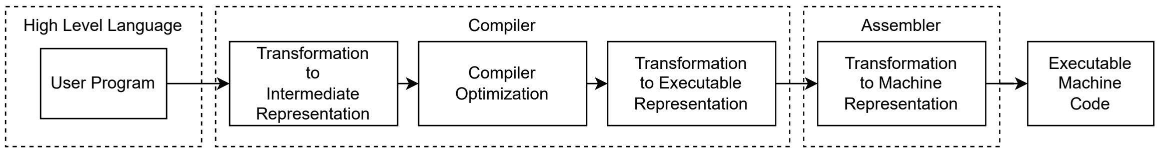

High Level Language
User Program
Compiler
Transformation to Intermediate Representation
Compiler Optimization
Transformation to Executable Representation
Assembler
Transformation to Machine Representation
Executable Machine Code

Figure 2.10: Code Transformation Process

Figure 2.10 shows a simplified code generation process which uses separate representations for optimization and execution. The following subsections describe relevant intermediate representations used for compiler optimization.

2.6.1 Control Flow Graph

Instructions from a program can be grouped into basic blocks; regions of code dependent on the same sequence of control. A control flow graph (CFG) is a representation composed of a directed graph with basic blocks as nodes within the graph. CFGs utilize two special nodes to denote the entry and exit points of the program. Edges between nodes in the CFG represent control transfer within the original program. Nodes for basic blocks that end in a branch have two outgoing edges, one to the fall through target and another to the branch target. Nodes for basic blocks that end in a jump have a single outgoing edge to the jump target. Nodes for blocks that fall through into the next block without a control transfer instruction have a single outgoing edge to the fall through block. Blocks with multiple successors represent branch nodes and blocks with multiple predecessors represent join nodes. Data dependencies are represented separately.

Chapter 3

Characterizing the Fetch Problem

3.1 Front End Architecture

Figure 3.1 shows a simplified, but characteristic model of fetch unit design for superscalar processors based on the *interleaved sequential* fetch unit presented in [11]. The fetch unit incorporates the instruction cache, branch target buffer, return address stack, branch predictor and interchange logic. The fetch unit shown uses a fetch width capable of delivering up to eight instructions. The fetched instructions are buffered at output to be sent to the decoder. The instruction cache line size is configured for eight instructions per line. The instruction cache is 2-way interleaved to support full fetch utilization across contiguous cache line boundaries. If a fetch target lands in

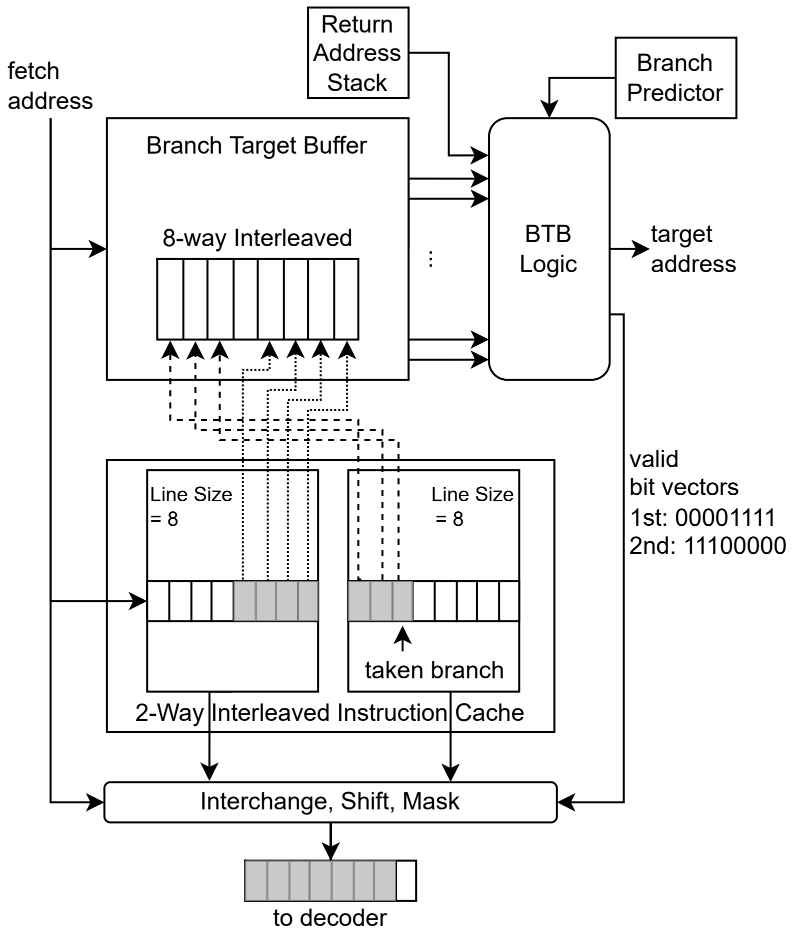


Figure 3.1: Interleaved Sequential Fetch Unit

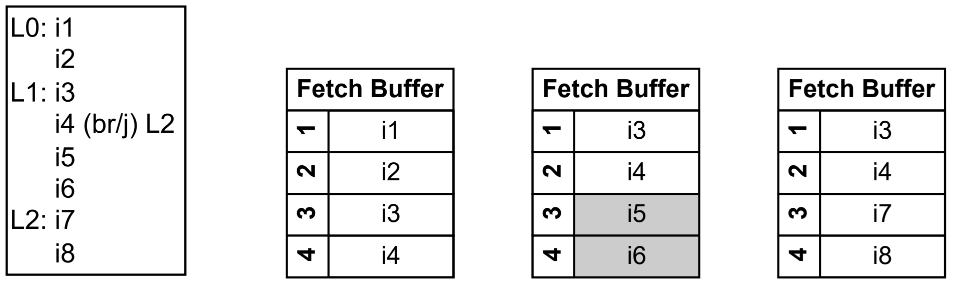
the middle of a cache line, as shown, instructions can be considered from the second cache line to provide the full fetch width. The interchange, shift and mask logic is responsible for rearranging instructions from cache lines as required and routing the ordered instructions to the output buffer. The branch target buffer is 8-way interleaved to support accesses for all instructions covered within the fetch width. The BTB provides branch detection and target information for control transfer instructions within the fetch block. Branch detection information, as well as return target provided by the RAS and branch outcomes provided by the branch predictor are used to drive the BTB logic. BTB logic outputs include the next fetch address and valid instruction information needed for correct interchange, shift and masking of fetched instructions.

Interleaved sequential fetch unit design is characteristic of the functionality present within contemporary control flow processors. Fetch width, instruction cache line size, branch predictor throughput and structure interleaving factor vary between implementation. It is possible to extend the branch predictor to provide predictions for multiple branches within a fetch block. Accesses to the fetch unit and its components can be pipelined or single cycle. Shift and interchange logic can be extended to mask off not taken regions for taken branch targets within the same cache line(s). The instruction cache can be supplemented with a dynamic trace cache or loop cache to supersede instruction delivery in select regions of the program.

3.2 The Fetch Problem

The inability of superscalar processors to fetch the optimal number of instructions is a direct consequence of the need to fetch multiple instructions from a single starting address, which implies implies sequential ordering of instructions. On the other hand, branch and jump instructions frequently disrupt this contiguous access pattern, rendering a number of instructions fetched from contiguous locations useless.

The starting fetch address as well as the position of a taken branch in the instruction sequence have a profound impact on the fetch efficiency.



(a) Program (b) Instruction issue from L0 (c) Issue from L1 (d) Ideal issue from L1

Figure 3.2: Instruction Fetch Example

Figure 3.2a shows a small program where *i4* is a taken control transfer instruction. Also shown is the fetch buffer for a processor front end capable of fetching and issuing 4 instructions per cycle. When the fetch address is *L0*, the fetch buffer is filled with four instructions *i1*, *i2*, *i3* and *i4*. Since there are no instructions following the taken control transfer, all four of the instructions in the fetch buffer are valid for issue as shown in Figure 3.2b. However, if the fetch target was *L1* as shown in Figure 3.2c, the instructions after the control transfer are not valid for issuing as instructions *i5* and *i6* are along the not taken path. Figure 3.2d shows the fetch buffer contents for the ideal case of fetching four instructions.

Full fetch buffer utilization cannot be achieved in a single fetch cycle given the program shown in Figure 3.2a. To achieve full fetch buffer utilization in the presence of taken control transfer instructions, the program must be restructured to place instructions along the taken path in contiguous locations following *i4*, or represented in a way that removes the effect of control transfer on instruction sequencing and validity.

Several dynamic and static techniques have been developed to address the fetch problem across taken control transfer. These techniques are described in Chapter 2.

3.3 Fetch Problem Components

The fetch problem can be decomposed into several key components.

3.3.1 Instruction Sequencing

Modern superscalar processors primarily use in-order fetch mechanisms. This works well when the dynamic instruction sequence matches the static instruction sequence, however, a problem is introduced when the dynamic sequence deviates from the static order. When the sequencing does not match, the processor must transfer control to another instruction address to continue constructing the dynamic instruction sequence. This problem is two fold as the processor must also omit or discard any instructions between the point of control transfer and the destination. Transferring control and omitting instructions not present in the dynamic sequence add latency to filling the back end with useful instructions. This latency is quite significant when control transfer is conditional and the condition is not yet resolved.

3.3.2 Control Speculation

To reduce conditional control transfer latency, superscalar processors implement branch prediction as a means to predict the outcome of a conditional control transfer before the outcome is computed. Although control speculation shortens the latency, it introduces another problem that effects fetch performance, the need to recover from a miss-speculation. Recovery includes redirecting the front end to the correct target as well as differentiating between correct and incorrect path instructions in the back end of the processor. The incorrect instructions must be removed without effecting the in order state of the program. During this time the front end is typically stalled and does not begin issuing instructions from the corrected target until recovery is complete. Although there are varied recovery mechanisms, all suffer some penalty when removing incorrect instructions as well as causing a bubble in the correct path instruction delivery. When branch prediction is employed, high accuracy is critical to maintain performance. Fetching more instructions is only useful when they are along the correct path.

Maintaining high control speculation accuracy is dependent not only on making correct predictions, but also recognizing control transfer instructions in the instruction stream. Processors that utilize branch prediction typically implement separate structures for identifying control transfer instructions and for making the predictions. Thus, there are multiple physical structures involved and performance is reliant upon correct outcomes from each.

3.3.3 Cache Residency and Latency

Aside from the performance factors introduced by sequencing and control transfer, latency associated with retrieving an instruction from memory impacts performance. Cache residency and the latency associated with moving instructions from each level of the memory hierarchy interact with the aforementioned factors as well. Fetching an instruction sooner may mean executing and resolving branch outcomes sooner as well.

3.3.4 Fetch Width and Fetch Buffer Utilization

High performance superscalar processors feature wide issue widths. To reap the benefits of increased issue width and execution resources, the front end must be able to provide comparable instruction fetch width. As the issue width increases, it becomes more challenging for the front end to maintain an average fetch width near the issue width. For an 8 issue superscalar, an average fetch width of 4 instructions is typical in integer code. The fetch limitation is due to the relatively short average depth of basic blocks in comparison to the issue width. Often the fetch width is limited by control transfer instructions. When a taken branch or jump is encountered, the remaining instructions in the fetch buffer are along the wrong path and not valid for issue. In cases where there are still correct path instructions present following the control transfer, these instructions are not typically fetched in the same cycle. When there are more branches present than the processor has means to predict, the front end must stall. Thus, part of the front end resources are regularly unused resulting in lower effective fetch width, or fetch buffer utilization.

3.3.5 Negative Interaction with Memory Speculation

Speculative instructions are not permitted to change the in-order state of the processor. They can however impact the data cache contents. As loads are issued speculatively, they may trigger cache misses and line fills to the data cache. If the load is discarded as part of miss-speculation recovery, no effort is spent to restore the data cache state. Thus, it is possible that a useful line was evicted for a line that is not guaranteed to be needed. Performance impacts of data cache pollution are lesser than the aforementioned components of the fetch problem, but it must be stated that it is possible for front end speculation to have negative interactions with other components of the processor depending on organization.

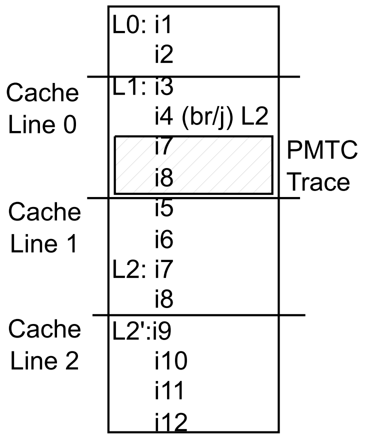
Chapter 4

Poor Man's Trace Cache

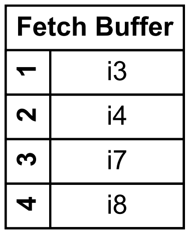
4.1 PMTC and the Fetch Problem

Considering the fetch example for the program presented in Figure 3.2, full fetch buffer utilization cannot be achieved in a single fetch cycle by conventional means. To achieve full fetch buffer utilization in the presence of taken control transfer instructions, the program must be restructured to place instructions along the taken path in contiguous locations following *i4*. This is statically feasible when the control-transfer instruction is either a jump, or a branch that is heavily biased towards *taken*.

In order to see how a trace containing replicated instructions from the taken path is placed inline with the control transfer instruction, consider Figure 4.1. The cache line boundaries are shown with horizontal divisions in the program. When the fetch target is *L1*, instructions *i7* and *i8* are fetched from contiguous locations following *i4*. The target is dynamically adjusted such that the next fetch cycle begins fetching instructions from *L2'*. We now elaborate on the delay slot characteristics, delay slot and trace insertion algorithm and the relevant changes to instruction semantics.



(a) Modified Program with PMTC Trace



(b) Instruction Fetch from L1

Figure 4.1: PMTC Fetch Example

4.2 Delay Slot Characteristics

PMTC delay slots have the following characteristics:

1. Delay slots may optionally be applied to or omitted from any DCTI.

2. Instructions used to populate delay slots are copied from the blocks reachable from the target.

3. Delay slot size is determined by the position of the DCTI within an L1 instruction cache line and the size of the line since the slot ends at the cache line boundary.

4. Any type of instruction, including control transfer instructions, can be placed within a delay slot.

5. All instructions from a delay slot are valid for fetching when the DCTI is taken.

6. The DCTI target is dynamically adjusted retroactively to prevent fetching any instructions previously fetched from the delay slot.

The behavior of select control transfer instructions is modified such that when these instructions are encountered, the processor takes note of the target, but delays the control transfer until it has fetched as many instructions remaining between the DCTI and the next cache line boundary as possible. The micro-architecture mechanism which adjusts the fetch target and invalidates any instructions previously fetched from the delay slot as necessary is described in Chapter 5. Note that, the behavior of other control transfer instructions is unmodified.

4.3 Traces in Delay Slots

PMTC's trace construction relies on instruction replication. The replicated instruction traces are stored in the I-cache distributed throughout the original program. Placing traces in the same cache line as the DCTI guarantees that traces held within delay slots are fetched in their entirety with the control transfer instruction. During trace insertion, the code is modified to push the instructions following the DCTI to the next cache line boundary to open-up the delay slot. A trace of instructions starting with the DCTI target can then be copied into the created slot. PMTC traces may include copied control transfer instructions. If a DCTI is copied, instructions from the copied DCTI's target may also be copied into the trace.

4.4 Identifying Candidates

Every direct control transfer instruction is a valid candidate for trace insertion. However, not every DCTI should receive a delay slot. DCTIs should only receive a delay slot if there is an assumed static taken bias and the delay slot would yield one of the following benefits:

1. When a loop body is smaller than the cache line size and all instructions between

the target and candidate DCTI fit in the delay slot, the entire loop body is replicated in the trace. Replicating the loop body produces a static unroll of the loop, but more importantly it doubles the effective fetch width while iterating over the loop.

2. Every instruction from the target to the cache line boundary beyond the target fits inside the delay slot. Better cache alignment across fetch cycles is achieved as a result.

3. The target contains any control transfer instruction that would be copied into the delay slot. Copying a control transfer instruction results in fetching the control transfer instruction a cycle early. Additionally, the intermediate fetch target may be entirely eliminated. If the copied instruction is either a taken branch or a jump then the next fetch address will be the target of the copied control transfer instruction rather than the instruction's actual location.

Ideally every jump and loop back edge branch will be treated as a candidate for trace insertion. Since our insertion algorithm is implemented in the assembler without compiler support, complete loop information is not available. Therefore, we consider all backwards branches as trace insertion candidates instead. Traces are also inserted following all unconditional jumps.

Figure 4.2 shows a block of code taken from *400.perlbench* after undergoing slot

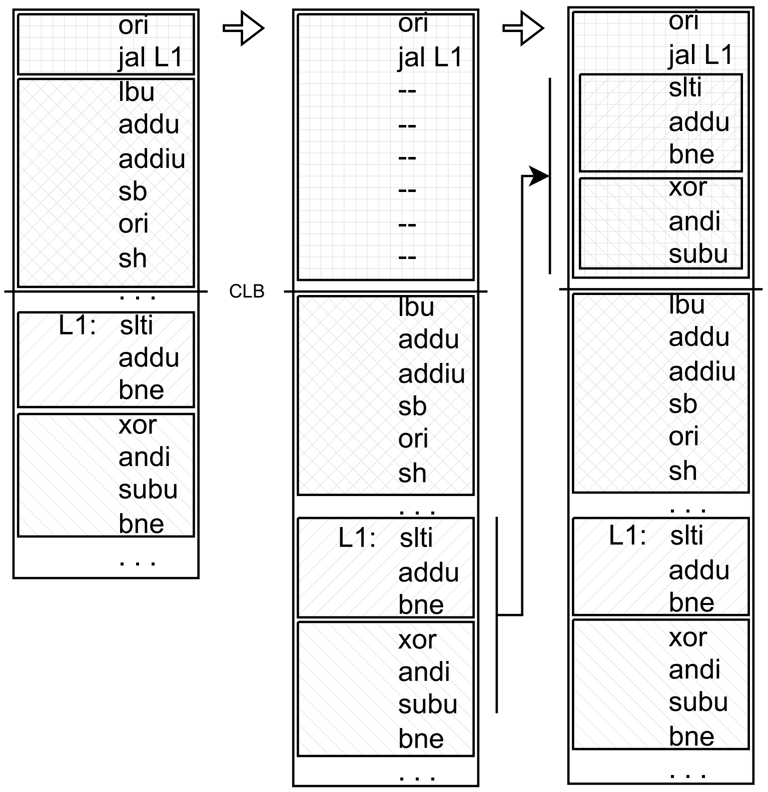


Figure 4.2: Populating the Delay Slot

insertion and population following a jump instruction. The code on the left shows the original sequence of instructions. The center column shows the adjusted code after the delay slot is inserted. Immediately after insertion the delay slot is empty. The code on the right shows the modified sequence of instructions after instructions have been copied into the delay slot from the target.

4.5 Addressing

Micro-architecture support for PMTC requires distinguishing between the instruction addresses of copied instructions and the addresses of instructions being copied so that proper address adjustments can be made, and precise exceptions can be maintained. We define an instruction's *actual address* to be its address within the program image. An instruction's *logical address* is the actual address of the original instruction from which the instruction in the delay slot is a copy. For instructions not in a delay slot, the actual and logical addresses are the same.

4.6 Semantic Changes

Since traces can be selectively applied to branches, we extend the MIPS instruction set with dedicated delayed-branch instructions. There are four delayed branch instructions added: *bned*, *beqd*, *blezd* and *bgtzd*. These branches function identical to their original counterparts except that the instructions following the branch to the next cache line boundary make up the delay slot. It is assumed that the front end can invalidate instructions already fetched from the delay slot if they reside at the target in the next cycle. Although instructions are replicated, they are only delivered to the processor back end once. Fall through blocks for delayed branches now begin at the next cache line boundary. Therefore, when a delayed branch is predicted to be not taken, the next fetch target is the following cache line boundary. Not taken behavior for a delayed branch requires the processor to jump over or mask off the delay slot and fetch subsequent instructions starting at the next cache line boundary.

For any branches in a delay slot branch, targets are not modified. Instead, it is assumed that the front end is aware of every instruction's logical address. Targets remain relative to their logical PC. Branch targets, and BTB entries, are computed as the logical address plus the branch offset.

Since all direct jumps are accompanied by a trace, jump and jump and link semantics are modified to fetch instructions following the jump to the next cache line boundary. For jump and link, the return address is always the next cache line boundary beyond the logical address of the jump.

4.7 Control Transfer Chain Reduction

The ability to copy instructions from nested targets into contiguous addresses provides significant benefits. When instructions from multiple noncontiguous targets are copied into the delay slot, the front end is able to avoid fetching intermediate cache lines between the first control transfer instruction and the final target.

Control transfer chain reduction takes place when basic blocks fit entirely within a delay slot. Dynamic execution of the modified code results in fewer overall cache lines being fetched versus the original program. Figure 4.3 shows a block of code taken from *400.perlbench*.

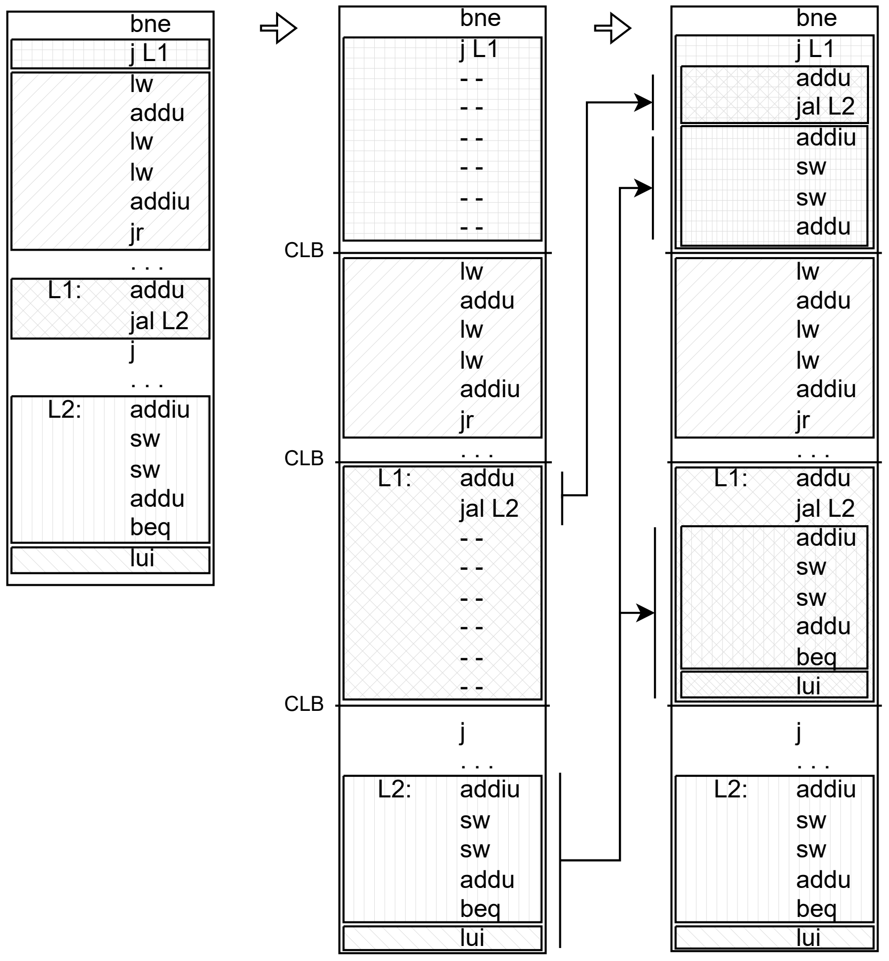


Figure 4.3: Trace Construction Across Multiple Basic Blocks

The placement of three noncontiguous basic blocks along a taken path within a single cache line yields a delay slot which includes instructions across multiple basic blocks forming a continuous trace. In the figure, the left column shows the original code, the center column shows the spacing added for the delay slots and the right column shows the modified code with populated delay slots. Horizontal lines mark the cache line boundaries. Basic blocks of interest have been annotated. The modified program removes the need to fetch the intermediate block and instead directly fetches the remaining instructions beyond *L2* from the third basic block of the control flow chain in the next cycle.

4.8 PMTC Assembler

No compiler support is needed to insert delay slots and construct PMTC traces. All code modifications are performed by the assembler. The assembler is ignorant of any loop information. A simple backwards branch heuristic was implemented to approximate branches that may be controlling loop iteration. All unconditional direct jumps trigger delay slot insertion. Backward branches are considered as candidates. Algorithm 3 shows the function used to determine if a candidate should precede a delay slot. Branches determined to be prime candidates trigger delay slot insertion and are marked as a delayed branch by updating their opcode.

The assembler used for PMTC is modified to insert empty delay slots during the second pass once backwards branch targets are known. Algorithm 1 shows the modifications to the second assembler pass that inserts the delay slot. The second assembler pass walks over the instruction stack and emits no-ops to push the subsequent instructions to the following cache line boundary for select DCTIs.

Algorithm 1: PMTC Assembler Pass 2 Modifications: Delay Slot Insertion

```
Result: Insert space for a delay slot after select DCTIs
DO: pad text segment start to cache line boundary with no-ops;
i =text_begin;
while i != text_end do
    if instruction_stack[i] is a direct jump then
        while i mod cache line size ¿ 0 do
            insertNo-op(i, instruction_stack);
            i++;
    else if (instruction_stack[i] is a backwards branch) &&
      (primeCandidate(i) == true) then
        setDelayedOpcode(instruction_stack[i]);
        while i mod cache line size ¿ 0 do
            insertNo-op(i, instruction_stack);
            i++;
    else
        i++;
```

Algorithm 2: PMTC Assembler Pass 3 Modifications: Delay Slot Population

```
Result: Copy instructions from targets into the delay slots
i =text_begin;
while i != text_end do
    if instruction_stack[i] is a delayed branch or direct jump then
        target = instruction_stack[i].target;
        i++;
        while (i mod CLS) ¿ 0 do
            instruction_stack[i++] = instruction_stack[target++];
```

Delay slots are populated on the third pass of the assembler. Algorithm 2 shows the necessary modifications. Whenever an unconditional jump or a branch that has been converted to a delayed branch is encountered, instructions are copied from the target into the delay slot up to the following cache line boundary.

```
Algorithm 3: primeCandidate(index i)
Result: Determine if this br should be converted to a delayed br
1  CLS = cache sub block size;
2  j = instruction_stack[i].target;
3  if (i - j) ¡ (CLS - ((i + 1) mod CLS )) then
4      return true; //Loop unroll in DS. Double fetch width
5  else if (CLS - (j mod CLS)) ¡= (CLS - ((i + 1) mod CLS )) then
6      return true; //Better cache alignment.
7  else
8      while (j mod CLS) != 0 do
9          if instruction_stack[j++] is a control transfer instruction then
10             return true; //DS will contain a CTI
11 return false;
```

4.9 Effect on Branch Prediction

Replication of control transfer instructions positively interacts with conventional prediction mechanisms. PMTC code may have multiple instances of the same branch at multiple different addresses throughout the program. We refer to branches that have been replicated as *path separable*. Path separability exists when the same branch instruction is encountered at a different address along different paths. PMTC exploits path separability to improve branch prediction accuracy.

Figure 4.4 shows how PMTC replicates branches along distinct paths. Branches br_a and br_b are delayed and followed by a trace. Both traces contain a replicated instance

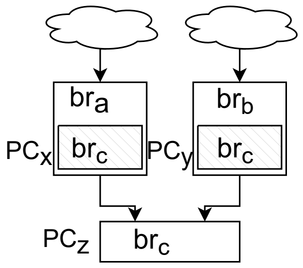


Figure 4.4: Path Separation of Control Transfer Instructions

of br$_c$. Branch br$_c$ has a unique address along each path, PC$_x$ when following br$_a$ and PC$_y$ when following br$_b$. Conventional branch prediction utilizes the instruction address and a history of branch outcomes. It is possible that along both paths the history is identical when br$_c$ is reached. In this case, there is nothing to distinguish between which path was executed up to br$_c$. If there exists correlation between the computation along the separate paths and the outcome of br$_c$, the branch predictor will be unable to learn the behavior of br$_c$ with high confidence. For conditional branches, path separability explicitly encodes additional path information via the instruction address used by the branch predictor. When the branch history is identical, br$_c$ will index separate locations within the branch predictor along the two paths shown when utilizing the PMTC traces.

Path separability affects the BTB in 2 ways.

1. Indirect jumps may store different targets along different paths. If br$_c$ in Figure 4.4 is an indirect jump with different targets assigned along the paths through br$_a$ and br$_b$, the BTB will be better able to learn the path relative targets as

they reside in different locations within the BTB.

2. For a given execution path, branches encountered along the way take-up positions at the same (interleaved) index. As a result, if a branch on that path hits in the interleaved BTB, a copied branch will also hit in the BTB with high probability.

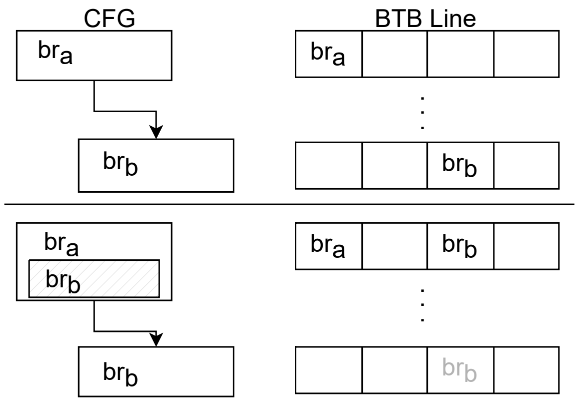


Figure 4.5: BTB Entries

Figure 4.5 shows the phenomenon. The top portion shows the original locations where the entries for br$_a$ and br$_b$ reside. The bottom shows the effect when a PMTC trace containing br$_b$ is utilized. Branch br$_b$ now resides in the same interleaved index as br$_a$.

The original entry may also be present in the BTB and such redundancy will vary over the dynamic execution depending on if the same branches are reached along multiple paths. Sensitivity to BTB size and the performance impact of additional pressure from replicated instructions are examined in Chapter 6.

Chapter 5

Micro-Architecture

5.1 Baseline Architecture

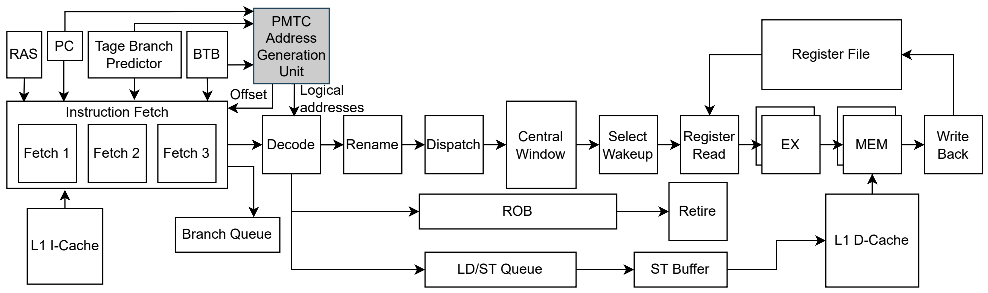


Figure 5.1: Superscalar Processor Pipeline

The baseline architecture used as the basis for PMTC and for comparison is a realistic 11-stage 8 issue superscalar pipeline. A modified version of MIPS without fixed size delay slots similar to PISA ISA [4] was used as the instruction set. The pipeline consists of three stages of fetch, decode, rename, dispatch, select/wake up, register read, execute, memory access and writeback. The architecture is a central window and reorder buffer based implementation. Memory instructions use a load/store queue and a store buffer. Memory prediction is done using store-sets [9] and memory order violations are detected using an out-of-order store value-matching algorithm [41]. Figure 5.1 shows the general pipeline organization.

5.1.1 Fetch Unit

The fetch unit design was modeled after the Core Fetch Unit proposed in [58] and extended to a 3-stage pipeline. A dual bank L1 instruction cache is used to provide up to 8 instructions spanning a single cache line boundary. A fetch cycle begins by bringing two i-cache lines containing the PC address to the fetch unit. These cache lines are interchanged as necessary. Eight instructions beginning at the PC address are shifted and inserted into a fetch buffer. The PC is simultaneously used to access an 8-way interleaved BTB. Once any branches have been identified in the 8 instruction fetch block, the TAGE branch predictor [64] is accessed for up to 2 conditional branches. When a return is detected, the return address stack (RAS) is popped to obtain the next fetch address. Once the BTB information and branch predictions are available, fetch buffer entries are masked off as necessary. Any taken branch will result in masking off all subsequent instructions in the fetch buffer. The valid instructions in the fetch buffer are then sent to decode.

5.2 PMTC Fetch unit

PMTC uses the same superscalar processor pipeline as the baseline with a few modifications. As discussed before, static code modifications do not modify branch offsets when in a trace. Using unmodified branch offsets has a few implications:

1. Logical addresses for each instruction in a trace must be computed.
2. Logical addresses must be used to compute relative branch targets. The BTB must be updated with the targets relative to a branch's logical address.
3. In the fetch cycle following a delayed branch the fetch target must be dynamically adjusted to mask off any instructions that were already fetched from a trace.

Computing logical addresses in the front end significantly reduces the complexity of executing instructions within a trace and supporting precise exceptions. To the remainder of the pipeline, instructions from a trace appear as if they were fetched from their original location. The rest of the pipeline and the mechanisms implemented that rely on an instruction's address do not need to be modified to support execution of instructions coming from a PMTC trace.

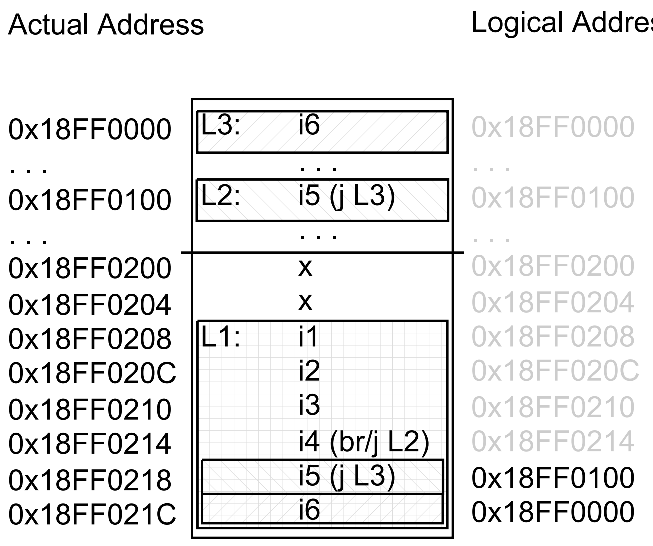


Figure 5.2: Addressing for Instructions Within a Trace

Figure 5.2 conceptually shows the logical addresses which need to be computed for instructions within a PMTC trace. *L1* is the initial fetch target. The horizontal line above *L1* marks the cache line boundary. Instruction *i4* is a delayed branch or jump and *i5* and *i6* make up the trace contained within *i4*'s delay slot. Instruction *i5* is copied from *L2* and *i6* is copied from *L3*. The next cache line boundary follows the end of the trace. Observe that the instructions outside of a trace do not have a logical address, or rather their logical address is identical to their actual address. The logical address for an instruction in the trace is the actual address of the instruction that was replicated. We now elaborate on the actual fetching mechanism.

5.2.1 Updating the BTB and Branch Queue

As branches exit the fetch unit they are placed into a branch queue. The branch queue stores all relevant information that will be needed in the future, including the instruction address. PMTC extends the branch queue to contain the logical address of every branch in the branch queue and an additional bit to mark delayed DCTIs. Branch targets are computed relative to the logical address in the execution pipeline stage. The BTB target is updated with the logical address relative target. Although the logical address is used for the target, the actual address of the branch is used to index the BTB and branch predictor for updates. Actual address indexing is used because the BTB and branch predictor are accessed in the front end, before logical addresses for instructions are computed. Updating the BTB with logical addresses also ensures that no control transfer target will land within a trace. Updating the BTB with logical addresses also decouples fetch target generation from logical address generation. The fetch unit can then use BTB targets to direct instruction fetch and adjust the fetch buffer contents as needed in the subsequent fetch cycle to properly handle instruction fetch from PMTC traces. Proper trace handling involves adjusting fetch targets to avoid duplicate fetching of instructions from PMTC traces.

5.2.2 Adjusting the Next Fetch Address

When there are no taken branches in a fetch block, the next fetch target generation is identical to the baseline. The continuation address in the next cache line will be used as the fetch target. When a taken branch or jump is detected, the next fetch address used is also the same as the baseline. The address is provided by the BTB. However since BTB targets do not account for instructions fetched from traces, if PMTC began fetching instructions from the target in the next cycle, any instruction already fetched from a trace in the prior fetch cycle would be fetched twice. Therefore, for each instruction fetched, the PMTC *Address Generation Unit* shown in Figure 5.1 computes an *offset* defined to be *the distance between the instruction and the prior branch target*. These offsets are used in the next fetch cycle to prevent duplicate fetching of instructions already fetched from a trace by applying them to the fetch target when filling the fetch buffer. Instructions already fetched from a trace are masked off.

The offset needed is determined by the offset computed for the last valid instruction fetched in the last fetch cycle plus 1. Applying the offset shifts the fetch target to the address of the last instruction fetched from the prior trace. An increment of 1 is applied to adjust the fetch target to the next instruction beyond the last instruction fetched, which should be the first instruction fetched in the current fetch cycle. The offset is only applied when a delayed DCTI has been encountered, which is known by the BTB bit corresponding to delayed control transfer. If the last instruction fetched was the delayed DCTI itself, then the offset is not applied in the next fetch cycle as no instructions have been fetched from the trace and the next instruction that must be fetched is located at the target with no offset. Since PMTC can permit instructions spanning multiple branch targets in a single cache line, offsets will be reset to 0 following a delayed branch or jump.

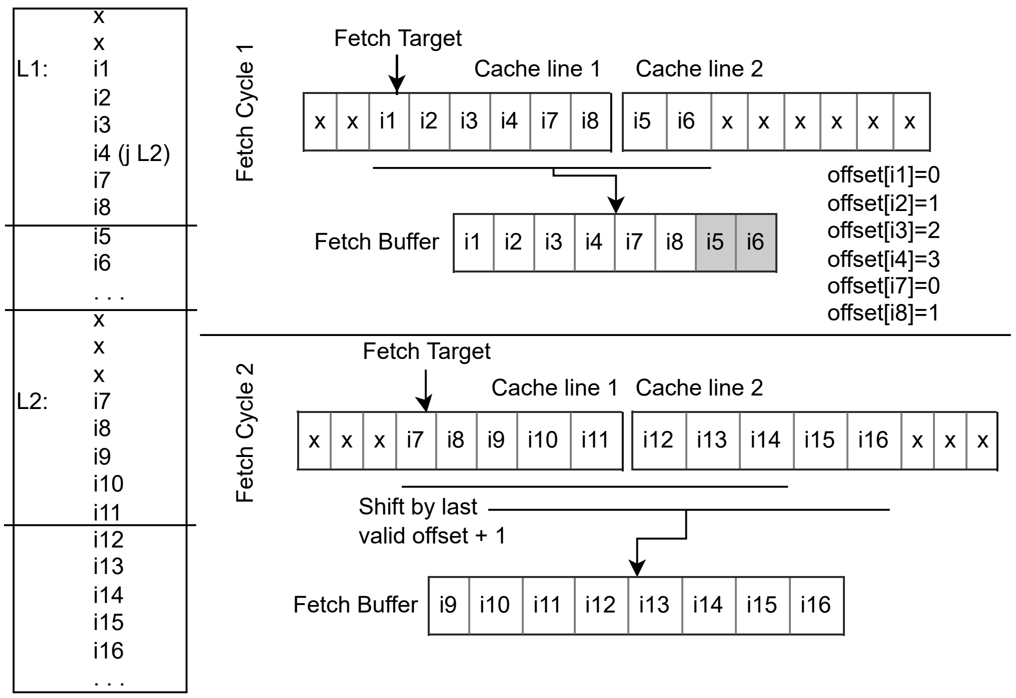


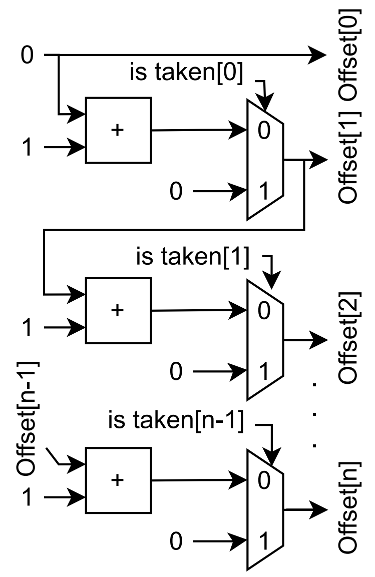
Figure 5.3: Fetch Address Adjustment

The offset can be at most 8 instructions, or one cache line away from the BTB target used as the fetch target. Fetching two cache lines every cycle ensures that there will still be 8 instructions that can be considered for fetching after adjustment. Figure 5.3 shows an example where an offset is applied in the second fetch cycle. *L1* is the fetch target in the first fetch cycle. Instruction *i4* is a taken delayed branch or jump and the instructions from *i4* to the cache line boundary are part of the trace. In Fetch cycle 1 instructions *i7* and *i8* are fetched from the trace. The offset computed for *i8* is 1. The address *L2*, the branch target of *i4*, is used as the fetch target for the second fetch cycle. In the second fetch cycle the fetch buffer contents are adjusted by 2 instructions, the offset of *i8* plus 1, to avoid fetching trace instructions a second time.

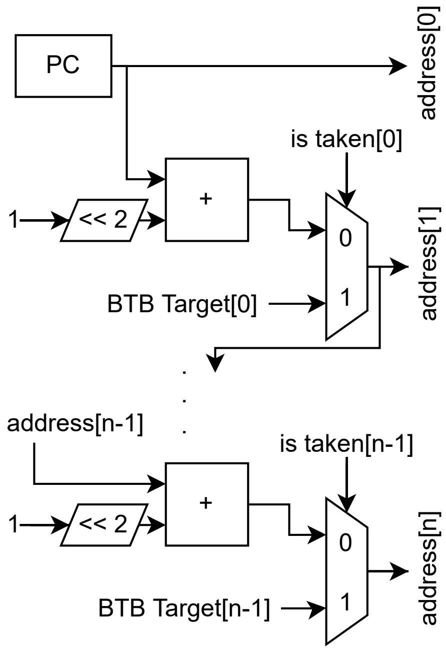
Revisiting the example shown in Figure 5.2 where *L1* is the fetch target, the offsets computed for *i1*, *i2*, *i3* and *i4* are 0, 1, 2 and 3 respectively. Once *i4* is recognized as a delayed DCTI the offset of the next instruction is reset thus the offset of *i5* is zero. Instruction *i5* is also detected to be a delayed DCTI, thus the offset of *i6* is reset to zero as well. When the cache lines corresponding to the next fetch target arrive in the next fetch cycle, the offset of the last valid instruction will be used to adjust the fetch buffer starting address such that *i6* is not fetched again.

5.2.3 Logical Address Generation

The Address Generation Unit implemented in the pipeline front-end is responsible for computing an instruction's logical address. Shown in Figure 5.4, this unit is conceptually split into two components, the first to generate an instruction's offset beyond the last target and the second to generate the logical address for each instruction.



(a) Computing Instruction Offsets



(b) Computing Logical Addresses

Figure 5.4: Address Generation Unit

Generating the offset is off the critical path. Offsets are used in the next fetch cycle to drive shift and masking logic. PC addresses are determined by BTB targets or the line continuation. Figure 5.4a shows the organization of the offset generation component. A series of 3 bit adders and 2 input multiplexers are used. Each adder computes the offset of an instruction relative to the prior target. When a taken branch or jump is detected given the BTB information and branch predictions, the offset beginning at the next instruction following the taken branch is reset to 0. The offset is reset because it is relative to the prior target and the current instruction is the first instruction from the current target. Each offset computed is passed to the next adder in series as input.

The organization of the second component responsible for generating the 32 bit logical instruction addresses is shown in Figure 5.4b. Each instruction's logical address is the instruction word size added to the previous instruction's logical address. The left shift converts the 3 bit instruction offset to a byte offset. The left shift is conceptual and used for display purposes only as the value used by the multiplexor will be the instruction word size. If the prior instruction is a taken branch or jump, the logical address is replaced by the branch target provided by the BTB. Logical address generation is off the critical path. The next fetch target is provided by either the BTB or the line continuation. Logical addresses are used when computing branch targets in EX and updating the BTB. Considering the example shown in Figure 5.2 where *L1* is the fetch target, computing logical addresses for instructions *i1*, *i2*, *i3* and *i4* results in logical addresses identical to the actual address for those instructions. Since *i5* and *i6* are delayed DCTIs, the mux control signal *is_taken* is digital 1. The mux for *i5* replaces the continuation of the prior address with the BTB target for *i4*. The mux for *i6* replaces the continuation of the prior address with the BTB target for *i5*. Thus, the logical addresses for *i5* and *i6* are the actual addresses of the instructions that were copied when the trace was constructed.

5.2.4 Masking the Delay Slot

PMTC offers the unique ability to fetch from the taken or not taken path when a trace is present within the first cache line fetched. When a delayed branch present in the first cache line is predicted not taken, the not taken instructions begin at the next cache line boundary. Since the fetch unit reads 2 cache lines every fetch cycle some of these instructions are available in the current cycle. The instructions from the trace can be simply masked off. A more sophisticated approach would shift the not taken instructions over to reclaim any fetch slots wasted by instructions from the trace. The simple masking approach was implemented to avoid adding unnecessary complexity to the fetch unit. Traces are only inserted after branches with a statically assumed highly taken bias so the effects of rescuing a few fetch slots in this case are insignificant.

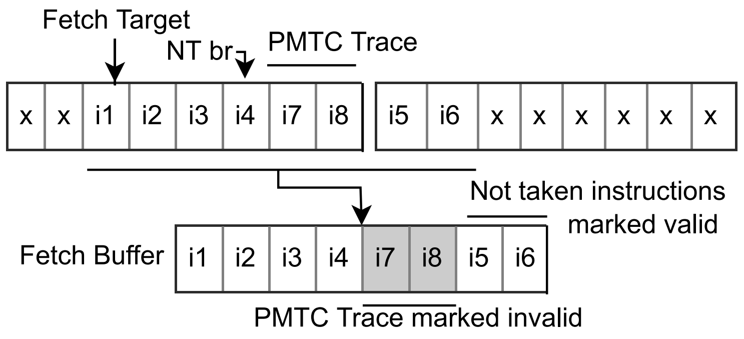


Figure 5.5: Trace Masking for Not Taken Branches

Figure 5.5 revisits the example in Figure 5.3, but assumes *i4* is predicted not taken.

When *i4* is not taken, every instruction following *i4* to the cache line boundary is marked invalid, and every instruction following the cache line boundary is marked valid.

5.3 Exception Handling

Detecting a branch misprediction for a delayed branch is similar to a regular branch except that if there is a BTB miss and the branch was predicted not taken, that branch has been mispredicted. The instructions from the delay slot were fetched instead of being masked off. When delay slot instructions are fetched erroneously, recovery is triggered at the fall through address at the next cache line boundary.

When recovering from a misspeculation the fetch unit assumes the fetch target is not within a trace. Therefore, care must be taken to ensure that recovery never restarts at an address within a trace. Load and store instructions entered into the LD/ST queue keep their logical address rather than their actual addresses. Storing logical addresses instead of actual addresses ensures that any restart triggered by a load misspeculation will not begin in a trace. Preventing restart at an address within a trace is also easily accomplished for predicted not taken branches as branch targets are never within a trace. The branch target is used as the recovery address. When recovering from a predicted taken branch, the logical address stored in the branch queue is read and adjusted to determine the address of cache line boundary beyond the mispredicted branch. Recovery begins at the address of the next cache line.

Chapter 6

Evaluation

Cycle accurate simulators were developed using the Architecture Description Language [40]. Spec 2006 integer and floating point benchmarks [19] with ref inputs were evaluated using Simpoint methodology [65]. Table 6.1 shows the datasets used and the baseline IPC for each. [1]

Processor configuration used is shown in Table 6.2. Simpoints were generated for the baseline and PMTC using the same starting locations in terms of retired instructions during dynamic execution of the benchmarks. Each simpoint was executed using 100M instruction intervals from a cold start.

[1]IPC improves 40%-100 % when a fixed 100 cycle memory subsystem is used.

Benchmark	Dataset	IPC
perlbench	checkspam	.584
bzip2	00-input.source	.669
gcc	00-166.in	.522
bwaves	bwaves	.564
mcf	00-inp.in	.260
milc	su3imp	.683
leslie3d	leslie3d	.816
gobmk	13x13	.650
sjeng	ref	.854
libquantum	ref	.590
h264ref	foreman_ref_encoder_baseline	1.404
lbm	lbm.in	.388
astar	rivers	.644
sphinx3	default	.571

Table 6.1
Datasets & Baseline IPC

6.1 Ideal Fetch Unit

PMTC and the baseline share the same pipeline except for the address generation unit and instruction semantic changes described in Chapter 5.

For comparison to prior art, notably the Trace Cache [58], we chose to implement an *ideal fetch unit*. The configuration *Baseline Ideal* uses the same pipeline, but allows the fetch unit to fetch instructions from noncontiguous blocks across taken control transfer instructions. In other words, the instruction fetch for the ideal configuration is not limited to the dual i-cache sub block fetch mechanism used in PMTC and the baseline. As long as the target of a control transfer instruction is resident in the

Branch Predictor	TAGE 2 predictions 8KB 7 tagged tables 1 bimodal	BTB	8 way interleved 2048 entry
Branch Queue	32 entry	Fetch	2 sub blocks
Issue Width	8 instructions	Retire Width	16 instructions
Register File	180 registers 18 read ports	Scheduler	Central Window 97 instructions
Reorder Buffer	224 instructions	Execution Units	8 integer 4 float 2 address
LD/ST Queue	128 entry	Store Buffer	56 entry
Memory Speculation	OOOVM Store Set ssit bits 12 lfst bits 12	Recovery	flush and restart br penalty 19cy ld penalty 15cy
L1 I Cache	32KB sets 128 assoc 4 way block 64B sub block 32B LRU hit latency 3 cy dual bank	L1 D Cache	32KB sets 128 assoc 4 way block 64B LRU hit latency 4 cy MSHRs 16x8 load ports 2 store ports 2
L2 Unified Cache	2MB sets 4096 assoc 8 way block 64B LRU hit latency 27 cy	DRAM	DRAMSim 2 [57]

Table 6.2
Processor Configuration

i-cache, instructions from that target are fetched in the same cycle limited only by the branch predictor throughput and the number of branch queue entries. Hence, it provides an upper bound of performance for techniques that improve only the fetch bandwidth, which includes the Trace Cache. Using an ideal fetch unit also allows us to eliminate a potential issue with the branch prediction. Trace Cache and trace processor implementations typically utilize a multiple branch predictor or path predictor, and these predictors are not compatible with PMTC. Using different branch prediction mechanisms for each configuration would prevent any meaningful comparison.

6.2 Static Text Composition

Table 6.3 shows the fraction of instructions converted to delayed control transfer in the original text segment for each benchmark. Roughly 10 percent of original instructions are converted to delayed CTI. The delayed CTI include every unconditional direct jump and select integer branches in the original text.

Figure 6.1 shows the instruction mix contained within traces added to the text segment. The fraction reported is in terms of the unmodified text size. The height of each bar shows the overall text segment size increase. PMTC increases text segment size by roughly 38 percent. Trace contents are primarily regular instructions. The code growth is a direct result of the trace insertion heuristic used. More conservative or compiler implemented heuristics may be used to curb code growth, but were not attempted in this work.

Benchmark	DCTI % of Control Transfer Instructions
perlbench	12.2%
bzip2	9.6%
gcc	14.6%
bwaves	8.8%
mcf	9.8%
milc	9.7%
leslie3d	8.5%
gobmk	9.6%
sjeng	9.9%
libquantum	9.9%
h264ref	7.6%
lbm	9.6%
astar	11.6%
sphinx3	10.3%

Table 6.3
Delayed Instruction Conversion

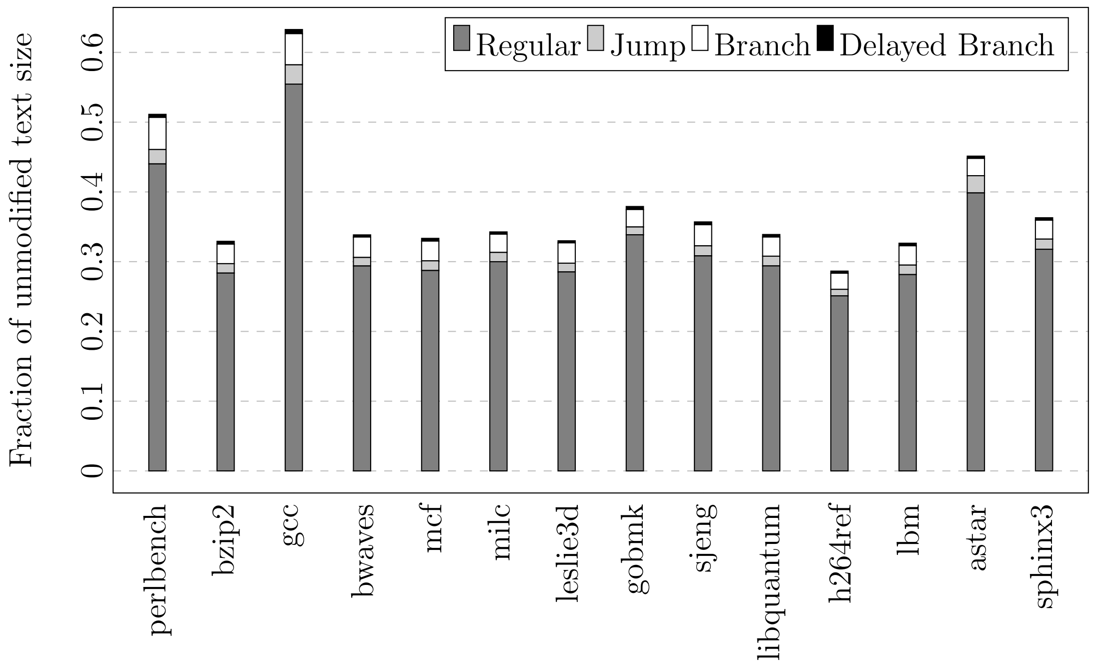


Figure 6.1: PMTC Trace Composition

6.3 Dynamic Trace Utilization

On average 24 percent of all control transfer instructions executed were delayed control transfer instructions. Table 6.4 shows the dynamic utilization of PMTC traces. Dynamic utilization reported shows the percent of all instructions retired that were fetched from a trace. Although the static program image size increases substantially, not all of the replicated instructions can be utilized. Trace instructions are used to rescue fetch slots. When fetch slots are not available, no instructions will be fetched from the trace. Fetching branches from a trace requires branch predictor throughput to be available.

Benchmark	Trace % of Retired Instructions
perlbench	11.17%
bzip2	11.48%
gcc	18.83%
bwaves	3.96%
mcf	10.76%
milc	3.67%
leslie3d	3.63%
gobmk	8.13%
sjeng	11.79%
libquantum	2.57%
h264ref	5.23%
lbm	0.01%
astar	12.7%
sphinx3	5.18%

Table 6.4
Dynamic Trace Utilization

The difference between static text composition and dynamic instruction execution suggests traces could be better utilized and more conservatively inserted. A more sophisticated algorithm could align targets that contain traces to cache line boundaries such that the trace would be within range of the available fetch slots. Target alignment was not explored in this work and ideal alignment is an open research problem.

6.4 Performance Evaluation

The benefits of PMTC included increased average instruction fetch width, increased branch prediction accuracy using explicit path separability and reduction of dynamic fetch cycles and i-cache fetch traffic. Metrics used to evaluate the processor configurations are speedup, reduction in branch mispredictions per thousand instructions (MPKI) and dynamic fetch cycle count.

Figure 6.2 shows the reduction in branch mispredictions for PMTC vs the baseline divided by miss category. In the graph, the first bar in each pair of bars corresponds to the baseline and the second to PMTC. The ideal baseline had comparable prediction accuracy to the baseline and is not displayed. As it can be seen, PMTC's reduction of branch mispredictions is its greatest strength which is due to its ability to exploit path separability of branches.

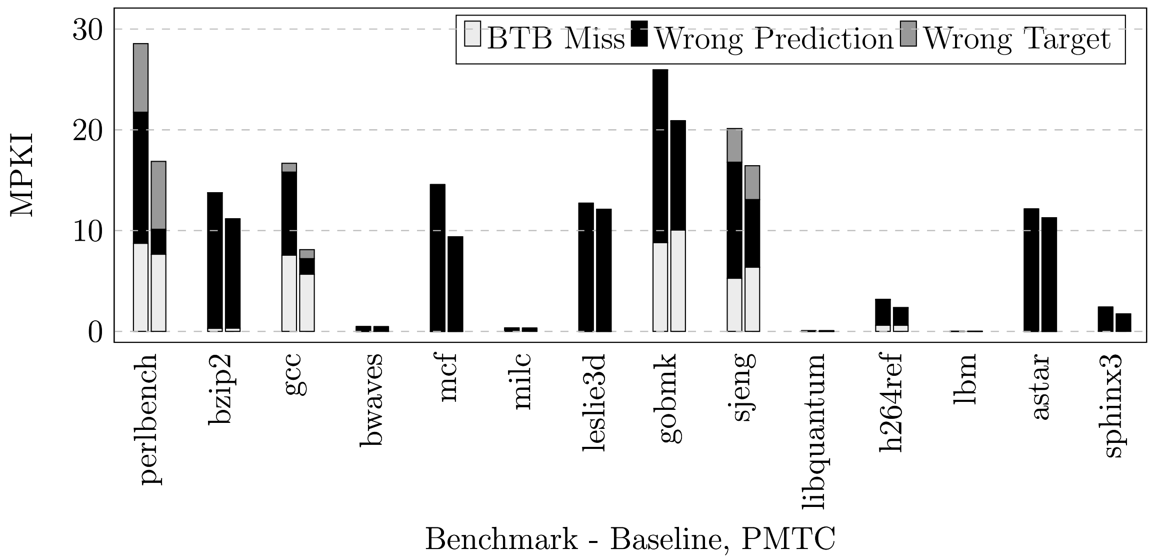


Figure 6.2: Branch Mispredictions per 1K Instructions

PMTC does not negatively effect branch history distributions which has been shown to be problematic in related work like Software Trace Cache that alters branch directions [52]. Added BTB pressure from replicated instructions does not lead to increased BTB misses in perlbench and gcc. Added BTB pressure does however increase the BTB miss rate for gobmk and sjeng, although the increase in BTB misses is not enough to overcome the positive effect on branch direction. Sensitivity to BTB size is discussed in Section 6.5.

Figure 6.3 shows the average fetch width for all configurations. We define average fetch width to be the number of instructions fetched (includes speculative instructions) divided by the number of fetch cycles. GMEAN fetch bandwidth for the Baseline, Ideal Baseline and PMTC is 5.95, 7.73 and 6.28 respectively. The Ideal Baseline does not reach the maximum fetch width of 8 instructions due to constraints imposed

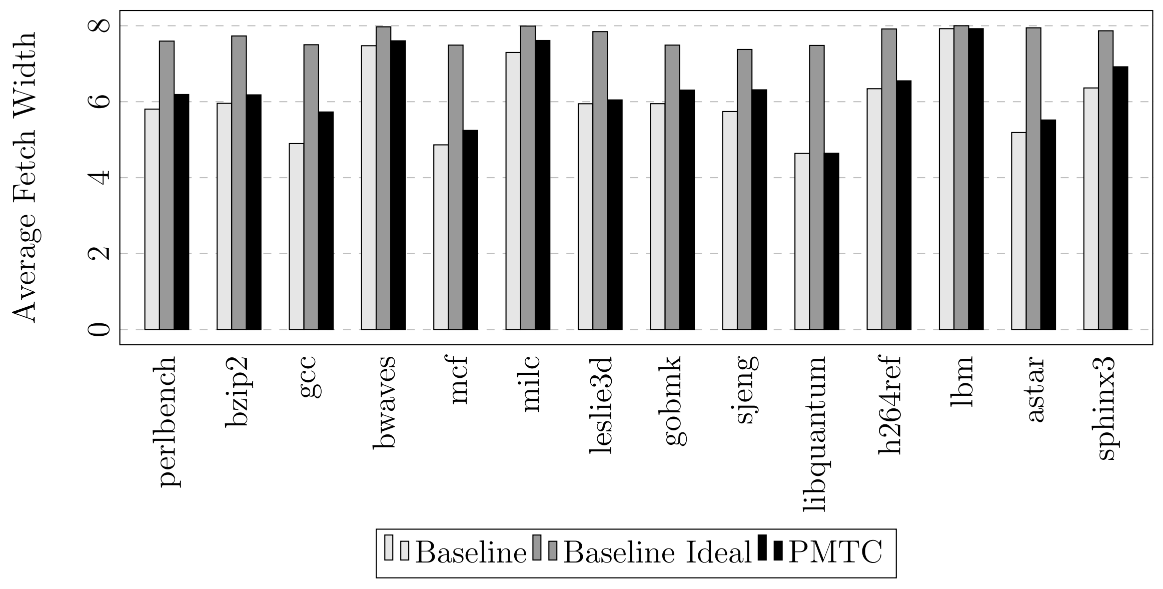


Figure 6.3: Average Fetch Width

by branch predictor throughput and cache residency. The results show that, while increasing prediction accuracy, PMTC also effectively increases fetch width in most cases. Increased fetch width is due to the ability to fetch beyond select taken DCTIs in a single cycle. Minimal increase in fetch bandwidth for libquantum and astar is due to low dynamic trace utilization.

Table 6.5 shows the reduction in dynamic fetch cycles for PMTC compared to the baseline. Only fetch cycles where the fetch unit was not stalled due to back end resource constraints or i-cache misses are displayed. Two contiguous sub blocks are fetched from the i-cache in each active fetch cycle. In some cases, the second sub block is not resident and instruction fetch is limited to instructions from the first sub block only. Reducing the number of fetch cycles directly translates to reduced i-cache traffic. The reduction in fetch cycles is due to PMTC's ability to fetch across taken

Benchmark	%
perlbench	8.98
bzip2	2.19
gcc	16.45
bwaves	1.67
mcf	5.99
milc	4.1
leslie3d	1.91
gobmk	3.05
sjeng	5.94
libquantum	0.11
h264ref	3.27
lbm	0.001
astar	6.44
sphinx3	7.76

Table 6.5
% Reduction in Dynamic Fetch Cycles

control transfer instructions and eliminate the need to fetch intermediate targets via control transfer chain reduction. Although one might expect the modified i-cache access pattern combined with the text segment size increase to cause an increase in the i-cache miss rate, we observed that the increase in miss rate was negligible for all benchmarks. Sensitivity to i-cache size is discussed in Section 6.5.

Figure 6.4 shows the overall speedup of PMTC and the ideal fetch unit over the baseline. PMTC has a GMEAN speedup of 5.037 percent over the baseline. The ideal fetch unit has a GMEAN speedup of 4.639 percent over the baseline. Although the fetch width increase in PMTC does not compare to the ideal fetch unit, comparable performance is achieved. In some benchmarks the performance improvements of

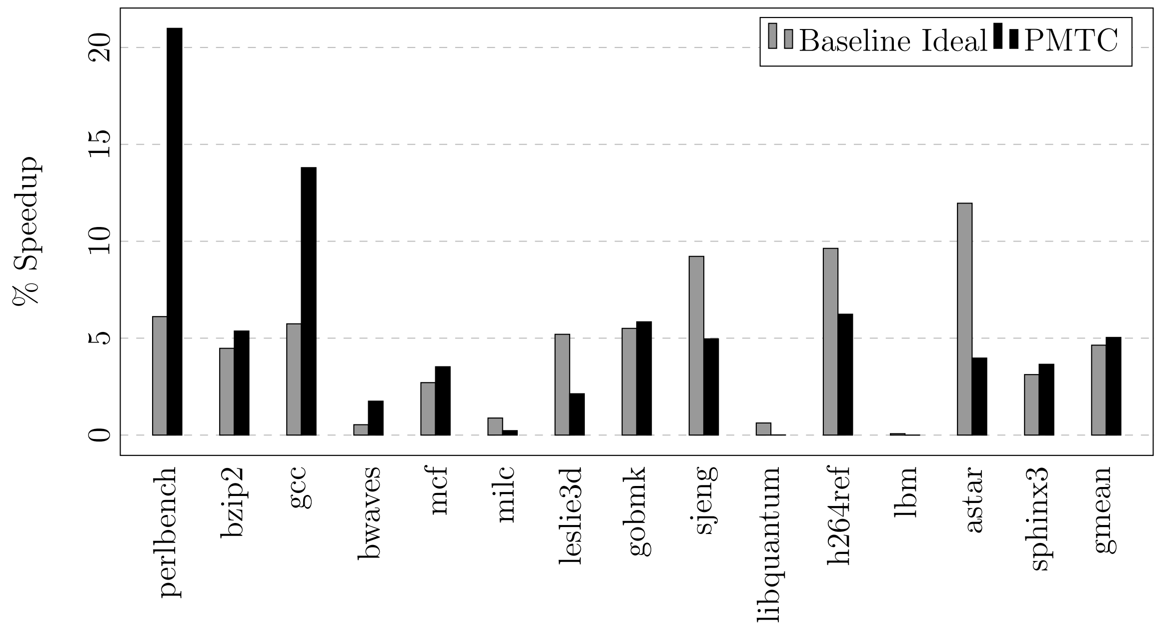


Figure 6.4: Percent Speedup of PMTC and Baseline Ideal

PMTC exceed the ideal fetch unit. The ability to exceed the ideal fetch unit's performance shows just how important improving branch prediction accuracy is. Small increases in fetch bandwidth, combined with reduced i-cache traffic and substantial reduction in branch mispredictions leads to greater average speedup using PMTC.

Benchmarks libquantum and lbm have very low branch misprediction rates to begin with and exhibit very low trace utilization. Thus, they do not show any significant gain in branch prediction accuracy, increase in fetch width or reduction in fetch cycles. Without observing any of the typical benefits of PMTC, the speedup over the baseline is negligible. Branch accuracy is already very high for bwaves and milc and prediction improvements in bwaves and milc are minimal. Benchmark milc shows modest improvement in fetch width and reduction in fetch cycles, providing the small speedup over the baseline. Benchmark bwaves sees a small increase in fetch width and slight reduction in fetch cycles. Although dynamic trace utilization is low in bwaves, 50% of all branches executed are delayed branches. This suggests that the reduction in branch mispredictions is the most significant benefit of PMTC in regards to speedup. The significant increase in prediction accuracy paired with minor fetch width increase and fetch cycle reduction in perlbench yielding the highest speedup supports this claim. Considering all of the results shown we claim that benefits to branch prediction accuracy via path separability have the most significant impact on overall speedup of all the metrics presented.

6.5 Sensitivity Analysis

Figure 6.5 shows the geometric mean speedup across the Spec suite for configurations with varying i-cache hit latency, i-cache size and BTB size. I-cache size was varied in an attempt to observe the effect of added i-cache pressure as Spec 2006 has a relatively small i-cache footprint. BTB size was varied to observe the effect of added pressure from PMTC's replicated instructions. With a BTB size of 256 entries the PMTC algorithm presented still provides positive results. The trend shown in Section 6.4 is observed albeit with lower gains as the BTB size decreases. The outlier is astar, which reduces performance by 5% when the BTB is 256 entry. With BTB sizes above 256, astar shows positive results of up to 4%. We varied other pipeline parameters not shown, but the primary sensitivity in regards to PMTC was to BTB size.

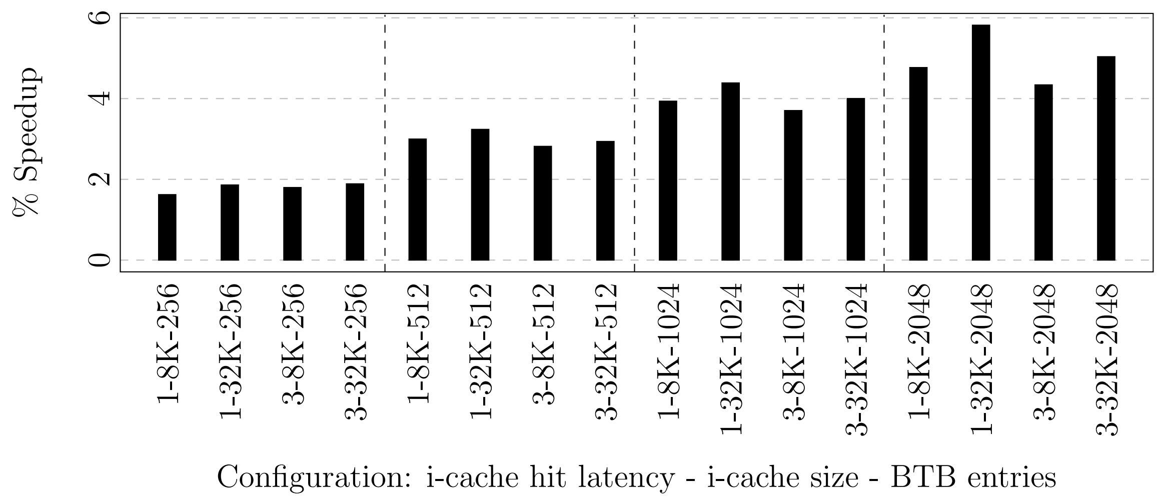


Figure 6.5: Average Speedup Compared to Baseline

We also experimented with 4 - 32 instruction L1 cache line sizes and 2-8 way associativity. PMTC specifically did not appear particularly sensitive. PMTC and the baseline did show similar variation in hit rates. Instead the sensitivity in this regard is to the PMTC trace size. PMTC trace size can be configured as any power of 2 subset within the cache line size. Trace sizes of 4, 8 and 16 were tried. Using 4 instruction traces provided some benefit, but we found that 8 instruction traces proved more effective. Traces with 16 instructions proved too large as many branches were copied into the trace, but not enough branch predictor throughput was available to consider the entire trace. Thus, part of those longer traces are never issued and the space is wasted.

6.6 Power Evaluation

Power evaluation for the pipeline and memory system was conducted using McPAT [32] assuming 22 nm process technology and a clock rate of 2.5 GHz. Energy for the Address Generation Unit for PMTC was estimated using the results presented in [68] for 4 bit and 32 bit adders assuming 22 nm process technology. Approximated energy per access of the Address Generation Unit is .0236 pJ and considered to be negligible with respect to CPU energy as the Address Generation Unit is only accessed once per fetch cycle. The most significant energy computed for the Address Generation Unit was 741 nJ which accounted for significantly less than 1 percent of the total CPU energy. Although a trace cache was not directly implemented in the Baseline and compared, we expect that the relative energy difference would be even more significant. Thus, we consider PMTC to be an energy efficient trace utilization technique.

Table 6.6 shows percent energy reduction and percent reduction in energy-delay^2 product (EDDP) for PMTC compared to the Baseline. On average, PMTC leads to 5.1 percent reduction in Energy and 10.7 percent reduction in EDDP. Energy reduction observed is primarily due to 2 effects of using PMTC:

1. The dynamic fetch cycle count has decreased compared to the baseline resulting

in reduced i-cache traffic.

2. Branch prediction accuracy has increased compared to the baseline reducing the amount of wasted work.

Benchmark	% Energy Reduction	% EDDP Reduction
perlbench	19	44.66
bzip2	5.43	13.97
gcc	13.25	7.73
bwaves	1.86	4.39
mcf	4.22	9.77
milc	0.57	1.15
leslie3d	2.36	6.14
gobmk	5.46	14.66
sjeng	4.79	13.17
libquantum	0.02	0.01
h264ref	6.13	16.94
lbm	0	0
astar	4.79	7.15
sphinx3	4.25	10.85

Table 6.6
Energy Reduction

Fewer wrong path speculative instructions proceed through the pipeline, thus reducing the accesses to the most power hungry components.

Chapter 7

Related Work and Conclusions

Increasing effective instruction fetch bandwidth has been a major research topic spanning several decades. Most work to date focuses on increasing the fetch bandwidth and attempt to reduce the branch mispredictions by targeting the predictor itself. We believe PMTC is the first work to show that it is possible to improve the fetch bandwidth while improving the predictor accuracy in a consistent manner, using the same basic mechanism.

The sheer number of related publications alone makes it extremely difficult to cover the prior art in a comprehensive manner. Therefore, we give an overview of the design space using a taxonomy shown in Figure 7.1. While the taxonomy itself is a gross over-simplification, it allows us to discuss some of the most relevant work.

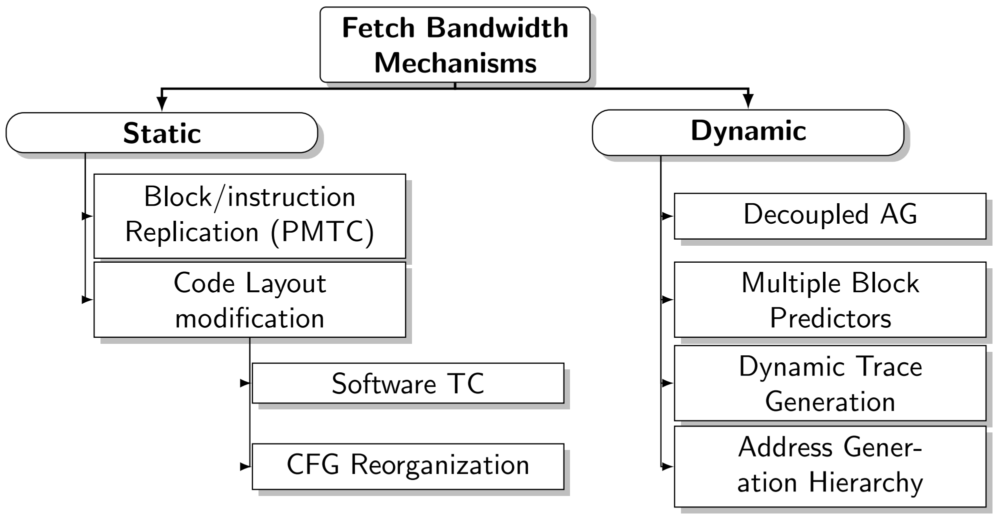

Fetch Bandwidth Mechanisms
Static
Dynamic
Block/instruction Replication (PMTC)
Code Layout modification
Software TC
CFG Reorganization
Decoupled AG
Multiple Block Predictors
Dynamic Trace Generation
Address Generation Hierarchy

Figure 7.1: Targeting Fetch Bandwidth

Conventional delayed branching permits executing an instruction which follows a branch unconditionally. Such delay slots have been used in older reduced instruction set (RISC) architectures to avoid control hazards. Although delay slots can be effective in simple single-issue pipelines, they create difficulties for processors which fetch multiple instructions. Such delay slots also cannot contain control transfer instructions as doing so would render the delay-slot semantics invalid, unless exception behavior is properly tracked [31, 70]. In contrast, PMTC delay slots are variable in length and may contain other control transfer instructions. To the best of our knowledge, PMTC is the first technique to utilize the concept of delay slots in this manner.

Our taxonomy classifies existing techniques for improving fetch bandwidth broadly into *static* and *dynamic* techniques. Most work to date fall under the dynamic category. Generally speaking, these techniques either attempt to generate multiple fetch addresses and then, in an out-of-order manner fetch the required instructions, or cache a dynamic instruction stream, as in various trace cache mechanisms. Notably, an earlier mechanism, *branch target instruction cache* (BTIC) caches instructions from branch targets and can supply additional instruction(s) given a taken branch PC [8]. A much more elaborate mechanism, Trace Cache actually caches the dynamic instruction traces as they are encountered.

It is possible to view PMTC as a technique which stores traces into the instruction cache by using the concept of delayed branching and instruction replication. Nevertheless, PMTC is a static technique as traces are formed at assembly time. In this respect, it is possible to call PMTC a *Software Trace Cache* (STC). An earlier work [52] with this title however does not insert traces into existing code, but rather rearranges the control-flow graph to make the program more amenable to the natural sequential fetching of the fetch unit. STC causes negative interaction with branch prediction as branch history distribution reduces as branches directions become more biased towards not taken. PMTC does not alter the program CFG, instead, it embeds traces in the existing program structure. Interestingly, storing dynamic traces at the expense of significant storage is most useful when the sequential nature of the instruction stream is disrupted by a taken branch. As well, this is also precisely when a PMTC trace is utilized. In this respect, we believe PMTC is unique in showing the dynamic construction and caching of traces may not be worth the cost.

Statically, a given program can be modified in one of two ways. Either the basic block contents can be modified, or the basic block organization can be modified. The *Block/Instruction replication* category is for techniques like PMTC that statically add instructions into a block. It must be noted that PMTC requires micro-architectural modifications to handle delayed branches, but these modifications do not involve dynamic analysis of the fetch frontier. Therefore, under this view we still consider PMTC to be static. Techniques have been proposed to organize CFGs in a tree like structure to identify block chains [12]. Alternatively software traces can be constructed within the CFG using advanced placement algorithms [52]. In this case, the traces are built from chains of blocks rather than employing a more complicated structure. Most existing static techniques utilize code layout modification through CFG reorganization or a grouping mechanism [47] [49] [51] [14] [17] [59]. Some static code reorganization is done based on profiling [46]. Static branch alignment has also been implemented in static code to improve performance [5] as well as reorganization to alter branch directions [50]. [44] increased branch-less regions by replicating branches with assertions into frames, similar in some respect to PMTC although atomic in nature. Compiler optimization and reordering techniques have been developed to target specific fetch architectures like the collapsing buffer [11] and trace cache [43] as well as instruction cache performance [20] [29] [7] [15] [2]. Block based ISA has also been explored to increase fetch rate [16].

During execution the processor can use multiple branch and block prediction to predict targets corresponding to noncontiguous basic blocks [73] [63] [69] [12] [53]. Alternatively a dynamic instruction trace can be generated and cached [58] [25] [3] [67] [27] and predicted [23]. This trace can be reused when encountered again. Other techniques seek to reduce the latency associated with multiple branch prediction via decoupling or early issue of address generation [55] [56]. Alternatively, a hierarchy of address generators can be used [62]. A fast generator immediately provides a prediction and a slower, but more accurate generator corrects when the fast generator does not agree. Superscalar machine design has been combined with VLIW techniques to increase issue rates [13] as well as multiple fetch streams [38] [39]. Duplicate issue within loops has been explored to reduce i-cache accesses [48] as well as loop caching [54]. Most dynamic techniques require substantial hardware changes. [6] uses an additional target cache to separate paths for indirect jumps.

One of the significant results of the PMTC approach is the improvement in branch prediction accuracy. This improvement is due to a branch instruction having multiple *home locations* via the use of actual addresses while accessing predictors. These results strongly indicate that any technique that attempts to increase front end fetch bandwidth should consider branch prediction or how the technique will interact with the prediction algorithms.

PMTC is also extremely versatile for deployment in future processors. Although we assumed that any unconditional direct branch can be given a delay slot, introducing delayed versions of unconditional direct branches will permit a PMTC microarchitecture to execute non PMTC-enhanced code without a problem. In this respect, PMTC can be deployed in existing processors without breaking backwards code compatibility. Better yet, our algorithm that is implemented in the assembler can easily be transported into a binary rewriting system. Hence, legacy code can be optimized for execution on a PMTC architecture as well.

References

[1] J. Aragon, J. Gonzalez, J. Garcia, and A. Gonzalez, “Selective branch prediction reversal by correlating with data values and control flow,” in *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, 2001, pp. 228–233.

[2] Azam Beg and Yul Chu, “Improved instruction fetching with a new block-based cache scheme,” in *International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005.*, vol. 2, 2005, pp. 765–768 Vol. 2.

[3] B. Black, B. Rychlik, and J. P. Shen, “The block-based trace cache,” in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, 1999, pp. 196–207.

[4] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, Jun. 1997. [Online]. Available: http://doi.acm.org/10.1145/268806.268810

[5] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: Association for Computing Machinery, 1994, pp. 242–251. [Online]. Available: https://doi.org/10.1145/195473.195553

[6] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target prediction for indirect jumps," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 274–283. [Online]. Available: https://doi.org/10.1145/264107.264209

[7] B. Chen, L. Li, Y. Li, H. Luo, and D. Guo, "Compiler assisted instruction relocation for performance improvement of cache hit rate and system reliability," in *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, 2014, pp. 243–246.

[8] N. K. Choudhary, M. S. Mcilvaine, D. E. Streett, V. K. Reddy, S. S. Srikantaiah, S. S. Navada, R. D. Clancy, J. N. Dieffenderfer, and T. A. Sartorius, "Branch target instruction cache (btic) to store a conditional branch instruction," Patent 20 170 083 333, March, 2017. [Online]. Available: http://www.freepatentsonline.com/y2017/0083333.html

[9] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store

sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA 98. USA: IEEE Computer Society, 1998, pp. 142–153. [Online]. Available: https://doi.org/10.1145/279358.279378

[10] G. Chrysos and J. Emer, "Memory dependence prediction using store sets," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, July 1998, pp. 142–153.

[11] 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*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 333–344. [Online]. Available: https://doi.org/10.1145/223982.224444

[12] S. Dutta and M. Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, ser. MICRO 28. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, pp. 258–263. [Online]. Available: http://dl.acm.org/citation.cfm?id=225160.225201

[13] M. Franklin and M. Smotherman, "A fill-unit approach to multiple instruction issue," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: Association

for Computing Machinery, 1994, pp. 162–171. [Online]. Available: https://doi.org/10.1145/192724.192748

[14] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder, "Procedure placement using temporal ordering information," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 303–313.

[15] A. Gordon-Ross, F. Vahid, and N. Dutt, "Combining code reordering and cache configuration," *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2362336.2399177

[16] E. Hao, Po-Yung Chang, M. Evers, and Y. N. Patt, "Increasing the instruction fetch rate via block-structured instruction set architectures," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996, pp. 191–200.

[17] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," *SIGPLAN Not.*, vol. 32, no. 5, pp. 171–182, May 1997. [Online]. Available: https://doi.org/10.1145/258916.258931

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Elsevier, 2011.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1 – 17, Sep. 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[20] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89. New York, NY, USA: Association for Computing Machinery, 1989, pp. 242–251. [Online]. Available: https://doi.org/10.1145/74925.74953

[21] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, ser. ISCA '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 18–26. [Online]. Available: https://doi.org/10.1145/30350.30353

[22] Q. Jacobson, S. Bennett, N. Sharma, and J. Smith, "Control flow speculation in multiscalar processors," in *Proceedings Third International Symposium on High-Performance Computer Architecture*, 1997, pp. 218–229.

[23] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 14–23.

[24] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. USA: IEEE Computer Society, 1997, p. 14–23.

[25] J. Jaison and P. K. Mukherjee, “Modified selective way based trace cache,” in *2014 International Conference on Electronics and Communication Systems (ICECS)*, 2014, pp. 1–4.

[26] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, p. 364–373, may 1990. [Online]. Available: https://doi.org/10.1145/325096.325162

[27] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, and R. Ronen, “extended block cache,” in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, 2000, pp. 61–70.

[28] D. R. Kaeli and P. G. Emma, “Branch history table prediction of moving target branches due to subroutine returns,” *ACM SIGARCH Computer Architecture News*, vol. 19, no. 3, pp. 34–42, 1991.

[29] J. Kalamationos and D. R. Kaeli, “Temporal-based procedure reordering for improved instruction cache performance,” in *Proceedings 1998 Fourth International Symposium on High-Performance Computer Architecture*, 1998, pp. 244–253.

[30] Lee and Smith, “Branch prediction strategies and branch target buffer design,” *Computer*, vol. 17, no. 1, pp. 6–22, 1984.

[31] R. B. Lee and A. J. Baum, "Bidirectional branch prediction and optimization," 1985, uS Patent US4755966A. [Online]. Available: https://patents.google.com/patent/US4755966A/en

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, pp. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[33] S. Manne, A. Klauser, and D. Grunwald, "Branch prediction using selective branch inversion," in *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*, 1999, pp. 48–56.

[34] S. Mcfarling, "Combining branch predictors," Tech. Rep., 1993.

[35] S. Melvin and Y. Patt, "Enhancing instruction scheduling with a block-structured isa," *Int. J. Parallel Program.*, vol. 23, no. 3, p. 221–243, jun 1995. [Online]. Available: https://doi.org/10.1007/BF02577867

[36] S. Mittal, "A survey of recent prefetching techniques for processor caches," *ACM Comput. Surv.*, vol. 49, no. 2, aug 2016. [Online]. Available: https://doi.org/10.1145/2907071

[37] R. Nair, "Dynamic path-based branch correlation," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, ser. MICRO 28. Washington, DC, USA: IEEE Computer Society Press, 1995, p. 15–23.

[38] P. Oberoi and G. Sohi, "Out-of-order instruction fetch using multiple sequencers," in *Proceedings International Conference on Parallel Processing*, 2002, pp. 14–23.

[39] P. S. Oberoi and G. S. Sohi, "Parallelism in the front-end," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003, pp. 230–240.

[40] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proceedings of the 1998 International Conference on Computer Languages*, ser. ICCL '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 80–. [Online]. Available: http://dl.acm.org/citation.cfm?id=857172.857236

[41] S. Onder and R. Gupta, "Dynamic memory disambiguation in the presence of out-of-order store issuing," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 32. Washington, DC, USA: IEEE Computer Society, 1999, pp. 170–176. [Online]. Available: http://portal.acm.org/citation.cfm?id=320080.320105

[42] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary

cache replacement," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, p. 24–33, apr 1994. [Online]. Available: https://doi.org/10.1145/192007.192014

[43] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, 1998, pp. 262–271.

[44] S. J. Patel, T. Tung, S. Bose, and M. M. Crum, "Increasing the size of atomic instruction blocks using control flow assertions," in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, 2000, pp. 303–313.

[45] C. Perleberg and A. Smith, "Branch target buffer design and optimization," *IEEE Transactions on Computers*, vol. 42, no. 4, pp. 396–412, 1993.

[46] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: Association for Computing Machinery, 1990, pp. 16–27. [Online]. Available: https://doi.org/10.1145/93542.93550

[47] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi, "Control flow prediction for dynamic ilp processors," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, ser. MICRO 26. Los Alamitos, CA,

USA: IEEE Computer Society Press, 1993, pp. 153–163. [Online]. Available: http://dl.acm.org/citation.cfm?id=255235.255277

[48] P. Rajamani, J. P. Shah, V. Sankaranarayanan, and R. Sangireddy, "High performance and alleviated hot-spot problem in processor frontend with enhanced instruction fetch bandwidth utilization," in *2006 IEEE International Performance Computing and Communications Conference*, 2006, pp. 8 pp.–70.

[49] A. Ramirez, J. L. Larriba-Pey, C. Navarro, X. Serrano, M. Valero, and J. Torrellas, "Optimization of instruction fetch for decision support workloads," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 238–245.

[50] A. Ramirez, J. L. Larriba-Pey, and M. Valero, "The effect of code reordering on branch prediction," in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, 2000, pp. 189–198.

[51] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 371–382.

[52] A. Ramírez, J.-L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *ACM International Conference on Supercomputing 25th Anniversary Volume.* New York, NY, USA: Association for Computing

Machinery, 1999, p. 261–268. [Online]. Available: https://doi.org/10.1145/2591635.2667175

[53] N. Ranganathan, R. Nagarajan, D. Jiménez, D. Burger, S. W. Keckler, and C. Lin, “Combining hyperblocks and exit prediction to increase front-end bandwidth and performance,” Tech. Rep., 2002.

[54] M. Rawlins and A. Gordon-Ross, “Adaptive loop caching using lightweight runtime control flow analysis,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2435227.2435251

[55] G. Reinman, B. Calder, and T. Austin, “Optimizations enabled by a decoupled front-end architecture,” *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 338–355, 2001.

[56] G. Reinman, T. Austin, and B. Calder, “A scalable front-end architecture for fast instruction delivery,” *SIGARCH Comput. Archit. News*, vol. 27, no. 2, p. 234–245, May 1999. [Online]. Available: https://doi.org/10.1145/307338.300999

[57] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “Dramsim2: A cycle accurate memory system simulator,” *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2011.4

[58] E. Rotenberg, S. Bennett, and J. Smith, “Trace cache: a low latency approach to high bandwidth instruction fetching,” in *Proceedings of the 29th Annual*

IEEE/ACM International Symposium on Microarchitecture. MICRO 29, 1996, pp. 24–34.

[59] O. J. Santana, A. Ramirez, and M. Valero, "Reducing fetch architecture complexity using procedure inlining," in *Eighth Workshop on Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004.*, 2004, pp. 97–106.

[60] A. Seznec, "A 256 kbits l-tage branch predictor," *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.

[61] ——, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 117–127. [Online]. Available: https://doi.org/10.1145/2155620.2155635

[62] A. Seznec and A. Fraboulet, "Effective ahead pipelining of instruction block address generation," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ser. ISCA '03. New York, NY, USA: ACM, 2003, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/859618.859646

[63] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*,

ser. ASPLOS VII. New York, NY, USA: ACM, 1996, pp. 116–127. [Online]. Available: http://doi.acm.org/10.1145/237090.237169

[64] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, p. 23, Feb. 2006. [Online]. Available: https://hal.inria.fr/hal-03408381

[65] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGPLAN Not.*, vol. 37, no. 10, pp. 45–57, Oct. 2002. [Online]. Available: https://doi.org/10.1145/605432.605403

[66] D. Sima, T. Fountain, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures: A Design Space Approach*, ser. International computer science series. Addison-Wesley, 1997. [Online]. Available: https://books.google.com/books?id=AdVQAAAAMAAJ

[67] J. E. Smith and S. Vajapeyam, "Trace processors: moving to fourth-generation microarchitectures," *Computer*, vol. 30, no. 9, pp. 68–74, 1997.

[68] R. Suganya and D. Meganathan, "High performance vlsi adders," in *2015 3rd International Conference on Signal Processing, Communication and Networking (ICSCN)*, 2015, pp. 1–7.

[69] S. Wallace and N. Bagherzadeh, "Multiple branch and block prediction," in *Proceedings Third International Symposium on High-Performance Computer Architecture*, 1997, pp. 94–103.

[70] F. Worrell, “Enhanced branch delay slot handling with single exception program counter,” 1995, uS Patent US5774709A. [Online]. Available: https://patents.google.com/patent/US5774709A/en

[71] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, “Increasing the instruction fetch rate via multiple branch prediction and a branch address cache,” in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS ’93. New York, NY, USA: Association for Computing Machinery, 1993, p. 67–76. [Online]. Available: https://doi.org/10.1145/165939.165956

[72] 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*, ser. ISCA ’92. New York, NY, USA: Association for Computing Machinery, 1992, p. 124–134. [Online]. Available: https://doi.org/10.1145/139669.139709

[73] T. Yeh, D. T. Marr, and Y. N. Patt, “Increasing the instruction fetch rate via multiple branch prediction and a branch address cache,” in *Proceedings of the 7th International Conference on Supercomputing*, ser. ICS ’93. New York, NY, USA: ACM, 1993, pp. 67–76. [Online]. Available: http://doi.acm.org/10.1145/165939.165956