

Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

11-18-2002

An Investigation of trace cache organizations for superscalar processors

Edward Mulrane Jr

Follow this and additional works at: <https://repository.rit.edu/theses>

Recommended Citation

Mulrane, Edward Jr, "An Investigation of trace cache organizations for superscalar processors" (2002). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

An Investigation of Trace Cache Organizations for Superscalar Processors

by

Edward M. Mulrane Jr.

November 18, 2002

A thesis submitted in partial fulfillment of
the requirements for the degree of

Masters of Science in
Computer Engineering

Rochester Institute of Technology

Primary Advisor: Dr. Muhammad Shaaban, Assistant Professor

Advisor: Dr. Roy Czernikowski, Professor

Advisor: Dr. Ken Hsu, Professor

Release Permission Form

Rochester Institute of Technology

An Investigation of Trace Cache Organizations
for Superscalar Processors

I, Edward M. Mulrane Jr., hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part, on or after November 18, 2002. Any reproduction will not be for commercial use or profit.

Edward M. Mulrane Jr.

11-18-02

Date

Abstract

Techniques such as out-of-order issue and speculative execution aggressively exploit instruction level parallelism in modern superscalar processor architectures. The front end of such pipelined machines is concerned with providing a stream of schedulable instructions at a bandwidth that meets or exceeds the rate of instructions being issued and executed. As superscalar machines become increasingly wide, it is inevitable that the large set of instructions to be fetched every cycle will span multiple noncontiguous basic blocks. The mechanism to fetch, align, and pass this set of instructions down the pipeline must do so as efficiently as possible, occupying a minimal number of pipeline cycles. The concept of trace cache has emerged as the most promising technique to meet this high-bandwidth, low-latency fetch requirement. This thesis presents the design, simulation and analysis of a microarchitecture simulator extension that incorporates trace cache. A new fill unit scheme, the Sliding Window Fill Mechanism is proposed. This method exploits trace continuity and identifies probable start regions to improve trace cache hit rate. A 7% hit rate increase was observed over the Rotenberg fill mechanism. Combined with branch promotion, trace cache hit rates experienced a 19% average increase along with a 17% average rise in fetch bandwidth.

Contents

Abstract	i
Table of Contents	ii
List of Figures	v
List of Tables	vii
Glossary	viii
Acknowledgments & Trademarks	xiv
1 Introduction	1
1.1 Motivation for Trace Cache and Instruction Pre-processing	1
1.2 Organization of Thesis	2
2 Background	3
2.1 Emergence of the Superscalar Processor	3
2.2 Tasks of Superscalar Processing	5
2.3 Issue Policies	8
2.4 Processing of Control Transfer Instructions	13
2.5 Elements of the Instruction Fetch Subsystem	19
2.6 The SimpleScalar Simulator Architecture	26
3 Trace Cache Schemes	29
3.1 Concept of a Dynamic Trace	29
3.2 Rotenberg Trace Cache	30
3.3 Path-based Next-trace Prediction	33
3.4 Branch Promotion and Trace Packing	37
3.5 Fetch Address Renaming	39
3.6 Partial Matching and Inactive Issue	40
3.7 Trace Preconstruction	41
3.8 Block Cache	44
3.9 Cost Regulated Trace Packing	46

3.10	Selective Trace Storage.....	47
3.11	Fill Unit Optimizations	47
3.12	An Alternate Approach to Filling the Trace Cache.....	50
3.13	Additional Related Research	51
3.14	Trace Cache Implementation: The Pentium 4	53
4	Simulator Extension Design	55
4.1	Existing Superscalar Simulator	55
4.2	Motivation for Utilizing an Object-Oriented Extension	57
4.3	Modifications to <i>sim-outorder</i> Source.....	57
4.4	High Level Design	59
4.5	Component Level Classes	60
4.6	Creational Level Classes	62
4.7	Helper Level Classes.....	63
5	Implementation and Results	65
5.1	Simulation Strategy.....	65
5.2	Benchmarks Used	67
5.3	Multiple Branch Predictors.....	69
5.4	Rotenberg Trace Cache.....	74
5.5	Trace Cache Associativity.....	77
5.6	Reassociation.....	80
5.7	Branch Promotion	81
5.8	Alternate Fill Mechanism.....	85
5.9	Trace Continuity and Probable Entry Points.....	87
5.10	The Sliding Window Fill Mechanism & Fill Selection Table	89
5.11	Combining Trace Cache Schemes	94
5.12	Other Simulated Schemes.....	96
5.13	Side Effects Associated with Trace Cache	98
6	Conclusion	100
6.1	Summary of Work.....	100
6.2	Difficulties Encountered.....	101
6.3	Recommendations for Future Work	102
	References	104
	Appendix A – Trace Cache Extension Header Files	
	Appendix B – Kernel Benchmark Source Files	

Appendix C – Benchmark Input Sets

Appendix D – Example Simulator Output File

List of Figures

<i>Number</i>		<i>Page</i>
2.1	- Functional Processor Model.....	5
2.2	- Issue Policy Design Space.....	8
2.3	- Methods for Handling Issue Blockages	9
2.4	- Shelving	11
2.5	- Design Space Outline of Tomasulo's Algorithm	13
2.6	- Local (PAg) Predictor.....	17
2.7	- Gshare Predictor.....	17
2.8	- General Combined Predictor.....	18
2.9	- Fetch Subsystem Components	21
2.10	- MGAg Predictor.....	23
2.11	- Collapsing Buffer.....	24
2.12	- Branch Address Cache	25
2.13	- Design Space Outline of <i>simoutorder</i>	28
3.1	- Static Program Order versus Dynamic Execution Order.....	30
3.2	- Example Trace Cache Segment.....	31
3.3	- The Rotenberg et al Trace Cache	32
3.4	- Combining Trace Predictor.....	34
3.5	- Hashed History Register Entry.....	35
3.6	- DOLC Index Generation	35
3.7	- Static Code and Corresponding Preconstruction Graph.....	42
3.8	- Extended Pipeline Model.....	44
3.9	- Conventional versus Blocked-Based Trace Cache	45
3.10	- Register-to-register move optimization	49
3.11	- Rotenberg Fill versus the Fill Policy of [20]	51
4.1	- <i>Simoutorder</i> Pipeline	56
4.2	- Uniform Trace Cache Model	59
4.3	- UML Diagram of the Abstract Component Classes	61
4.4	- UML Diagram of the Abstract Component Classes (cont.)	62
4.5	- UML Diagram of the Creational Level Classes	63

5.1	-	Multiple Branch Predictor Accuracy.....	72
5.2	-	Fetch Bandwidth Increase vs. m	73
5.3	-	Prediction Accuracy vs. m	73
5.4	-	Average Trace Cache Segment Length	74
5.5	-	Fetch Bandwidth after Incorporating Trace Cache	75
5.6	-	Trace Cache Hit Rate.....	76
5.7	-	Trace Cache Speedup	76
5.8	-	Distribution of Trace-Terminating Conditions	77
5.9	-	Hit Rate vs. Associativity	78
5.10	-	Utilization vs. Associativity.....	79
5.11	-	Speedup vs. Associativity	79
5.12	-	Average Fetch Bandwidth vs. Branch Bias Threshold.....	82
5.13	-	Speedup vs. Branch Bias Threshold	82
5.14	-	Speedup Comparison of BBT Organizations	84
5.15	-	Speedup using Branch Promotion	85
5.16	-	Hit Rate for Alternate Fill Scheme.....	86
5.17	-	Unique Traces Added	86
5.18	-	Speedup Using Alternate Fill Scheme	87
5.19	-	Fill Unit Example	88
5.20	-	The Sliding Window Fill Mechanism	90
5.21	-	Hit Rate for SWFM using various FST Thresholds	93
5.22	-	Speedup using the SWFM/FST	93
5.23	-	Hit Rate Increase for Combined Scheme	94
5.24	-	Fetch Bandwidth Increase for Combined Scheme	95
5.25	-	Decrease in Branch Prediction Accuracy for Combined Scheme.....	96

List of Tables

<i>Number</i>		<i>Page</i>
3.1	- Trace Cache Limitations and Solutions.....	33
5.1	- Base <i>simoutorder</i> parameters	66
5.2	- Benchmark Set Utilized.....	68
5.3	- Average Basic Block Sizes.....	69
5.4	- Reassociation Statistics.....	81
5.5	- Resulting Example Traces.....	88
5.6	- Conditions for Adjusting the <i>trace tail</i> Pointer.....	91
5.7	- Number of Unique Traces Added using SWFM/FST	92

Glossary

Block Cache: Structure that stores basic blocks, which are referenced by the trace table in a cost reduced trace cache scheme.

Branch Address Cache: An alternate fetch mechanism aimed at improving fetch bandwidth.

Branch Address Prediction: A generic term for any technique that provides a target address for an instruction before it is decoded/evaluated.

Branch Bias Table (BBT): A structure used in branch promotion that records the number of successive taken/not-taken directions for a particular branch.

Branch Direction Prediction: A generic term for any technique that provides the branch direction for an instruction before it is decoded/evaluated.

Branch Flags: Bits indicating the direction of each conditional branch within a trace.

Branch Mask: A field indicating the number of conditional branch within a trace.

Branch Promotion: A scheme that identifies strong taken/not-taken branches and removes them from the consideration of the dynamic branch predictor.

Branch Target Buffer (BTB): A table that returns a branch target when indexed with the address of a control transfer instruction. Serves as a Branch Address Prediction method.

Collapsing Buffer: An alternate fetch mechanism aimed at improving fetch bandwidth.

Combined Predictor: A technique that allows various types of predictors to be utilized in tandem, recording the method that proves more accurate for individual branches.

Control transfer instruction: Any operation that alters the sequential execution of a program (jumps, conditional branches, subroutine calls and returns).

Cost-reduced BBT: Similar to the Merged BTB/BBT with the exception that only taken targets & bias values are recorded.

Cost-regulated Trace Packing: A fill unit scheme that only permits full trace packing under certain conditions in a trade off between resource consumption and trace continuity.

Decoded Instruction Queue (DIQ): Internal structure to the decode/issue stage that buffers decoded instructions prior to being issued to the dynamic scheduling mechanism.

Degree of Speculativeness: How far down the pipeline a speculatively executed instruction is allowed to proceed.

DOLC: “Depth, Older, Last, Current”. A strategy to generate an index value from a set of history register entries.

Dynamic Execution Order: The order in which instructions are processed as a program executes.

Effective Issue Bandwidth: How many instructions on average are issued per cycle.

Extended Pipeline Model: Abstraction of a processor from the perspective of three independent pipelines: Pre-processing, Fetch/Decode and Execute.

Fall-through Address: The address corresponding to a not-taken result for a trace-terminating branch.

Fill Select Table (FST): Identifies probable entry points by maintaining a count for every PC accessed by the core fetch unit.

Full BBT: The implementation of a BBT as an independent structure in hardware.

Functional Processor Model: An abstraction that models a processor as a number of sequentially linked subsystems.

Global (GAg) Predictor: A branch predictor that utilizes a global history register indexing into a pattern history table.

Gselect Predictor: Functionally synonymous to the GAg Prediction scheme.

Gshare Predictor: A predictor that merges (XOR) the global history register with the current PC to form an index into a pattern history table.

Inactive Issue: A trace cache technique that effectively implements predication of traces.

Instruction Dispatch: In the context of shelving, when an instruction is disseminated to a functional unit for execution from a shelving buffer.

Instruction Fetch Queue (IFQ): Structure that separates the fetch stage with the decode/issue stage where fetched instructions are buffered.

Instruction Issue: In the context of shelving, the point at which an instruction is decoded and sent to a shelving buffer.

Instruction Predecoding: A method that associates additional bits that provides extra information (such as instruction class) to each I-Cache entry. This allows for early decisions to be made within the pipeline.

Intrablock Branches: In the context of the collapsing buffer, branches that are predicted taken with a target address within the same cache line.

Issue Policy: The collective techniques that govern the manner in which instructions are issued.

Issue Rate: How many instructions are capable of being issued per cycle.

Level of Speculativeness: The maximum number of speculative branches that may exist in the machine (in-flight) at any time.

Local (PAg) Predictor: A branch predictor that utilizes a 2-level table structure (per-address and pattern history table).

***m*-constrained:** Implies that the maximum number of basic blocks (*m*) has been reached for a particular trace.

Merged BTB/BBT: The combined BTB and BBT structure. Both taken and not-taken branch targets & bias values are recorded.

MGAg Predictor: A multiple branch prediction extension of the GAg Predictor. Forms m predictions by iteratively accessing the pattern history table.

n -constrained: Implies that the maximum number of instructions (n) has been reached for a particular trace.

Partial Matching: A trace cache technique that allows a hit to occur if only a few branch flags match the prediction.

Path-Based Next Trace Prediction: A prediction scheme coupled with the trace cache that replaces traditional multiple branch prediction.

Pattern History Table (PHT): An element of branch predictors that stores entries consisting of previous branches outcomes as saturating counters.

Per-Address Table (PAT): An element of a per-address (i.e. PAg) prediction scheme that is indexed by the PC value to produce an index into the PHT.

Probable Entry Point: A fetch address that starts a trace that is likely to be encountered again as a PC value over the course of execution.

Read-After-Write (RAW) Hazard: Condition occurring when one instruction reads from a location that was the write target of a previous instruction before the value is available.

Reassociation: An instruction pre-processing technique that optimizes instructions with immediate operands that follow each other sequentially or are separated by a branch.

Register Renaming: A technique that eliminates false data dependencies within a dynamic scheduling mechanism.

Register Update Unit (RUU): A mechanism that combines the techniques of dynamic scheduling (shelving, renaming and reordering) into a single hardware structure.

Rename Buffers: An allocated entry for an instruction that has been renamed. Dependent instructions will internally refer to this entry rather than the architectural registers.

Reorder Buffer: A common mechanism that provides in-order commit for machines that utilize out-of-order issue/execute.

Return Address Stack (RAS): A mechanism that pushes subroutine CALLs to a stack, a pops RETURNs off. Serves as a Branch Address Prediction method.

Selective Trace Storage: A technique that eliminates redundancy between the I-Cache and the trace cache.

Shelving Buffers: A generic structure that holds instructions as part of a shelving scheme as they await dependency resolution.

Shelving: A generic term for the mechanism that decouples instruction issue from dependency resolution.

Sliding Window Fill Mechanism (SWFM): A fill unit scheme that exploits trace continuity and probable entry points. It is paired with the Fill Select table.

Speculative Execution: Proceeding to fetch, issue and execute instruction along an assumed path prior to evaluation of a branch.

Static Program Order: The layout of a program in I-Cache cache as structured by a compiler.

Superscalar: The ability of a processor to issue more than a single instruction per cycle.

Trace Continuity: A property belonging to a sequence of traces that can be fetched back-to-back over multiple fetch cycles.

Trace ID: The value that uniquely identifies a trace segment. Often this is the address of the first instruction in the trace.

Trace Packing: The idea of filling trace cache lines irrespective of basic block boundaries.

Trace Preconstruction: A method that generates trace segments for a region before execution reaches the location by utilizing the idle fetch stage resources.

Trace Table: In the context of the Block Cache, the structure that replaces the traditional trace cache by storing references to blocks rather than explicit instructions.

Trace Target Address: The address corresponding to a taken result for a trace-terminating branch.

Trace Terminating Condition: A condition that must exist to finalize a trace in the fill buffer and force a segment to be written.

Trace: A finite “snapshot” of the dynamic execution order of a program.

Uniform Trace Cache Model: Flow diagram from which the trace cache simulator extension was designed. Represents a modularization of the specific tasks of the trace cache.

Write-After-Read (WAR) Hazard: Condition occurring when one instruction writes to a location before a previous instruction reads from the same location.

Write-After-Write (WAW) Hazard: Condition occurring when one instruction writes a location before a previous instruction writes to the same location resulting an inconsistent state for other instructions in between.

Acknowledgments & Trademarks

The author wishes to acknowledge Dr. Muhammad Shaaban, Dr. Roy Czernikowski and Dr. Ken Hsu for serving as advisors to this work. Thanks is given to Brad Virkler and Eric Majewicz for providing some of the scientific application kernels utilized as benchmarks.

“Pentium 4” is a trademark of the Intel Corporation.

Chapter 1

Introduction

Trace cache has emerged as a promising mechanism to achieve high-bandwidth, low-latency instruction fetch of multiple, non-contiguous basic blocks. This chapter states the motivation driving trace cache research and outlines the organization of subsequent chapters.

1.1 Motivation for Trace Cache

For superscalar processors, an increase in issue width results in a significant performance gain if the majority of available issue slots are utilized every cycle. If this is not the case, the under-utilization of issue slots is characterized as horizontal waste. Several hardware methods that exploit instruction level parallelism have been proposed to minimize this waste, including out-of-order issue and speculative execution. These ILP related optimizations effectively boost instruction execution bandwidth. To sustain an increase in execution bandwidth proportional to issue width, the schedulable pool of instructions must remain robust. This implies that the instruction fetch bandwidth must be at least equal to the number of instructions issued per cycle. Traditional superscalar fetch mechanisms can provide a set of instructions up to, and including, a single branch instruction. Furthermore, it has been stated that the average basic block size is five instructions for general-purpose code [1]. As superscalar widths extend past the current four-way implementations of today, it is evident that the instruction fetch bandwidth has the potential to be a major bottleneck within the pipeline. Several mechanisms have been proposed to fetch past basic block boundaries, including the branch address cache and collapsing buffer. The most promising though has been the trace cache.

In addition to providing a solution to increasing fetch bandwidth, trace cache allows for dynamic optimizations in the stream of instructions that are delivered down the pipeline. This has the advantage of reducing execution and scheduling overheads by eliminating redundant operations and minimizing instruction dependencies.

Incorporating trace cache can the effect of substantially increasing the average number of instructions per cycle that a superscalar machine can complete, which in turn decreases the execution time required for running a particular application – the basic goal of any architecture related enhancement.

1.2 Organization of Thesis

This thesis will identify some various trace cache and instruction pre-processing schemes that were examined using an extension to an existing microarchitectural simulator.

- First, an overview of the modern superscalar processor is presented as background theory in Chapter 2. Specifically, design space topics for instruction issue, the core fetch unit and alternate fetch mechanisms are examined. The SimpleScalar toolset, used as the basis for simulator extension, is described.
- Chapter 3 is devoted to presenting some trace cache schemes currently proposed and implemented.
- The object-oriented design of the trace cache simulator extension is explained in Chapter 4.
- Chapter 5 entails discussion of simulation results for various schemes and parameter variations. The Sliding Window Fill Mechanism is introduced as a new scheme to improve trace cache hit rates.
- Conclusions are made in Chapter 6, along with difficulties encountered and recommendations for future work.

Chapter 2

Background

This chapter sets the stage for discussion of trace cache in the context of the modern superscalar processor. Topics such as instruction issue, the core fetch unit and alternate fetch mechanisms are examined. The SimpleScalar simulator is presented and classified using the design space established in the following sections.

2.1 Emergence of the Superscalar Processor

Over the past few decades, the field of processor design has been marked by continuous evolution and improvements. Prior to the 1980s, performance accelerated at an approximate rate of 1.35 times per year. This growth was driven largely by technology improvements, including the integrated circuit and the microprocessor. Since then, architectural advancements along with steady gains in process technology have lead to performance rate increases of over 1.5 times per year [2]. The adoption of RISC instruction set architectures (or RISC processor cores), pipelining, and dynamic scheduling have all been important milestones in the development of microarchitectures. One of the most substantial gains since the early 1990s has been multiple, or **superscalar**, instruction issue. The simple motivation to utilizing superscalar issue is to process instructions in parallel, thereby increasing the number of instructions executed per cycle (IPC) and decreasing the execution time of a given program.

The fundamental concept that allows for concurrent execution is instruction level parallelism. Unfortunately, the level of parallelism able to be extracted from a set of instruction is limited by a number of factors, or dependencies. Dependency

relationships can be classified into three groups: data dependencies, name dependencies, and control dependencies [2].

Data, or true, dependencies between instructions are the most straightforward. If the result of instruction i is required as a source operand to a subsequent instruction j , then j is data dependent on i and must wait for its result before executing. This corresponds to a **read-after-write (RAW)** hazard within the processor pipeline.

Name dependencies (false dependencies) can be further separated into anti-dependencies and output dependencies. An instruction j is anti-dependent on a preceding instruction i if j writes to a location that is a source operand of i . Similarly, output dependence exists if both instructions i and j write to the same location. These name dependencies correspond to **write-after-read (WAR)** and **write-after-write (WAW)** hazards respectively.

The final type of dependency that hinders the extraction of instruction level parallelism is control dependence. An instruction j is dependent on control instruction i if it cannot be moved before instruction i such that it is no longer controlled by the outcome of the branch. Likewise, instruction i is dependent on control instruction j if it can not be moved after instruction j such that it is controlled by the outcome of the branch.

While not exclusive to superscalar design, the effect of hazards on performance become most prevalent as issue rates exceed a single instruction per cycle. To circumvent the restrictions to parallel execution that these dependencies cause, mechanisms must be incorporated in a superscalar processor. While many microarchitectures are based on the simplicity of RISC ISAs, the overhead required to minimize the effect of dependencies add a fair amount of design complexity. In further elaborating these mechanisms, is convenient to establish a functional processor model. This abstraction consists of a number of sequentially linked subsystems: Fetch, Decode/Issue, Dispatch, Execute and

Reorder/Commit. The link between the fetch and issue subsystems is the **instruction fetch queue (IFQ)**. Within the decode/issue subsystem is the **decoded instruction queue (DIQ)**. The remaining subsystems are linked implicitly, dependent on the scheme. To avoid obvious bottlenecks, the bandwidth (rate) of each stage should be roughly equal [1]. The next section provides an overview of the latter 4 subsystems. The fetch unit will be treated to more rigorous detail in Section 2.3, as it provides the basis for later discussions on Trace Cache.

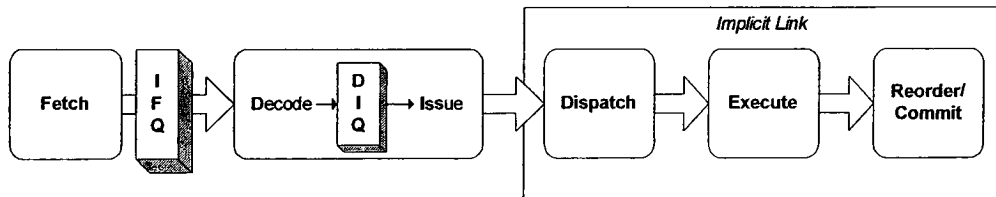


Figure 2.1: Functional Processor Model

2.2 Tasks of Superscalar Processing

The decode/issue, dispatch, execute and reorder/commit subsystems collectively are concerned with performing the tasks of superscalar processing, which can be subdivided as follows [1]:

- Parallel Decoding
- Superscalar Instruction Issue
- Parallel Execution
- Instruction Reordering
- Exception Processing

The first task is to decode the set of instructions that has been passed down the pipeline from the fetch unit. In the traditional scalar processor, dependency checking is combined with instruction decoding to ensure that all source operands are available before an instruction is issued. A superscalar processor treats these two instruction-processing tasks independently, as they may be

decoupled within the pipeline. In addition to requiring additional resources to decode multiple instructions simultaneously, interdependencies among the set of instructions being decoded must be flagged for consideration during instruction issue.

Two aspects comprise the task of superscalar instruction: **Issue Rate** and **Issue Policy**. The issue rate, or issue width, of a processor indicates how many instructions *are capable* of being processed from the DIQ per cycle. One should note that this value is distinct from **effective issue bandwidth**: how many instructions on average *are* being issued per cycle. The issue policy aspect encompasses a sizeable design space, outlined in Figure 2.2. At this point it is appropriate to make a distinction between the terms **issue** and **dispatch**. An instruction is issued after it has been decoded. Dispatch occurs when an instruction, along with the appropriate operand values, is disseminated to a particular functional unit for execution. In a simple pipeline issue and dispatch are one in the same: An instruction either proceeds to execute or is blocked after being decoded. The issue/dispatch distinction is required in the superscalar case for reasons that will soon be apparent.

Parallel execution is a matter of hardware resources. Multiple execution units (EUs) are required to maintain a bandwidth equal to that of the rest of the system. Aspects of functional units such as multiple-cycle latencies and pipelined versus non-pipelined operation are considerations that are inherent in the other subsystems and need not be addressed in the context of parallel execution.

Instruction reordering is a task concerned with maintaining the sequential consistency of instruction execution. Advanced issue policies necessitate the explicit control of instruction retirement such that the architectural state of the processor remains valid. The **reorder buffer (ROB)** is a common mechanism that accomplishes instruction reordering. Often implemented as a circular buffer, each issued instruction is appended in-order at the tail of the ROB. The reorder buffer is related to aspects of the issue policy and is linked with rename buffer

deallocation, mentioned in the next section. Modern machines utilize reordering to enforce a strong processor consistency model and precise exception handling [1].

2.3 Issue Policies

There are four main aspects to classifying the issue policy of a superscalar processor [1]. First is whether speculative execution is utilized, a topic that will be addressed in Section 2.4. Second is how issue blockages are handled. Regardless of how the rest of the issue policy is implemented, blockages can occur via data and/or structural hazards. Four possibilities for this scheme exist: fixed window/in-order, fixed-window/out-of-order, sliding window/in-order, and sliding window/out-of-order. The issue window affects how instructions in the DIQ are processed each cycle. A fixed window scheme will treat a set of instructions as a unit, proceeding to the next set only when every instruction in the current set has issued. A sliding window considers for issue the n -most instructions at the head of the queue, effectively re-adjusting itself every cycle. If an in-order issue scheme is employed, an instruction at the head of the DIQ that is blocked will inhibit any other instruction from issuing. An out-of-order scheme will allow instructions beyond a blocked instruction (within the window) to issue if possible. A graphic description of the four possible window and in/out-of-order combinations is shown in Figure 2.3.

The third aspect of a superscalar issue policy is how data dependency checking is handled. The blocking issue approach (i.e. wait until all dependencies are resolved) is standard among scalar and first-generation superscalar machines. This method is restrictive in that it is subject to, and limited by, the issue blockage schemes described above.

1. Issue Blockages
 - a. Issue Window
 - i. Fixed
 - ii. Gliding
 - b. Issue Order
 - i. In-order
 - ii. Out-of-order
2. Name dependence Handling
 - a. Register Renaming (or No Register Renaming)
 - i. Scope
 - ii. Layout of Rename Buffers
 1. Types
 - a. Merged Architectural and Rename Register File
 - b. Separate Rename Register File
 - c. Reorder Buffer
 - d. DRIS/RUU
 2. Number of reorder buffers
 3. Accessing reorder buffers
 - a. Associative values
 - b. Mapping Table
 - iii. Operand fetch policy
 1. Issue Bound
 2. Dispatch Bound
 - iv. Rename Rate
3. Unresolved control dependency Handling
 - a. Speculative Execution (or No Speculative Execution)
 - i. *Design Space of Control Transfer Instructions*
4. Dependency Checking
 - a. Decouple instruction issue and dependency check – Shelving (or Blocking Issue/No Shelving)
 - i. Scope
 - ii. Layout of shelving buffers
 1. Reservation Stations
 2. Group Station
 3. Central Station
 4. DRIS/RUU
 - iii. Number of buffer entries
 - iv. Number of input/output ports
 - v. Operand Fetch Policy
 1. Issue Bound
 2. Dispatch Bound
 - vi. Dispatch Scheme
 1. Dispatch Policy
 - a. Dispatch Order
 - i. In-order
 - ii. Out-of-order
 - b. Selection Rule (applicable for out-of-order dispatch)
 - c. Arbitration Rule (applicable for out-of-order dispatch)
 2. Dispatch Rate
 3. Operand Availability Checking
 - a. Direct Check of Scoreboarding Bits
 - b. Checking of explicit Status Bits
 4. Empty Shelving Buffer Treatment
 - a. Bypassing
 - b. No bypassing

Figure 2.2: Issue Policy Design Space [1]

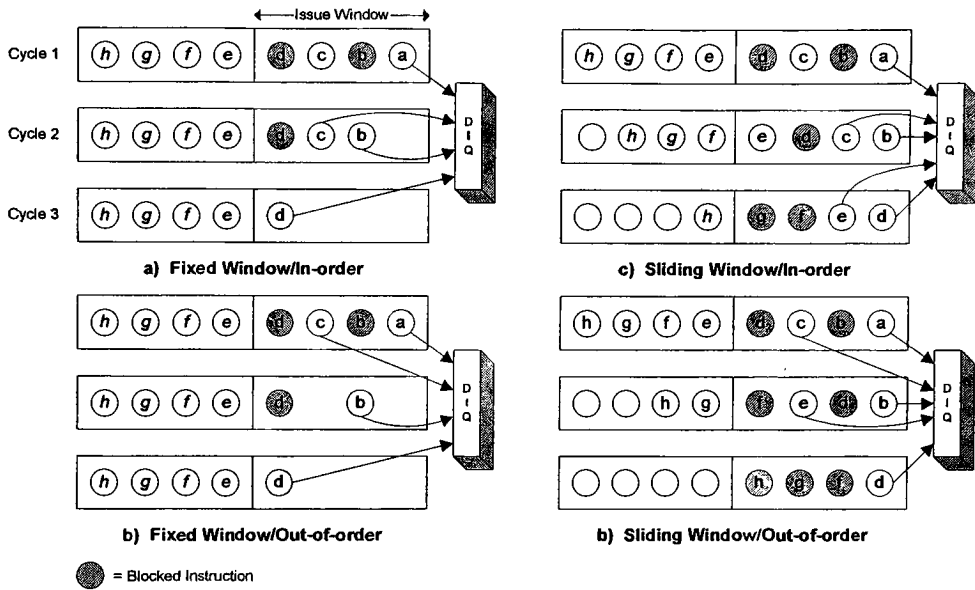


Figure 2.3: Methods for Handling Issue Blockages

A more attractive solution is one that decouples instruction decode from issue, deferring most dependency resolution to the later stage. Generically termed **shelving**, the mechanism to achieve this dependency resolution has various layouts and design options itself. **Shelving buffers** consist of one or more entries to which instructions are issued. Only until all dependencies are resolved may an instruction be dispatched from the shelving buffer to an appropriate EU. The number of buffer entries, and the layout are implementation specific. If a separate set of shelving buffers precedes each EU, the buffers are commonly referred to as reservations stations. Buffers can also service multiple EUs of the same type (group stations) or a single buffer may exist for all functional units (central station). When instructions are issued to a buffer, the available source operands can be retrieved from the register file and stored in the shelving buffer entry. Likewise, operands may be fetched upon dispatch, leaving the shelving buffer entry to store only the register identifier. Which operand fetch policy is used often correlates to how operand availability checking is handled. All

shelving schemes utilize a structure called a scoreboard. This can be visualized as a single bit extension to each register file entry indicating the availability of that value. When an instruction is issued (or in some cases dispatched) the scoreboard flag for the destination register is marked to indicate the value is not available. An instruction that has finished execution will reset the appropriate scoreboard bit. Subsequent instructions that are issued check the scoreboard bits for each of their source operands. A dispatch bound fetch policy entails a direct check of the scoreboard bits every cycle to determine if source operands are available. An issue bound operand fetch policy includes explicit availability flags for each source operand in the rename buffer entries. Upon issue, these explicit status bits in a newly allocated rename buffer entry are set to correspond with the value of the respective scoreboard bits. If an operand value is not available, the operand field in the rename buffer entry is set to the register identifier. Using a common data bus scheme, instructions that have finished execution provide the result value to not only the register file and scoreboard, but to each shelving buffer. If the identifier of the result matches that of any source operand that has been marked unavailable, the buffer entry is updated with the operand value and the explicit status bit is modified to reflect the availability.

Once both source operands are available, and no WAR or WAW hazards are detected for the instruction (a step eliminated by the soon-to-be-discussed process of register renaming), it is ready to be dispatched to an available EU. The dispatch policy determines how instructions are selected once they are ready. This also includes whether or not multiple instructions may be dispatched from the same shelving buffer in a single cycle. Similar to the issue rate, the dispatch rate determines how many possible instructions that are ready for dispatch may be disseminated per cycle. It has been found that it is optimal for this rate to slightly exceed the issue rate to account for multi-cycle latencies associated with non-pipelined EUs. One interesting optimization that can be made is bypassing empty shelving buffers. Normally, if an instruction is issued to an empty

reservation station and is free of dependencies, it will be recognized as a candidate for dispatch in the following cycle and sent to an EU. Bypassing saves a cycle by forwarding the instruction to the EU directly, performing issue and dispatch in the same cycle [1].

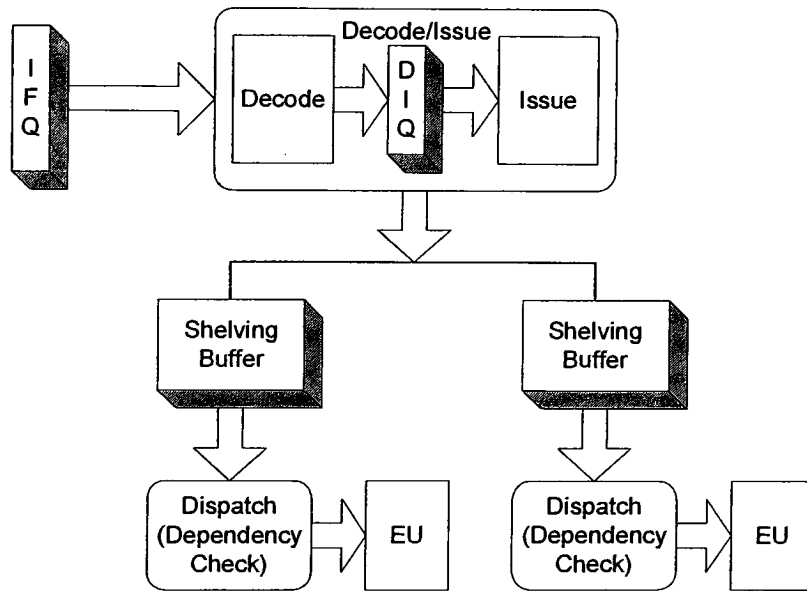


Figure 2.4: Shelving

Shelving alleviates the problem of RAW hazards from impacting the performance of superscalar machines, but still leaves WAR and WAW hazards to contend with. The last aspect of the superscalar issue policy addresses these hazards. Incorporating a technique known as **register renaming** can internally eliminate false dependencies. Like shelving, one can avoid addressing the layout specificity of a variety of register renaming schemes by generically referring to **rename buffers**. When an instruction is issued, a rename table is accessed. This associates architectural register identifiers to entries in the rename buffer. For some implementations, a single register identifier may have multiple table associations, in which case a “latest” bit will be set for one of these entries.

Accessing the rename table occurs in two steps. First, both of the source register identifiers are used to index the rename table. For schemes that allow one-to-many mappings, the entry with the “latest” bit is accessed. If an entry exists, the associated table value (rename buffer entry ID) replaces the source identifiers of the instruction. Secondly, a new location in the rename buffer is provided for the destination operand, and is associated with the destination register identifier via a freshly allocated entry in the rename table. This will overwrite any prior entry with the same destination register as the index, or set the “latest” bit for the new entry. The values provided by the rename table will become those used throughout the dispatch and execute stages for the instruction. During execution, each instruction will refer to a unique destination location; therefore checking for WAR and WAW hazards becomes irrelevant. This renaming prevents instructions from being held from dispatch because of output and anti-dependencies. Deallocation of rename buffer entries is often tightly coupled with the mechanism that preserves the sequential consistency of instruction execution. In the case of the reorder buffer, renamed registers are committed to architectural state when an instruction reaches the head of the ROB. The associated rename buffer entry is copied to the explicit architectural register and the rename buffer and table entry deallocated, or the rename register is mapped as the current architectural register, deallocating the previously mapped rename buffer entry [1,3].

For the purpose of explanation, treating shelving, register renaming and reordering as separate processes with independent resources is convenient. Though in many implementations, these mechanisms are tightly coupled (as has already been alluded in describing the process of rename buffer deallocation). Tomasulo’s algorithm, a popular example of dynamic scheduling, is an example of how both shelving and renaming are often presented and implemented together. The rename buffers of the Tomasulo approach are implicit in the

operand value fields of the reservation stations. Figure 2.5 outlines Tomasulo's algorithm from a design space perspective.

The combination of all three mechanisms (shelving, renaming and reordering) is realized in approaches such as the as DRIS [1], or RUU [4]. In the case of the RUU (Register Update Unit) a circular buffer that resembles a reorder buffer also serves the role of central reservation station with implicit rename registers and associative rename table (tag unit). While the organization of such an implementation may seem confusing, it should not obscure the purpose behind the underlying mechanisms.

- Scalar decoding/Issue Rate of 1
- In-order issue
- Speculative Execution
- Register Renaming
 - Rename registers integrated as operand value fields in reservation station entries.
 - Mapping Table Access – Each register is associated with a tag value (one-to-one).
 - Issue Bound Operand Fetch Policy
- Shelving
 - Multiple entry shelving buffers (reservation stations) for each EU
 - Issue Bound Operand Fetch Policy
 - In-order dispatch
 - Explicit status bits for operand availability checking
 - No shelving buffer bypassing

Figure 2.5: Design Space Outline of Tomasulo's Algorithm

2.4 Processing of Control Transfer Instructions

The previous section showed how advanced superscalar processors address the limitations of true data dependencies and false name dependencies. The issue of control dependencies will now be discussed as a prerequisite to describing the

fetch subsystem and trace cache. **Control transfer instructions** (often generally termed branch instructions) consist of five basic types: direct jumps, indirect jumps, subroutine calls, subroutine returns, and conditional branches. These should all be familiar to the reader in the context of assembly level code. From the perspective of the ISA, conditional branch instructions can be implemented in two distinct ways. Result state branches depend on the outcome of a previous instruction to set a condition code register that is implicitly used to determine the direction of a branch. Direct check branches provide the operands to compare and the branch target as a single self-contained instruction. [1]

The question often associated with control transfer instructions is: How can a seemingly straightforward set of instructions pose a problem in superscalar processing? When executing a sequence of instructions, the PC of the subsequent instruction following a branch operation is dependent on the evaluation of that branch. In a scalar pipelined architecture, this implies that a branch direction must be indicated to the fetch stage by the following cycle to avoid stalling the pipeline. For a superscalar architecture, this branch direction is needed in the same cycle to be able to continue to fetch beyond a given control transfer instruction in a fetch block. As architectures become increasingly superscalar and superpipelined, the limitations due to control dependencies become increasingly pronounced. Fortunately mechanisms have been introduced to deal with the “branch problem” just as they have for the other classes of instruction dependencies.

Branch processing can be considered in terms of branch detection, treatment of unresolved conditional branches, and accessing the branch target path. [1] The first of the processing steps, branch detection, can be handled at various points in the pipeline. If no special changes are made, detection occurs as each branch instruction is decoded. This also means that instructions continue to be fetched along a sequential path until the branch is decoded in a later stage. The obvious disadvantage to this scheme is the instructions that are “blindly fetched” after the

branch may not be along the correct control path. The sooner a control transfer instruction can be detected, the better. To this end, branch detection integrated with the fetch stage is optimal.

Once a branch is detected, it must be evaluated. This especially holds true for conditional branches, indirect jumps and subroutine returns. Unlike branch detection, this step cannot be arranged such that zero latency exists between branch evaluation and fetching of the next instruction. This presents two options: The first, and decisively less attractive option is to stall until the branch in question is evaluated (blocking branch processing). The second is to continue executing instructions as if the branch direction was known. Proceeding to fetch, issue and execute instructions along an assumed path prior to evaluation of a conditional branch is known as **speculative execution**. If a speculatively executed path proves to be correct after the branch is evaluated, those instructions are promoted to a non-speculative state and can be safely committed (thereby changing the architectural state of the system). The control instruction has been correctly predicted in this situation. If evaluation of the control instruction yields a different target than that starting the speculative path, then those instructions along the speculative path must be canceled and fetching must resume along the correct path. This constitutes a misprediction. Some quantitative measures of speculative execution are the **level of speculativeness** and the **degree of speculativeness**. The maximum number of branches that may be executed speculatively at any given time (the number of unresolved branch instructions “in-flight”) is the level of speculativeness, while the degree is how far down the pipeline speculatively executed instructions are allowed to proceed. [1]

The key factor of speculative execution is utilization of a branch prediction scheme that will yield high accuracy and a low misprediction percentage. Branch prediction can be fixed, static or dynamic. A fixed prediction scheme will always

choose a taken or non-taken path regardless of the branch. A static scheme involves encoding the branch prediction within the instruction. This prediction is generated by the compiler based on inferred branch behavior. Dynamic branch prediction is the most accurate, and most complex of the three forms of prediction. Hardware is utilized to retain state information pertinent to control instructions. This may be, but is not limited to, individual branch histories, the direction of the last n branches or a combination of both. Various methods for performing dynamic branch prediction exist and only a few are presented here.

In early studies of branch prediction, Smith proposed the simple bimodal predictor that maintains a history of specific branches as a two-bit table entry [1]. Single level prediction, of which bimodal is an example, has the limitation of considering each branch without correlation to a specific patterned behavior. Yeh and Patt addressed this in proposing a two level predictor. Such a predictor uses the lower bits of the branch address to index a table of n history bits. In turn, these history bits index a bimodal table of length 2^n . The branch prediction is based upon the entry in the bimodal table. Once a branch has been evaluated, the history bits of the first table are updated by shifting the current result into the table. The bimodal table is then updated based upon this new history index. This specific strategy is termed a **local predictor**, or in a later paper presenting a taxonomy of two level predictors, a **PAG** scheme [5]. A visual representation is provided in Figure 2.6.

The history pattern is indexed solely by the current branch address in the **PAG** predictor. Another predictor considers instead the history of the most recent branches collectively. This method is aptly referred to as the **global predictor** or **GAG** predictor. The history of the most recent branches is recorded in a single global history shift register. This has the advantage of choosing a prediction based upon the association of many independent branches [5].

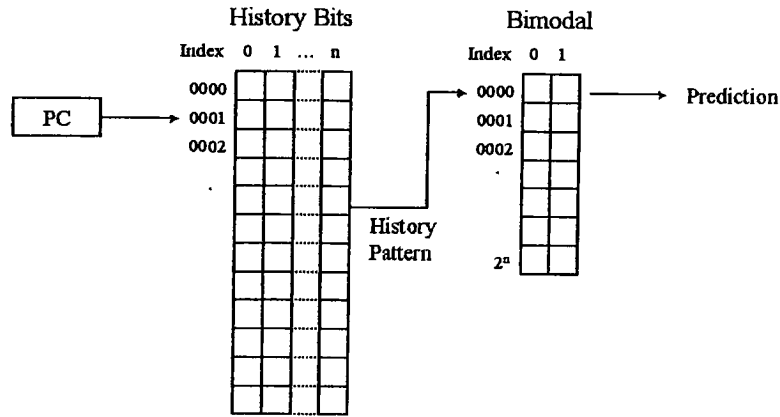


Figure 2.6: Local (PAg) Predictor

The **gselect predictor**, proposed by Pan, So and Rahmeh realizes such a combination by appending the global history to the y lowest bits of the PC. Another method of combining prediction, **gshare**, uses the XOR function to merge the global history register with the current PC [7], shown in Figure 2.7.

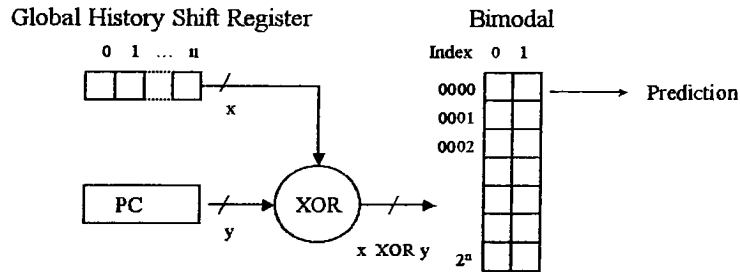


Figure 2.7: Gshare Predictor

The final method of dynamic branch prediction discussed here is the **combined predictor**. Proposed by McFarling, this technique allows different branch predictors to be used in tandem, and the best result to be chosen based upon past performance of the specific branch being predicted. A generalized scheme is shown in Figure 2.8. Accuracy counts for each of the individual predictors are recorded separately. These separate fields can be appended to the branch target buffer. Depending on the correctness of predictor n , the respective

accuracy count is incremented or decremented accordingly. A priority encoder chooses the scheme that has been most accurate and provides this output as the final prediction. McFarling has noted that using combining branch predictors in this manner provides prediction accuracy up to 98.1 percent [7].

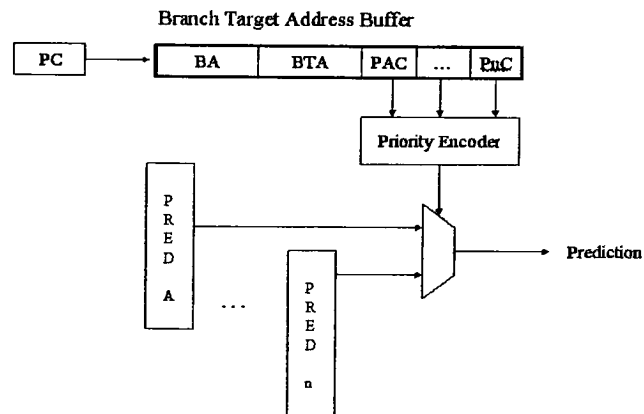


Figure 2.8: General Combined Predictor

Closely associated with branch prediction is the task of accessing the branch target as early as possible when processing a control transfer instruction. Predicting the direction of a branch alone is still only half of the information that is required to fetch instructions along a speculative path. In the case of predicted taken conditional branches and direct jumps, the fetch address is known only after the instruction is decoded. For indirect jumps and subroutine returns, this fetch address is often evaluated even later in the pipeline. Again, mechanisms have been introduced to provide a **branch address prediction** (versus a **branch direction prediction** as explained above). The **branch target buffer** (BTB) or more precisely, the **branch target address buffer** (BTAB) is an example [1]. The PC of each instruction being fetched serves as an index into a table. If the entry is valid and there is a tag match, it serves as an indication that the instruction is a branch, and the target address is provided. Such a scheme can serve as a simple implicit branch direction predictor as well. Implicit branch direction prediction

methods rely on the presence (or lack of) a table entry to indicate a taken (or not-taken) prediction.

For indirect jumps and subroutine returns the issue of branch address prediction is complicated by the fact that branch target itself relies on the dynamic state of a program. Branch address prediction for subroutine calls will be one of the topics addressed in the next section.

2.5 Elements of the Instruction Fetch Subsystem

Trace cache aims to improve the bandwidth of instructions delivered from the instruction fetch stage in wide issue superscalar processors [10]. This section will provide background on the traditional instruction fetch stage for superscalar machines, and why it proves to be inadequate for increasingly wider issue architectures.

The instruction fetch subsystem of a superscalar processor has the following core components:

- Program counter
- L1 predecoded instruction cache
- Branch target buffer
- Return address stack
- Branch direction predictor
- Branch target logic

Among these six, the program counter is simplest. Unchanged from the most basic architecture, this register holds the next fetch address from which the instruction cache is accessed.

At the top of the memory hierarchy for any processor is the first level cache. For scalar designs, there is no particular advantage to either a separate (Harvard) or unified L1 cache design. Superscalar machines utilizing **instruction predecoding** favor separate instruction and data caches. The I-Cache is

interleaved, i.e. 2 consecutive cache lines can be accessed. This allows retrieval of a set of instructions the size of an entire cache line regardless of where the initial instruction lies in the cache. The necessary interchange, shift and mask logic to realize this is assumed to consume a negligible (or at least acceptable) delay time within a single cycle [10]. Predecoding entails the incorporation of a predecode unit between the second-level cache and the L1 I-Cache. As lines are written to the I-Cache a number of bits (between four and seven based on recent implementations) are appended for each instruction. These bits can be used to indicate instruction class, resource requirements, and even possibly the branch target address [1]. For this thesis, predecode bits are assumed to provide a distinction between the various control transfer instructions.

The branch target buffer plays an important role in identifying branch targets early in the pipeline, as described in the previous section. The organization of this structure can vary in associativity depending on implementation, though is often two or four-way set associative. Conditional branches and direct jumps are stored in the BTB. Alongside the BTB, the **Return Address Stack (RAS)** provides a mechanism to help predict the target address for subroutine returns. The stack depth is set; usually eight proves to be sufficient in most implementations. For every subroutine call (jump-and-link) instruction encountered, the address of the subsequent sequential instruction is pushed onto the stack. When a subroutine return is encountered, the topmost address is popped and used as the branch address prediction. The RAS increases the accuracy of subroutine returns considerably, though there are situations when the RAS mechanism can become corrupted, resulting in address mispredictions. This can occur if the RAS overflows, or when certain sequences of instructions along a mis-speculated path are fetched. While these RAS misses do not affect the correctness of program execution, they impact performance as would a branch direction misprediction.

Branch prediction can be implemented as any of the various schemes discussed in the previous section. For each instruction fetched, the branch predictor is accessed simultaneously with the BTB and RAS. The branch target logic serves as the glue between these units to determine the set of instructions that will be passed down the pipeline and how the PC should be set for the next fetch cycle. First, the predecode bits are examined to determine the control transfer operations within the cache line. The first branch in the sequence will terminate the set passed to the IFQ. If this branch is a jump (direct, indirect, subroutine call) the PC will be set to the corresponding address generated by the BTB. Similarly a subroutine return will utilize the address generated by the RAS. The target address of conditional branches depends on the branch direction prediction. A prediction of taken will set the PC to the BTB generated address, a not-taken prediction will simply set the PC to the address of the next sequential instruction.

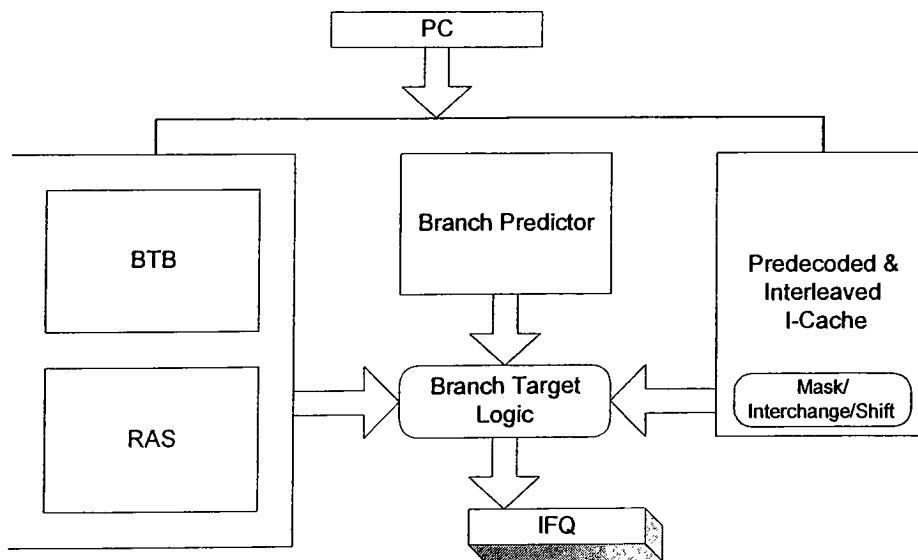


Figure 2.9: Fetch Subsystem Components

How does the core fetch unit described above hinder performance in a wide-issue superscalar machine? For most code, the percentage of branch instructions is between 10% and 20%. This equates to a basic block size of between 5 and 10 instructions [1]. As the issue width of architectures exceed 4 instructions per cycle, it becomes obvious that not only is extracting parallelism beyond a basic block crucial, but instructions must also be *fetched* beyond a single basic block per cycle. In an effort to resolve this problem, a series of revisions and alternatives to the fetch method just described are presented next.

Revision 1 – Fetching Past Predicted Not-Taken Conditional Branches

The first revision to the fetch subsystem is one that partially alleviates the single basic block fetch limit by targeting conditional branches that are predicted not-taken. Since the I-Cache is interleaved, a set of instructions the size of a full cache line can be retrieved every cycle. When a conditional branch is encountered, it is possible that the prediction will be not-taken. If this is the case, there is no cache limitation to fetching instructions beyond the branch and including them with the set of instructions that will be appended to the IFQ. If m conditional branches are to be predicted per cycle, then up to $m + 1$ basic blocks may be fetched in the same cycle if all branches are predicted not-taken. Changes must be made to the BTB and predictor for this modification. Since it is now required that more than one branch target be accessed per cycle, the BTB must be interleaved to allow parallel accesses. Similarly, the branch predictor must now be able to handle multiple predictions per cycle. Just as there are many approaches for single branch prediction, several variations to multiple branch prediction have been proposed. The scheme that will be considered here is an expansion of the GAg single branch prediction method. Called an MGAg predictor, the components remain the same (a global history register of width k and pattern history table of 2^k entries) though multiple predictions are made via

iterative access of the PHT up to m predictions. The access constituting the first iteration is made in the same fashion as the basic GAg predictor. The result becomes the prediction for the first branch. The second access is made using the $k-1$ most recent bits of the GHR. This provides a base index into the PHT. An additional offset (0 or 1) is added to this index depending on the result of the first prediction (NT or T). The prediction for the second branch is then determined from the bimodal entry indexed by this newly computed index. The next iteration uses $k-2$ bits of the GHR to determine a base index, and an offset value of 0, 1, 2 or 3 depending on the combined prediction of the previous two branches (NT/NT, NT/T, T/NT or NT/NT). This continues until m predictions have been made [8]. A graphical representation of the MGAg predictor is shown in Figure 2.10.

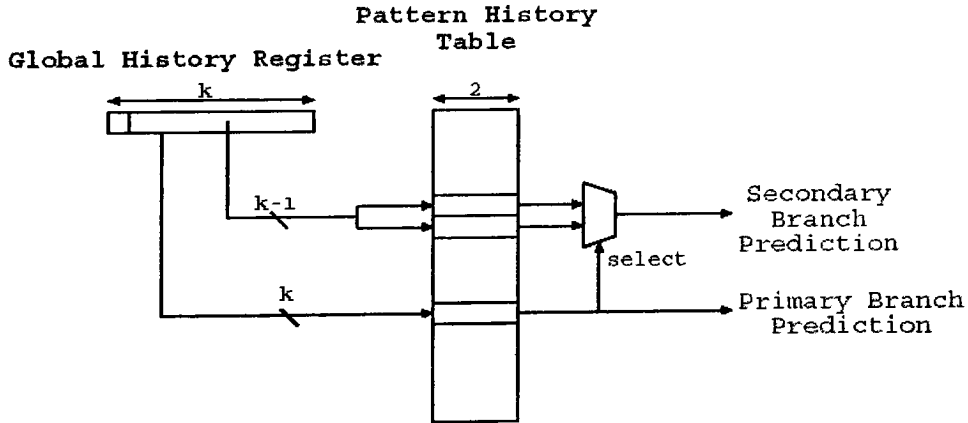


Figure 2.10: MGAg Predictor [8]

While this revision manages to improve the fetch bandwidth over the original, there still exist shortcomings. Statistics indicate that conditional branches consist of 66% of all control transfer instructions, of which only 25% are evaluated not-taken [1]. Therefore, approximately 83% of all branches (those that are taken) still manage to limit the rate of instructions passed down the

pipeline. The next two methods provide different approaches to overcoming the restriction posed by predicted taken conditional branches and jump instructions.

Revision 2a – Collapsing Buffer

The **collapsing buffer** [8] improves the fetch subsystem further by considering **intrablock branches**. These are branches that are predicted taken, but only constitute a short hop within the current pair of cache lines accessed in the interleaved I-Cache. The branch target logic must be expanded to determine such intrablock branches and communicate this to the collapsing buffer following the interchange/mask logic of the I-Cache. The collapsing buffer removes instructions that fall between intrablock branches and their target so that the result is a series of basic blocks that can be appended directly to the IFQ. The collapsing buffer is illustrated below in Figure 2.11.

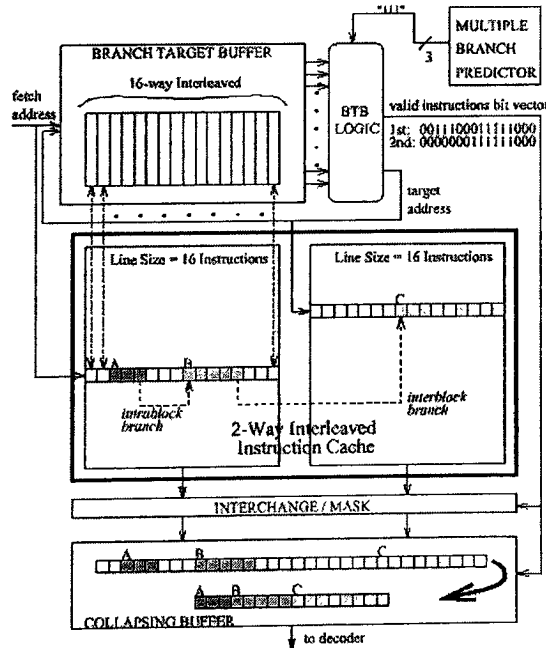


Figure 2.11: Collapsing Buffer [8]

Revision 2b – Branch Address Cache

Another approach that enables multiple basic blocks to be fetched per cycle is the **branch address cache** approach [9]. Required for this scheme is a m -way interleaved I-Cache (versus the two-way version utilized thus far). Presented as a two-stage implementation, the branch address cache is first accessed with the PC address. The BAC acts as an expanded BTB in that it returns information pertaining to the branch targets. But unlike the BTB that returns a single target address for a particular branch, the BAC returns a series of branch targets, arranged as a tree that provides all possible taken/not-taken targets for m branches. Using m results from the multiple branch predictor, the appropriate branch targets are chosen. The second stage retrieves the blocks from the interleaved I-cache, then interchanges/aligns the group to pass to the IFQ. The BAC approach is illustrated in Figure 2.12.

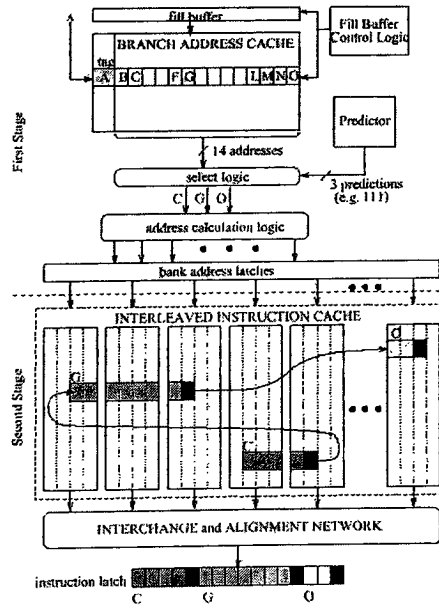


Figure 2.12: Branch Address Cache [9]

Both schemes presented in the second revision of the fetch subsystem have drawbacks as well. The collapsing buffer and branch address cache are fairly

complex, and would require multiple cycles in a hardware implementation. This adds to the latency required for instruction fetch. Cache bank conflicts may also occur when the cache line holding the initial fetch address resides in the same bank as successive lines. Scalability is an issue for the BAC (each entry requires 2^m branch targets) while the collapsing buffer can only fetch across a pair of cache lines. Trace cache schemes resolve these problems, providing the fetch subsystem with a instruction supply mechanism that provides both low latency and high bandwidth. Chapter 3 provides a thorough discussion of trace cache and associated instruction pre-processing optimizations. The next section introduces the SimpleScalar simulator in the context of the functional processor model.

2.6 The SimpleScalar Simulator Architecture

The simulation work for this thesis was based on the third version of the SimpleScalar tool set. SimpleScalar provides a fairly comprehensive functional model for investigating various aspects of a microarchitecture. Included with the standard distribution are a handful of simulators. The simulator of interest to this study was *simoutorder*, a detailed superscalar model that supports out-of-order issue and execution [11, 12]. To place the theoretical background of this chapter in perspective, this simulator will be described in context of the functional processor model.

The native target ISA of the SuperScalar tools is PISA, which is largely based on MIPS-IV with a few exceptions (no branch delay slot, expanded load/store instructions, a square root instruction and 64-bit instructions). Included with the simulator is a compiler, linker and other support tools that target the simulator instruction set. The out-of-order simulator is based on flexible parameters for decode, dispatch and commit widths. The size of the IFQ is also adjustable. Instructions are consumed from the IFQ in-order, with a sliding window the size

of the decode width. The issue mechanisms is based on the register update unit (RUU) [4]. This approach combines the discrete tasks of shelving, register renaming and reordering into a single structure that is realizable in hardware. One will recall that the RUU is arranged as a circular buffer. The value for the number of RUU entries is variable, and is set as a simulation parameter. The functional units that allow parallel execution are divided into 5 classes, integer and floating point ALUs and multipliers and a load store unit. The number of arithmetic EUs and number of load/store ports unit are all adjustable. By design, the RUU enforces strong sequential instruction consistency and precise interrupt handling.

The *simoutorder* fetch unit encompasses all aspects of the fetch subsystem illustrated in Figure 2.9 (Section 2.5). The BTB can be configured with different size and associativity values. Likewise, the number RAS of entries can be set. Predecode bits are implied, and are utilized by the RAS when identifying subroutine calls and returns. Several schemes are available for branch prediction. The method utilized in generating base results (for comparison against results utilizing the trace cache extension) is a two-level local predictor (PAg). The level of speculative processing supported by *simoutorder* is unbounded (no limit to the number of unresolved branches in-flight) and the degree of speculation extends through instruction execution. Figure 2.13 outlines the simulator architecture in the design space of the functional processor model.

- Superscalar issue/decode. Issue rate set via parameter.
- In-order, sliding window consumption from IFQ
- Speculative Execution
 - Unbounded level of speculativeness
 - Degree of speculativeness through Instruction Execute
- Register Renaming
 - RUU layout
 - Rename registers integrated as operand value fields in RUU entries.
 - Associative rename table (tag unit).
 - Issue Bound Operand Fetch Policy
- Shelving
 - RUU layout (serves as central reservation station)
 - Issue Bound Operand Fetch Policy
 - Out-of-order dispatch, loads/stores take precedence, oldest-first selection otherwise
 - Explicit status bits for operand availability checking
 - No shelving buffer bypassing

Figure 2.13: Design Space Outline of *sim-outorder*

Chapter 3

Trace Cache Schemes

Building off the background of Chapter Two, this chapter introduces the trace cache by presenting various schemes that have been proposed. Dynamic instruction pre-processing optimizations are also explored. The sole trace cache implementation, the Pentium 4's Execution Trace Cache, is mentioned at the conclusion of the chapter.

3.1 Concept of a Dynamic Trace

Before exploring the specifics of various trace cache schemes, it is necessary to first precisely define a **trace**. Standard I-caches stores instructions in **static program order**. This layout is the result of the compiler translation of high-level code to the operations that comprise the ISA of the processor. For sequences that don't contain branches, the **dynamic execution order** is the same as the static program order. Of course code also contains control transfer instructions that introduce non-sequential program flow. This causes the dynamic execution order to deviate from that of the static program order. A trace is finite piece of this dynamic execution. The fundamental concept behind trace cache is storage of these "snapshots" of previous execution. When processing returns to these points in the program, the stored traces can be utilized in overcoming the basic block fetch limitation explained in Section 2.5.

<i>Static Program Order</i>		
A: ADDIU R14,R31,#1		
BEQ R2,R14,	(1)	
ADDIU R21,R21,1		
B: SLTU R2,R8,R17		
BNE R2,R31,<C>	(2)	
ADDUI R8,R8,#1		
C: LW R2,0(R8)		
ADDUI R1,R1,#1		
ADDUI R2,R2,#3		
BNE R2,R3,<D>	(3)	
ADDIU R8,R8,#1		
D: J R31		

<i>Dynamic Execution Order</i> (Branches → T, NT, T)		
A: ADDIU R14,R31,#1		
BEQ R2,R14,	(1)	
B: SLTU R2,R8,R17		
BNE R2,R31,<C>	(2)	
ADDUI R8,R8,#1		
C: LW R2,0(R8)		
ADDUI R1,R1,#1		
ADDUI R2,R2,#3		
BNE R2,R3,<D>	(3)	
D: J R31		

Figure 3.1: Static Program Order versus
Dynamic Execution Order

3.2 Rotenberg Trace Cache

Rotenberg, Bennett and Smith proposed one of the first trace cache designs in 1995 [10]. As presented in their paper, trace cache supplements the usual instruction cache by storing individual traces of dynamic instruction streams. Each line of the trace cache contains up to n instructions, and at most m basic blocks. The maximum number of basic blocks within a trace cache line is constrained by the throughput of the multiple branch predictor (MGAg). The reason behind this constraint is the number of conditional branches in a trace cache line cannot exceed the number of parallel results from the predictor to which comparisons are made. Over the course of program execution, instructions that are retired are added to the fill buffer of the trace cache. When the number of instructions or basic blocks in the fill buffer meet n or m respectively, the group of instructions is added as a new line to the trace cache. An example trace cache line is shown in Figure 3.2. In addition to the instruction stream, additional information is recorded. **Branch Flags** indicate the path

followed by each conditional branch within the trace. Since a variable number of branches could occur within a trace, a **Branch Mask** field must also be provided, indicating the number of branches (with a maximum of $m-1$) and whether the trace ends in a branch. Two addresses are also stored, pointing to the next execution point following the last instruction of the trace. One is the **Fall-through Address** (corresponding to a not-taken result for a trace-terminating branch) and the other the **Trace Target Address** (corresponding to a taken result.) If the last instruction in the trace is not a conditional branch, both address fields point to the cache line starting with the next sequential instruction. In addition to the constraints n and m , an indirect jump or subroutine return also terminates a trace.


A	111	11,1	X	Y	
<i>Tag</i>	<i>Branch</i>	<i>Branch</i>	<i>Fall-through</i>	<i>Target</i>	<i>Instructions</i>
	<i>Flags</i>	<i>Mask</i>	<i>Address</i>	<i>Address</i>	

Figure 3.2: Example Trace Cache Segment

During instruction fetch, the trace cache is accessed in parallel with the I-Cache. When program execution presumably returns to an instruction that starts a cache line, the branches within the trace cache line are referenced with the results of the multiple branch predictor. If the predicted branch directions match each of the entries in the Branch Flags field, then the entire trace cache line can be fetched. If the predictions do not match the trace, then blocks are fetched from the instruction cache, and the trace cache line will be updated with the new stream of dynamic instructions.

Returning to the alternate fetch mechanisms that were introduced in the previous chapter, one can note the distinct advantages of the trace cache over the branch address cache and collapsing buffer approaches. The trace cache does not require any I-Cache modifications beyond the two way interleaved organization of the basic fetch unit. In addition, there exist no possibility for bank conflicts

that negate performance gains. The latency of the trace cache when fetching instructions is minimal; no critical path logic for interchange, mask or alignment of instructions is required. Finally, the fetch bandwidth provided by the trace cache is a substantial improvement over the core fetch mechanism, as will be shown in the simulation results of Chapter 5.

Despite the vast improvement over other alternate fetch mechanisms, the trace cache design presented above is limited by a few factors. In fact, Rotenberg, Bennett and Smith [10] suggest several additions to the general trace cache strategy that would yield better performance. One aspect of the scheme that can be improved is how frequently and infrequently visited traces are maintained.

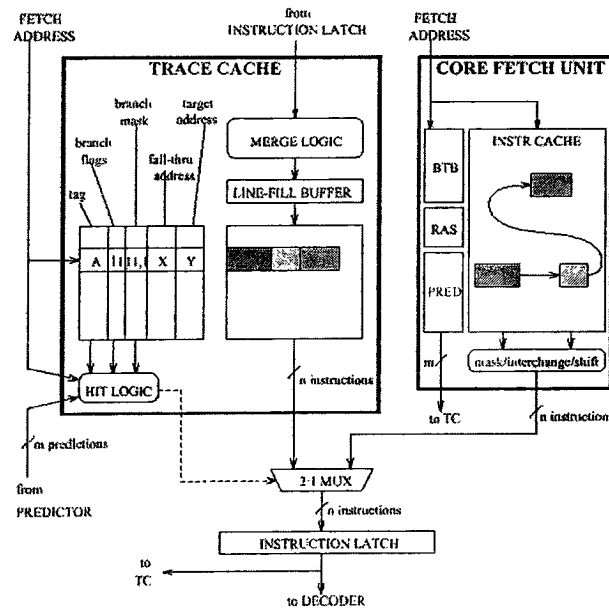


Figure 3.3: The Rotenberg et al Trace Cache [10]

Solutions include judicious trace selection (only committing traces that have been executed a minimum number of times) and a victim trace cache (storing recently displaced traces such that they could easily be recovered if needed.)

Table 3.1 presents a set of general limitations, and corresponding solutions that were subsequently proposed. Many of these schemes will be discussed in this section.

Limitation:	Maximum number of conditional branches within a Trace Cache line.	<i>Section</i>
Solution:	<ul style="list-style-type: none"> • Path-Based Prediction • Branch Promotion/Trace Packing 	(3.3) (3.4)
Limitation:	Trace Cache indexed by address of the first instruction; No multiple paths.	
Solution:	<ul style="list-style-type: none"> • Fetch address renaming 	(3.5)
Limitation:	Trace Cache Miss Rate	
Solution:	<ul style="list-style-type: none"> • Partial Matching /Inactive Issue • Trace Preconstruction 	(3.6) (3.7)
Limitation:	Storage/Resource Inefficiency	
Solution:	<ul style="list-style-type: none"> • Block-based schemes • Cost-regulated trace packing • Selective trace storage • Pre-processing/Fill unit Optimizations 	(3.8) (3.9) (3.10) (3.11)

Table 3.1: Trace Cache Limitations and Solutions

Overcoming the Maximum Number of Conditional Branches within a Trace Cache Line

3.3 Path-based Next-trace Prediction

The trace cache design presented above incorporated explicit multiple branch predictions to determine the next trace. Jacobson, Rotenberg and Smith found it possible to improve the next-trace prediction accuracy by 26% via **path based trace prediction** [13]. Rather than use the individual predictions from a multiple branch predictor, a predictor based solely on the trace path history determines the next trace. The trace cache is indexed via a combination of trace history and the

address of the first instruction in the trace. In contrast to the use of the starting address of the trace alone, this scheme inherently allows for multiple paths (path associativity), as suggested in [10]. The path-based trace predictor scheme incorporates a primary correlated trace predictor and a secondary predictor. The secondary predictor is included so mispredictions are reduced from “cold starts” (when the path history has not yet been established) and aliasing in the correlating table. A block diagram of the combining trace predictor is shown in Figure 3.4.

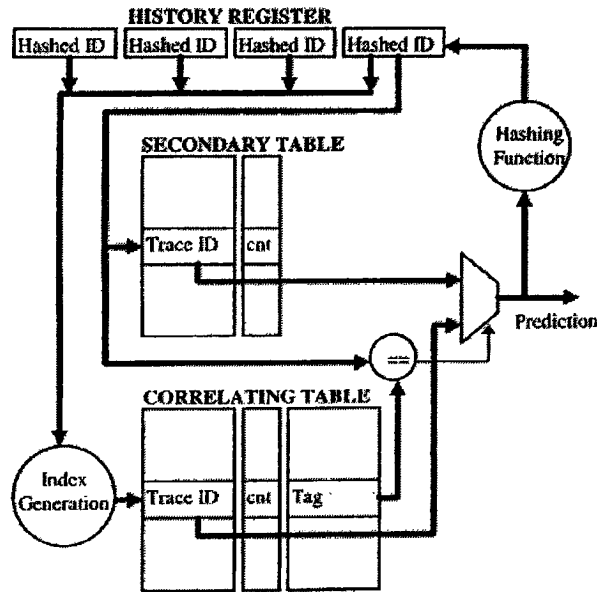


Figure 3.4: Combining Trace Predictor [13]

The primary predictor stores a path history of the previous n traces as a series of hashed history entries for each trace. Each entry is formed from a 36-bit Trace ID consisting of the PC for the starting address in the trace cache line along with up to six bits encoding the conditional branch outcomes within the line. The hashing function is illustrated in Figure 3.5.

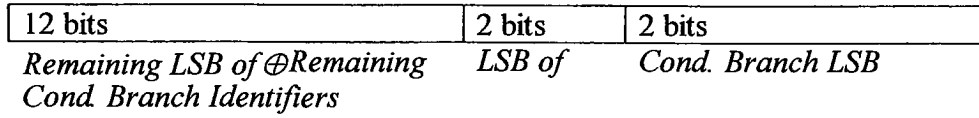


Figure 3.5: Hashed History Register Entry

From the series of entries constituting the history register, an index into the correlating table is formed. Taking a few bits from the history entries generates this index. The most recent entry in the history register is given the most weight in contributing to this index. The second most recent is given a slightly lower weight, while older history entries each contribute the least number of bits. This generation mechanism is shown graphically below in Figure 3.6. The term “DOLC” has been established as the name for this method, each letter representing an element of the scheme. *Depth* is the number of entries, not including the last history entry. *Last* and *Current* correspond to the second most, and most recent history entries. Older entries constitute the remaining entries.

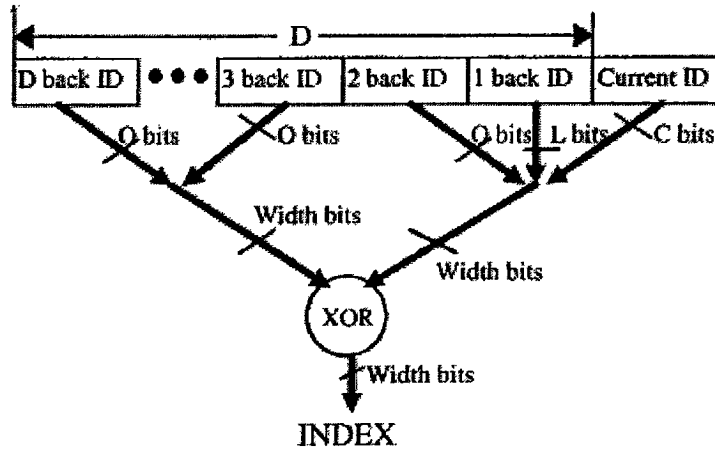


Figure 3.6: DOLC Index Generation [13]

The indexed location in the table provides the Trace ID of the predicted next trace. If this prediction is found to be correct further down the pipeline, then a two-bit saturating counter for the table entry is incremented by one. Otherwise, the counter is decremented by two for a misprediction. If the counter is equal to

zero after being decremented, the Trace ID for that entry in the correlating table is updated to the actual trace. Additionally, if a prediction is incorrect, the speculative entry in the history register is replaced by the hashed history entry of the actual trace.

If an index into the primary correlating table yields an entry that is unrelated (due to aliasing or “cold starts”) the “prediction” has a low percentage of being correct. This is unfortunate, as a conditional branch prediction in a similar scenario would still offer a 50/50 chance of being correct. To compensate for this potential downfall, a secondary predictor is incorporated and a tag field is added to the correlating table. The index value into the secondary table is the hashed history entry of the current trace (the “C” value). When an entry in the correlating table is updated, the least significant ten bits of the preceding entry are stored as the tag bits. When a prediction via the correlating table is made, the tag field of the associated table entry is compared to the current entry in the history register. If the corresponding ten bits match, then the primary prediction is used. Otherwise the entry stored in the secondary table is chosen. Regardless of what table the prediction was chosen from, both saturating counters are incremented or decremented respective to the correctness of each prediction. When the four-bit counter of the secondary predictor saturates (i.e. the next trace always follows the current) updates of the correlating table cease. This avoids clutter of the primary table, further reducing aliasing. The reason that Jacobson, Rotenberg and Smith chose a four-bit counter (decremented by eight on a misprediction) was to “avoid giving up the opportunity to use the correlated predictor unless there is a high probability that a trace has a single successor” [13].

Two additional (rather esoteric) mechanisms can be added to the combining trace predictor. A Return History Stack (RHS) can be provided to augment the performance by taking advantage of the fact that the control flow of a program is tightly correlated before and after a subroutine call. The trace history may not

capture this behavior though, as the trace history of the called subroutine will interfere. Thus, when a trace contains a subroutine call, a copy of the History Register is saved on a separate hardware stack. When a trace that ends in a return is encountered, the entry at the top of the RHS is popped. The most recent Hashed IDs contained in this entry replace the oldest IDs of the current history (the most recent IDs in the History Register are preserved.) It is noted that the RHS provides a way to balance the control flow history prior to the subroutine call and the history of the subroutine itself.

The second mechanism that can be added to the basic trace predictor is the Alternate Trace Field. By adding this field to the Correlating Table a “second guess” can be made at the same time as the regular prediction and be issued inactively (see Section 3.3). The alternate field is updated when the regular prediction is incorrect. When the saturating counter is non-zero the alternate field is updated with the actual next trace. When it is zero, the displaced primary prediction is moved to the alternate field. The alternate path must be issued such that results are not visible to those executing along the path of the primary prediction. Pending the outcome of the branches, the instructions along the alternate path are canceled if the primary prediction or neither prediction was correct. If the alternate prediction proved correct, execution will continue along that path.

3.4 Branch Promotion and Trace Packing

Two additional techniques for improving trace cache effectiveness are examined in [15]: **Branch Promotion** and **Trace Packing**. Branch promotion exploits the fact that over half of conditional branches tend to be strongly biased during execution. By identifying these branches in hardware, they may be treated as statically predicted. These “promoted” conditional branches are marked via a single bit flag, and are associated with a taken or not-taken path. When the fill

unit writes a cache line, the promoted branches are not included in the normal branch mask and flags fields. Thus, the predictor is alleviated from the overhead of storing the branch history for a promoted branch. This decreases aliasing within the predictor. If a static prediction proves incorrect (such as the final iteration of a loop) the machine is backed up to the end of the previous block and restarts along the correct path. The structure that decides to promote a branch to static prediction is the bias table. This table is a simple saturating counter indexed by the branch address (with tag compare.) The counter is incremented when the result of the branch is the same as the previous outcome. The fill unit will check the bias table against conditional branches in the retired instruction stream. If the count is greater than threshold value, the branch is promoted in the fill buffer. If the outcome of a promoted branch contradicts the static prediction twice in a row, the branch is demoted, and the branch bias table entry is cleared. Demotion upon two consecutive mispredictions was established to avoid the final iteration of a loop from demoting an otherwise strongly biased branch. Likewise, if the branch bias count falls below the threshold (the result of an alternating series of taken/non-taken results) the branch is demoted.

Patel et al note that in addition to reducing the volume of branches required to be output by the predictor, it could be possible to implement a single aggressive combining predictor (versus a less accurate multiple branch predictor) in an eight-wide machine. This is because the overwhelming number of trace lines would simply have only a single branch needing a dynamic prediction.

The idea of trace packing involves filling a trace cache line irrespective of fetch block boundaries. Hence if a block is larger than the space available in the pending cache line of the fill unit, the fetch block will be split. After filling the pending line, the remaining instruction will start a new line in the fill unit. For a dynamic sequence of instructions, this method implies there could exist many segments that contain redundant trace information. For example, consider a

sequence of three basic blocks (A, B, and C) that are encountered in a loop (i.e. ABCABC...). The blocks are sized such that two whole blocks could fit in a segment, yet all three would not. A scheme that treats blocks atomically would produce three cache lines: AB, CA, BC. If trace packing were used, the segments would be split such that a much greater amount of trace cache storage would be needed. In the above example, Patel et al showed that for block sizes of 8, 6 and 8 instructions for A, B and C respectively, eleven segments would be created. This results in a loop that is dynamically unrolled: the maximum number of instructions will be fetched per cycle throughout the multiple iterations. Trace packing increases the delivered fetch rate because a larger percentage of segments utilize the maximum number of instruction slots. This comes at the cost of redundant storage that increases the cache size requirement. It is stated in [15] that branch promotion and trace packing are complementary techniques, providing a 17% effective increase in fetch bandwidth when combined.

Allowing Multiple Path in the Trace Cache – Renaming the Fetch Address

3.5 Fetch Address Renaming

The Rotenberg scheme allowed a single trace with a unique fetch address (index value) to occupy the trace cache. A pre-existing trace with the same fetch address as a new trace would be simply overwritten. This can lead to consistent trace cache misses and thrashing of the trace cache if multiple paths with the same starting address occur frequently in the dynamic execution of a program. The generic solution to this problem is to provide a hashing function that generates trace cache indices, or Trace IDs, based on various inputs. The path-based prediction scheme makes use of fetch address renaming. The renamed trace ID consists of the fetch address concatenated with the branch flags. The mechanism

that generates the renamed trace IDs for this scheme also serves as the prediction mechanism (Figure 3.4). This allows the trace cache to remain a direct mapped structure while still providing the advantage of retaining multiple paths.

Improving the Trace Cache Hit Rate

3.6 Partial Matching and Inactive Issue

Friendly, Patel and Patt explored two techniques for improving the performance of trace cache: **Partial Matching** and **Inactive Issue** [14]. Partial matching, as suggested in [10] can cause a match in the trace cache even if only the first few branch predictions correspond. This requires target and fall through addresses for each branch in the trace. Inactive issue presents a variation on partial matching by essentially implementing predication of traces. Instead of discarding portions of a trace cache line that do not match the outcome of the multiple branch prediction, the entire line is issued. The blocks that do not match the prediction are referred to as *inactive*. Though these inactive instructions issue, their results are not visible to instructions executing along the speculative path determined by the multiple predictor. Inactive issue requires additional logic in the issue/scheduling mechanism so in-flight instructions that are inactive are differentiated from those that are along the primary path. If the outcomes of the multiple branch predictions are correct, the inactive instructions are discarded. Though if the predictions prove incorrect, some of the correct path has already been issued and possibly executed though the inactive blocks, and can be promoted to active status.

3.7 Trace Preconstruction

Trace preconstruction, discussed in [19], is a proposed technique to reduce trace cache miss rates, and is somewhat analogous to instruction prefetching. The distinguishing factor between the two methods is that traces are not an explicit part of the memory hierarchy (as are instruction blocks) and therefore must be preconstructed versus prefetched from a lower level of the hierarchy. A fundamental observation allows the preconstruction mechanism to be incorporated into a design that utilizes trace cache: While instructions are being provided via the trace cache, the *slow path* of the fetch unit is idle. The resources that constitute this slow path are the traditional branch predictor, the instruction cache and the trace constructor. The goal of the preconstruction algorithm is to build probable traces prior to the point where they are utilized. Determining these traces is achieved by identifying regions of code that will be encountered later in the course of normal execution. There are two possible types of *region start points*. The first is an instruction following a subroutine branch (jump-and-link.) This represents code that will be executed subsequent to the subroutine return. The second type of region start point is an instruction following a backward branch. A sequence of static code before a backwards branch represents a loop. After the closing branch is evaluated not-taken (following the final iteration), flow will continue at the next instruction.

When encountered, the region start points are pushed onto a *start point stack*. If a start point is identified, but matches the entry at the top of the stack, the duplicate is not pushed. A duplicate of this nature would indicate a loop encountering the closing branch over multiple iterations. The top entry of the stack is popped when the preconstructor is ready to begin working on a new region. This preconstruction effort involves a breadth first traversal of all possible paths, resulting in a set of possible traces. The region start point is used as the first entry in a worklist of *trace start points*. These trace start points differ

from region start points in that they identify where traces within the region may begin. Figure 3.7 shows a directed graph representation of static code and the resulting preconstruction regions and traces.

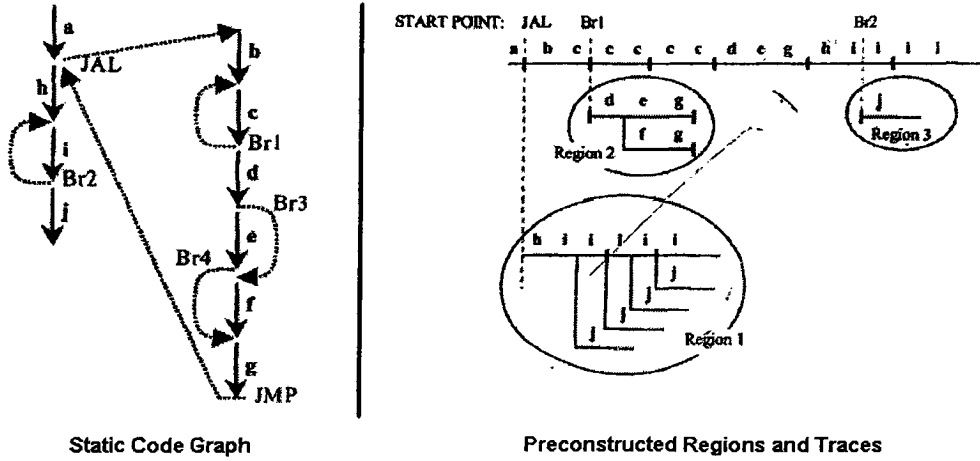


Figure 3.7: Static Code and Corresponding Preconstruction Graph [19]

After block a , a subroutine branch is encountered. The address following this JAL is pushed onto the start point stack. Assuming that the slow path hardware resources are available, the preconstructor generates a set of traces for the region. These traces might be $\langle h, i, j \rangle$ and $\langle h, i, i \rangle$ (assuming a non-packing fill unit). The two corresponding trace start points recorded in the worklist are the starting address of block i , and the address following block j . To reduce the potential number of paths within a region, highly biased branches are followed through their dominant direction only. Jacobson and Smith note that a trace construction scheme that utilizing pure trace packing is likely to defeat trace preconstruction, as it would be unlikely that the actual trace path would align with the region start points. A heuristic of allowing an even multiple of instructions past a block terminating branch (a variation of cost-regulated trace packing) is proposed.

Preconstruction for a region will terminate under one of three conditions. If the processor catches up (i.e. execution reaches the region of code) then the

preconstructor moves onto the next region identified in the start point stack. Each preconstruction region is allocated limited resources; if these resources are consumed, no further preconstructed traces for that region are generated. The preconstruction effort is also bounded by indirect jumps (subroutine returns), as the target is unknown in the static context.

The implementation of trace preconstruction utilizes buffers to avoid pollution of the trace cache. When the trace cache is accessed, the trace preconstruction buffers are accessed in parallel. If the particular index into the trace cache misses, though the entry is found in the preconstruction buffer, then that trace is moved into the trace cache. Similarly, the trace cache is referenced before inserting a preconstructed trace in the buffer. If the trace exists, then the redundancy of inserting into the preconstruction buffer is avoided. If the preconstruction buffers are organized in a set associative fashion, the replacement policy is governed by the region from which the traces were constructed. When execution catches up to a preconstructed region, it is termed the *active region*. No traces in the buffers are replaced if they are from the active region. The preconstruction buffer replacement policy then gives precedence to traces from newer regions. A small separate buffer records the most recent regions from which traces were preconstructed. If one of the same regions is again identified in the start point stack, it is skipped to avoid redundant processing.

Two hardware implementation optimizations are suggested in [19]. One is the use of a small prefetch cache. Since instructions within a region are used in multiple preconstructed trace segments, it is advantageous to have a small (256 instruction) cache to avoid the latency associated with accessing the instruction redundantly. Allowing this prefetch cache to be fully associative, and not allowing replacement once a cache has been assigned a segment effectively contains the preconstruction effort for a region by halting once the prefetch cache has been filled. After the prefetch cache is full, the cache is invalidated, and preconstruction starts with a new start point off the stack. The second

optimization involves incorporating multiple parallel trace preconstructors, each one able to work from a different trace start point, be it from a single or from multiple regions.

Jacobson and Smith report performance of the trace preconstruction mechanism in terms of total size of traces. This includes both the trace cache and trace preconstruction buffers. For many programs, there is a distinct tradeoff between allocating the available space resources between the two entities. As a possibility for further investigation, dynamic portioning of the trace cache/preconstruction buffer is suggested. The most significant advantage to preconstruction is when paired with pre-processing enhancements, such as those discussed in [16] or in the next section.

To conclude this section, a unique approach to modeling trace cache enhanced systems is mentioned. This **extended pipeline model** [19] considers a processor from the perspective of three independent pipelines: The instruction pre-processing, fetch/decode, and execution pipelines. The entity that decouples the pre-processing from the fetch/decode portion is the trace cache, much like shelving decouples issue from dispatch and execute.

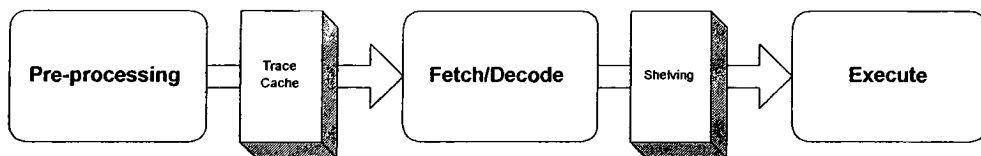


Figure 3.8: Extended Pipeline Model

Improving Storage and Resource Efficiency

3.8 Block Cache

The schemes of the last few sections proposed enhancement to a design based on a “traditional” trace cache. Black et al suggest a design that involves separating

the trace cache entity into a **Trace Table** and **Block Cache** [17]. A diagram representation of this scheme, compared with a traditional trace cache, is shown in Figure 3.9. The Block Cache, replicated w times, stores basic blocks at unique locations. A re-named identifier indexes each block in the replicated table. This method is similar to the one used in the correlating and secondary tables of the path based prediction scheme of [13] (versus the by-address indexing scheme used in [10].) The trace table is similar in purpose to the traditional trace cache entity, but instead of storing traces explicitly, the trace table stores a sequence of block cache indices, along with some additional information. A trace table read proceeds as follows: First, the block ID and branch history are hashed to form the predicted next trace ID. The trace ID serves as an index into the trace table. The block IDs from the trace table entry are retrieved from the block cache. Because the individual blocks are stored without respect to any trace order, they must be aligned when retrieved to form the group of instructions send to the decode unit. This instruction collapse is directed by a set of steering bits included with each Block ID within a trace table entry. This step adds some additional overhead to the critical path of the fetch stage (handled in the traditional trace cache by the fill unit.) Black et al. state that the collapse/alignment step may be efficiently implemented in hardware and its addition would not add appreciable latency.

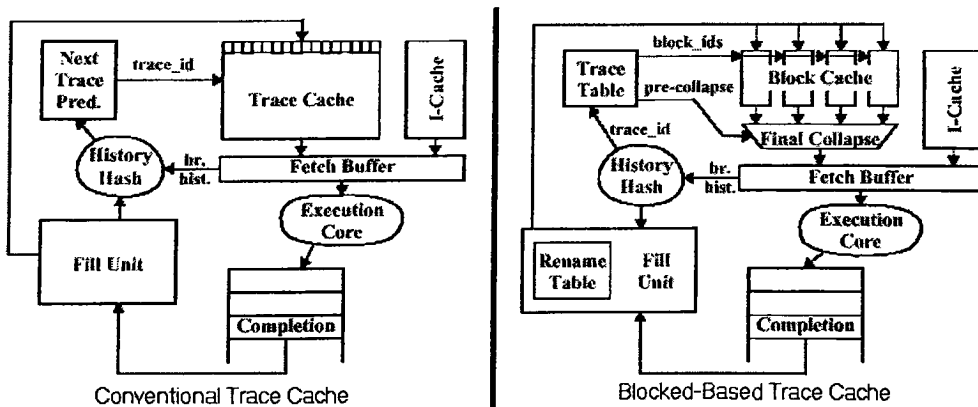


Figure 3.9: Conventional versus Blocked-Based Trace Cache [17]

When instructions retire, they are captured in order by the fill buffer. When a conditional branch is encountered or the physical block size limit is reached, the block fetch address (the address of the first instruction in the buffer) is compared to the entries of the rename table. If an entry already exists, then the fill buffer is simply flushed. Otherwise, the block fetch address is inserted into the set-associative, tag-checked rename table (LRU replacement policy) and the block is written to the location in the block cache referenced by the assigned Block ID. The newly allocated block can already exist in traces in the trace table. If one of these traces is issued as a next-trace prediction, the block re-identified with the Block ID will be erroneously issued. This case is handled as a misprediction. Traces are formed via the unit, recording the sequence of Block IDs, and are subsequently inserted into the trace table.

The motivation behind implementing the Block Cache is to avoid redundant storage of basic blocks in multiple traces. With ideal branch prediction, it is claimed that a 1K block cache achieves the same performance of a 32K traditional trace cache. The next section suggests another optimization geared at eliminating storage redundancy between the I-cache and trace cache.

3.9 Cost Regulated Trace Packing

A modified trace packing scheme adds a fill unit constraint: If the number of unused instruction slots is greater than or equal to half the size of next block to be written to the fill buffer, or, the last branch in the fill buffer has a backwards displacement of 32 or fewer instructions, trace packing is used. Termed **cost-regulated trace packing**, this allows space at the ends of segments that would otherwise be un-utilized to be filled. Small loops benefits from this dynamic unrolling. At the same time, space is conserved and trace alignment is improved (compared to full trace packing) [15].

3.10 Selective Trace Storage

Ramirez et al propose a scheme to avoid redundancy between the instruction and trace cache through **selective trace storage (STS)** [18]. Combined with a compiler optimization called software trace cache, they show that a 128 KB trace cache can be reduced to 2 KB and retain performance while lowering implementation cost. The redundancy targeted in this scheme is sequences of fetch blocks that exist contiguously not only in the trace cache, but also statically in the instruction cache.

Fetching multiple contiguous blocks from instruction cache in a single cycle has been shown to be fairly straightforward (*Revision One* of the fetch unit: Chapter 2, Section 5). When fetching a set of instructions whose branches are all predicted non-taken, no efficiency is gained by accessing the trace cache over the instruction cache. Such a trace (with no taken branches) is referred to as a *blue trace*. Ramirez et al. state that blue traces account for over 70% of program layout for floating point and 40% for integer applications (16 instruction fetch width). Software trace cache can increase the percentage for integer applications to 50 to 88%. Storing blue traces in the trace cache is allocating duplicate segments, possibly evicting a segment which contains *predicted* taken branches. Adding logic to the fill unit can recognize blue traces and simply ignore them when writing to the trace cache, alleviating the load on the trace cache substantially. *Red traces* (those constructed by the fill unit containing predicted taken branches) are stored in the trace cache as before.

3.11 Fill Unit Optimizations

Four optimization mechanisms are proposed by Friendly et al in [16] that can be incorporated with the fill unit of the trace cache. These constitute the first of the instruction pre-processing techniques that are examined. Placing extra logic in the fill unit does not hinder the fetch efficiency, as it is outside the critical path of

the fetch mechanism. Also, it is shown that the fill unit latency has little effect on performance, negligibly impacting the measure of fetch instructions per cycle when varied from 1 to 10 cycles.

The first fill unit optimization is to flag register-to-register move operations. This allows the register renaming mechanism to simply map the destination architectural register to the physical register which contains the data of the source. The advantage is that no resources (i.e. functional units) are occupied by the instruction. This is accomplished in an architecture that implements a reorder buffer. An entry for the move instruction is added to the tail of the reorder buffer. If the source operand (architectural) of the move operation is identified as “busy” (i.e. a previous instruction is waiting to commit its result) then a physical register with that result has been allocated. The value field of the reorder buffer entry will point to this physical register, or if the source was identified as not “busy”, the physical register identified by the architectural state will be used. The illustration in Figure 3.10 is based upon the code sample:

```
1) ADDI R1,R31,#30
2) ADDI R2,R31,#44
.
.
.
3) ADDI R1,R1,#5
4) ADD R2,R1,R31
```

The final instruction implements a register-to-register move by adding zero to the source R1 and storing the result in R2. The first two instructions are committed by the time the third instruction is fetched. Since no in-flight dependencies exist, the third instruction can be assigned a ROB entry (entry 1), issued and dispatched immediately. The fourth instruction, tagged as a register-to-register move in the trace cache, is not issued in the usual fashion. It is neither placed in a reservation station, nor handled by a functional unit. Instead it is allocated a reorder buffer

entry (entry 2), and points to the physical register allocated for the result of the third instruction (Figure 3.10a).

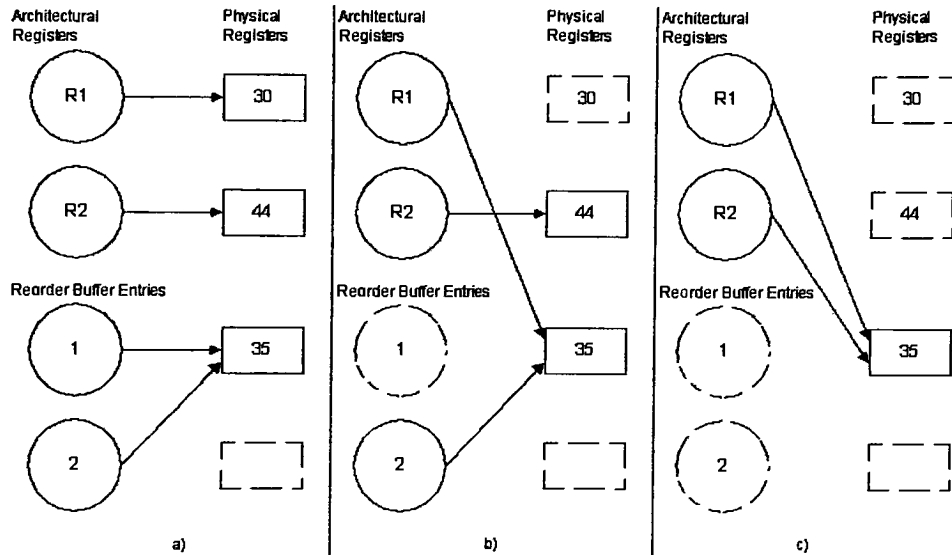


Figure 3.10: Register-to-register move optimization

The state of the system after instructions 3 and 4 are committed is shown in Figure 3.10b and 3.10c respectively. Dotted entries or physical register represent those which have been released to the available resource pool.

The second fill unit optimization is similar to a common compiler optimization. Called reassociation, it involves a series of immediate instructions such as:

```
ADDI R2, R1, #4
ADDI R3, R2, #4
```

and substitutes `ADDI R3, R1, #8` for the second instruction. This eliminates the data dependency between the two instructions. The advantage of this fill unit optimization over the compiler is that reassociation over block boundaries can be performed.

The third optimization presented is similar to reassociation in that it removes data dependencies between a pair of instructions. By adding the ability to perform scaled adds to the functional unit (i.e. `SLLADD Z, Y, X, i`, described in RTL as $Z \leftarrow (X \ll i) + Y$) the second instruction of the sequence

```
SLLI R2, R1, #1
ADDI R3, R2, R4
```

can be replaced with `SLLADD R3, R4, R1, #4`. Friendly et al state that the additional logic required for a scaled add instruction would be minimal and would not be a contributing factor to the critical path of the functional unit. As a conservative measure, the shift amount is limited to three.

In a superscalar processor, it is likely that the execution pipes will be clustered. This implies that the result of an instruction will be available to dependent instructions within the cluster the cycle immediately following execution. Because of implementation constraints, the result is not available globally until a certain delay. The fill unit can alleviate this cross cluster delay penalty by indicating the functional unit cluster to which the instructions should be issued. In addition to the single bit used for register-to-register move flag, four bits are required for each instruction within a segment to convey the order/EU steering information.

3.12 An Alternate Approach to Filling the Trace Cache

Prior to the initial Rotenberg paper on trace cache, a US patent was filed for a *Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line* [20]. Described is a mechanism that closely approximates the trace cache concept (published a year later). One interesting aspect of this pioneering design is the fill unit policy. The Rotenberg scheme entails flushing

the fill buffer to the trace cache once the maximum number of instructions (n) or basic blocks (m) has been reached. The next instruction to retire will be added to the empty fill buffer as the first instruction of a future trace. The fill method described in [20] differs by committing a trace line when n or m has been reached, then discarding the frontmost (oldest) basic block from the fill buffer and shifting the remaining instructions to free room for newly retired instructions. If effect, every new basic block encountered in the dynamic instruction stream causes a new trace to be added to the cache. This fill mechanism can be implemented as a circular buffer. Figure 3.11 provides a visual comparison between the Rotenberg et al approach and the method derived from [20,21].

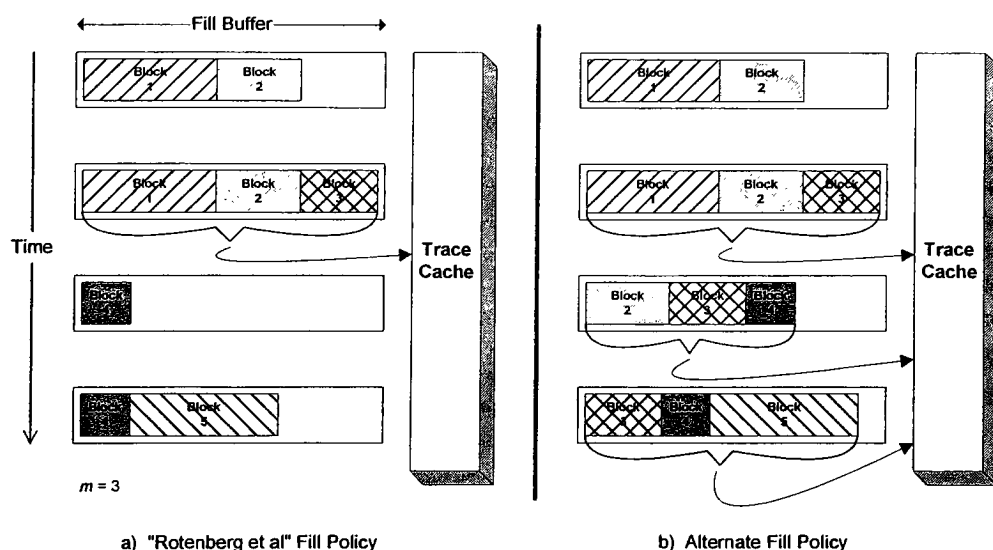


Figure 3.11: Rotenberg Fill versus the Fill Policy of [20]

3.13 Additional Related Research

To conclude the discussion of background theory, this section briefly identifies other areas that are closely related or derived from trace cache research.

Two mechanisms to determine the next trace were mentioned in this chapter: Multiple branch prediction using an extended GAg predictor (MGAg), and path-based prediction. Several other prediction schemes, especially of the explicit multiple branch variety have been proposed. A two level scheme that utilizes m sets of two-bit counters for each PHT entry is mentioned in [22] and [23]. A variation involves storing 2^m sets of two-bit counters arranged in a tree structure (similar to that of a BAC entry) [22]. These provide straightforward extensions that can be applied to a per-address scheme such as PAg, or the aggressive gshare single branch predictor. The drawback (especially for the 2^m version) is poor scalability with increasing values of m .

Rotenberg et al propose a novel architecture in [24]: The Trace Processor. This architecture is organized around the trace cache and treats the trace as the basic unit of processing. The complexity required to overcome ILP limitations in superscalar processors is eliminated in the trace processor by “distributing execution resources based on trace boundaries” and applying path-based prediction. By incorporating multiple functional units (that are essentially basic superscalar processors), traces are distributed as atomic units. Intra-trace processing utilizes resources (registers, shelving and rename buffers, etc.) within a functional unit, leaving only inter-trace resources to be allocated on a global scale. Further discussion on trace processors is beyond the scope of this chapter; the reader is encouraged to refer to [24] and [25] for more detail.

The concept of trace cache has brought the realm of instruction optimizations to the dynamic, hardware-based level. The Instruction Path Coprocessor (I-COP) is a unit that optimizes blocks of instructions according to its own programmed behavior. The I-COP has its own instruction set, allowing it to be extremely versatile and extensible as a dynamic instruction preprocessor. In addition to being able to perform trace optimizations (such as those discussed in Section 3.5), the I-COP can direct data prefetch for the I-Cache and

preconstruction for the trace cache. Papers that present further background and implementation specifics for the I-COP include [26] and [27].

Patel and Lumetta propose a framework (rePLAY) for dynamic program optimization that closely resembles the structure of the trace cache. The notion a *frame* is central to this framework, defined as a dynamic instruction stream with a single entry and exit point. Under a usual set of assumptions, one would identify a frame as synonymous to a basic block. The distinction is made when utilizing branch promotion. Strongly biased branches are replaced with an ASSERT instruction. If an ASSERT instruction “fires” during execution (i.e. the promoted branch is mispredicted) then the entire frame is canceled. Frames can span multiple lines in the *frame cache*, providing the opportunity to optimize sequences of instructions that far exceed the cache segment size of 16 instructions. After a frame is assembled, an entire host of optimizations can be performed irrespective to the boundaries formed by the ASSERT statements. It is claimed in [28] that a specific sequence of 136 instructions was reduced to 41 after the frame was processed by the optimization engine. The techniques incorporated in the optimization engine draw from classical compiler optimizations, extended basic block optimizations and software-based dynamic optimizations, all of which are referenced in [28].

3.14 Trace Cache Implementation: The Pentium 4

To date, the only hardware implementation to utilize a trace cache is the Intel Pentium 4 (Willamette) processor. Information published concerning the utilized scheme is limited, though some broad details have been made available [29]. The trace cache of the Pentium 4 replaces the level one I-Cache completely (versus co-existing, as is assumed in most research). The scope of how the trace cache is used in the context of the larger microarchitecture is also interesting. Standard approaches in modern machines executing the X86 instruction set have been to

convert the cumbersome CISC instructions to an internal RISC equivalent (Intel terms these uops). Since the overhead of the CISC→RISC translation is costly, the trace cache stores the decoded uops instead of the X86 instructions. A single trace cache line can accommodate 6 uops, with a total cache size of 12,000 uops. The branch prediction hardware includes a 4096 entry BTB, a 16 entry RAS and a “highly advanced” prediction unit that is not elaborated further. A separate microcode ROM allows X86 instructions with a complex uop decoding to be represented in the trace cache with a single “placeholder” op. When one of these placeholder ops is encountered, it is used to index the microcode ROM to retrieve the corresponding sequence of uops to be passed down the pipeline. Intel’s motivation to utilizing trace cache seems to be twofold: First the trace cache provides an efficient mechanism for fetching across basic blocks, as has been thoroughly discussed. Secondly, the trace cache is a suitable structure for storing decoded uops in the context of a decoded instruction stream. This second aspect may even overshadow the first, as the overhead of X86 translation is extensive. Being able to fetch instructions “behind” these decoders whenever possible is akin to the performance benefits of avoiding mispredictions.

Chapter 4

Simulator Extension Design

A large portion of research for this thesis was performed via functional simulation. This section describes the processor simulator used as a starting point, and how the trace cache extension was incorporated. Object-oriented C++ was used to implement the extension for reasons of flexibility and extensibility. The three level class design (Creational, Component and Helper) is explained in the context of the Uniform Trace Cache Model.

4.1 Existing Superscalar Simulator

Any proposed microarchitectural improvement is first verified through simulation. As part of the research for this thesis, several trace cache schemes were tested using a custom-designed extension to an existing simulator. The functional simulator used as a starting point was *sim-outorder*, which models a superscalar, out-of-order issue pipelined CPU. The pipeline is broken into 6 stages: Fetch, Dispatch, Issue, Execute, Writeback, and Commit. The fetch stage retrieves instruction and adds them to the instruction fetch queue (IFQ). Multiple instructions are fetched each cycle until one of three conditions becomes true: 1) the maximum number of instructions per fetch cycle has been reached, 2) the IFQ is full, or 3) a hit in the BTB occurs. The dispatch stage removes instructions sequentially from the IFQ, decodes the instruction and allocates a new entry in the RUU. After all dependencies for an RUU entry have been resolved, the instruction moves to a functional unit for execution. After executing, the instruction is marked complete and the result is broadcast through the RUU to update the state of any dependent instruction. In the commit stage, the simulator traverses the RUU in reorder buffer fashion. Starting at the head,

any instruction marked complete is retired and the associated RUU entry is deallocated. This traversal continues until an instruction is encountered whose result is still pending. A diagram the *sim-outorder* pipeline is shown in Figure 4.1.

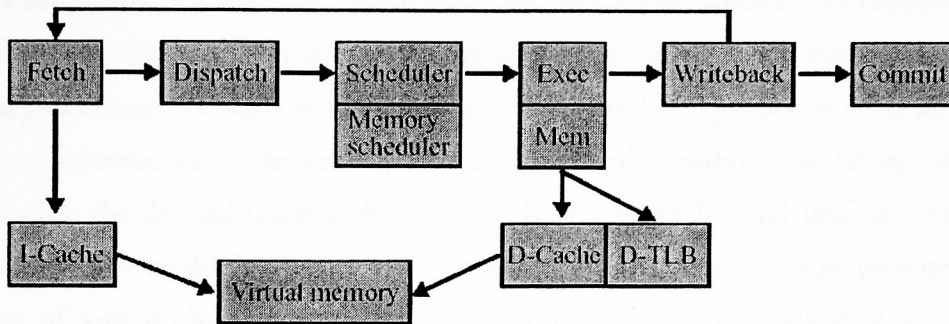


Figure 4.1: *Sim-outorder* Pipeline [32]

The six pipeline stages are each implemented in code as separate functions. The main loop of the simulator (*sim-main*) is structured as follows:

```

ruu_init();
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}

```

Calling the functions associated with each stage in a reverse order allows for proper synchronization between stages, modeling the data availability of a system that latches the output from each stage on a periodic clock edge. The simulator was written using procedural C in a manner that utilizes a multitude of file-scope variables. The next section provides motivation for implementing the trace cache simulator extension in object oriented C++; an approach that sharply contrasts with the design of *sim-outorder* itself.

4.2 Motivation for Utilizing an Object Oriented Extension

The primary goals associated with the design of the trace cache extension were flexibility and extensibility. More than one trace cache scheme was examined, therefore the extension needed to accommodate various data structures and organizations. Similarly, the simulator extension needed to be able to easily allow new schemes to be implemented and tested, preferably without modifying code outside the defined framework. These goals of extensibility and flexibility were achieved by establishing interfaces. From one aspect, the trace cache presents a set of virtual classes from which an individual looking to implement a new scheme would simply inherit. The extension also presents an interface to the *sim_outorder* simulator by abstracting trace cache operations (lookup, update, etc.) via virtual base class methods. These methods are called from a select few points in the *sim_outorder* code, establishing a solid separation of datapath and trace cache logic.

Implementing the simulator extension in C++ provided the means to fulfill the above goals in an object-oriented design. Several object-oriented principles such as encapsulation and polymorphism were heavily utilized. The simulator extension also made use of creational and relational design patterns. Not only did this approach provide extensibility and flexibility within this project, but established a platform for further expansion that should be intuitive for any individual with a background in OO software design.

4.3 Modifications to *sim-outorder* Source

A conscious effort to introduce as few modifications of the *sim_outorder* source file was made. Most of the code related to the extension is resident with the framework of the object-oriented design. The few modifications to the *sim_outorder* source code (renamed *sim_tg*) are limited as follows:

- Addition of trace cache related options and statistics using the SimpleScalar options/statistics databases (see [33] for more details).
- Top-level initializations of the multiple branch predictor, core fetch unit, and trace cache related components.
- Modified fetch stage (`ruu_fetch`) that incorporates trace cache lookups (`tc->fetchTraceSegment()`) and core fetch unit lookups (`fu->fetchInstructions()`). The `CoreFetchUnit` replaces the existing fetch stage as it more accurately models the Revision 1 design of Section 2.5. Multiple branch prediction lookups (`mpred->lookup()`) are performed prior to trace cache/core fetch unit accesses.
- Addition of fields to `fetch_rec` and `RUU_station` structures. Accommodates specific information, such as `PredictionTokens`, that must be “piggybacked” along with instructions as they precede though the pipeline.
- Identification of instructions that missed in the BTB such that the multiple branch predictor can be updated appropriately (`ruu_dispatch`).
- Call to `mpred->speculativeUpdate()` provided in dispatch stage.
- Hook to `tc->retireInstruction()` and `fu->retireInstruction` in the commit stage (`ruu_commit`). Initiates all back-end operations (fill unit, predictor updates, etc.).
- Clean up on mis-speculation that deallocates memory associated with “piggybacked” information for instructions that are canceled.
- General deallocation of trace cache components upon simulation completion.

4.3 High Level Design

The simulator extension was designed to accommodate trace cache schemes with a fair amount of complexity in a modular fashion. Using this approach, schemes ranging from the simple Rotenberg Trace Cache (Section 3.2) to the more complex Path-Based Next Trace scheme (Section 3.3) could be implemented within the same uniform template. A block diagram of this model is illustrated in Figure 4.2.

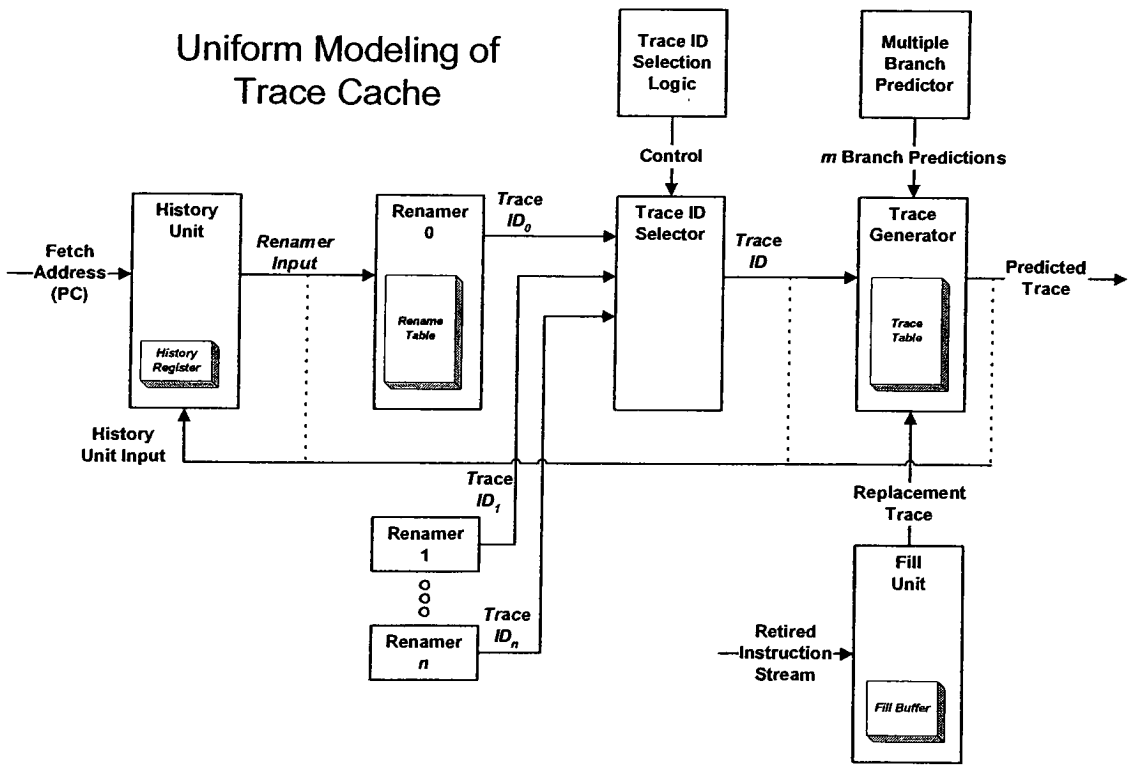


Figure 4.2: Uniform Trace Cache Model

This **Uniform Trace Cache Model** served as the basis for the class structure in C++, which can be divided into a three level design. The first level comprises the interface to the modified sim-outorder source and classes associated with the

creational pattern. The second encompasses the component classes, each roughly corresponding to a block in the above diagram. The third set consists of helper classes that include encapsulated data representations and exception classes. The remainder of this section provides an overview of each of the three design levels.

4.4 Component Level Classes

The set of component classes that form the interface (virtual base classes) from which scheme-specific implementations can be derived from consist of:

- HistoryUnit
- Renamer
- TraceIDSelector
- TraceGenerator
- FillUnit
- Predictor

Collectively, these constitute the *abstract component classes*. Using these interfaces, the top-level `TraceCache` class directs the tasks of lookups and updates. The polymorphic functions of the underlying *concrete component classes* are called and perform the scheme specific tasks related to that component. This process is best illustrated by a code snippet from the `lookup` method of the `TraceCache` class:

```
for ( RenamerVector::iterator iter = _renamers->begin();
      iter != _renamers->end();
      iter++ ) {

    ren_in = _history_unit->generateRenamerInput(
                                           fetch_pred_PC, *iter );
    ( *iter )->setCurrentTraceID( ren_in );
}
trace_id = &(amp; _trace_id_selector->selectTraceID() );
trace = &(amp; _trace_generator->lookupTrace( *trace_id ) );
```

For schemes such as those using Path-Based Prediction, more than one technique can generate an index into the trace table. The design of the simulator extension provides for this by allowing multiple Renamers. In the above code, the `for`

loop polls each of these renamers with the appropriate input data provided by the HistoryUnit. The TraceIDSelector utilizes an Observer pattern [34] to choose the appropriate ID to be forwarded on to the TraceGenerator. When the TraceCache is constructed (described shortly) the renamers are registered with the TraceIDSelector. The selectTraceID method of the TraceIDSelector chooses the index value used to access the TraceGenerator. Each of the component classes contain references to the TCLLogger class that acts as “writer” class for outputting the various configuration and simulation result data. The UML descriptions for each of the abstract components are illustrated in the following diagrams (Figures 4.3 and 4.4).

Component Classes

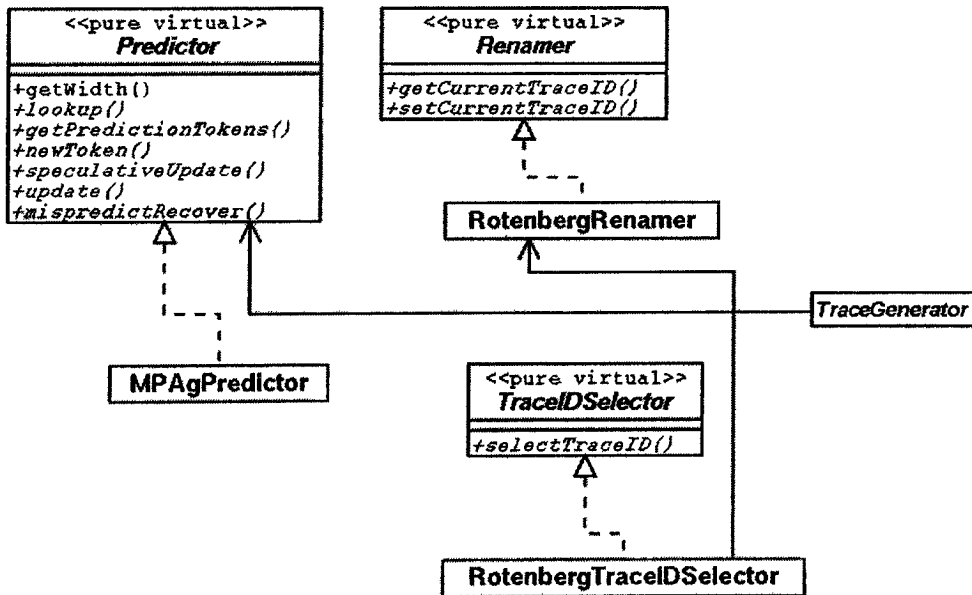


Figure 4.3: UML Diagram of the Abstract Component Classes

Component Classes

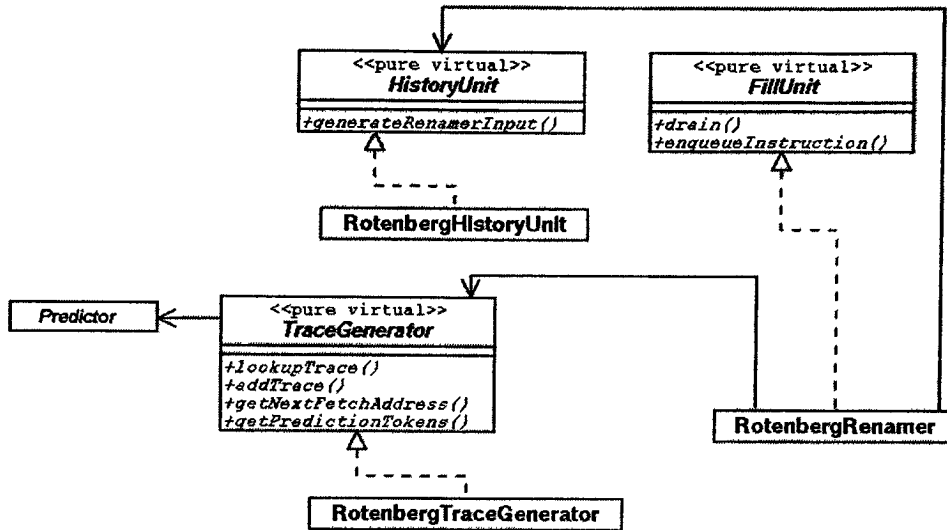


Figure 4.4: UML Diagram of the Abstract Component Classes
(cont.)

4.5 Creational Level Classes

The abstract components define a set of methods that establish a uniform interface for accessing the trace cache. The concrete classes derived from the abstract components are aggregates of the `TraceCache` class. In addition to being the top-level structure that provides an interface to the modified sim-outorder source file, `TraceCache` serves as the Client in an Abstract Factory creational pattern [34]. This design pattern alleviates the `TraceCache` class from requiring knowledge of the specific scheme being implemented. Any scheme that realizes the abstract components must also provide a concrete implementation of the `TraceCacheFactory` and `TraceCacheConnector`. These classes hide the logic that create and establish the interconnections between concrete instances of the trace cache components. When an instance of the `TraceCache` class is instantiated, the `Factory` and `Connector` are passed as arguments to the

constructor. Simply calling the *create* methods of the Factory followed by the *check* and *connect* methods of the Connector return polymorphic references to the concrete components. A UML description of the TraceCache and the associated classes consisting of the abstract factory pattern is provided in Figure 4.3.

Creational Pattern

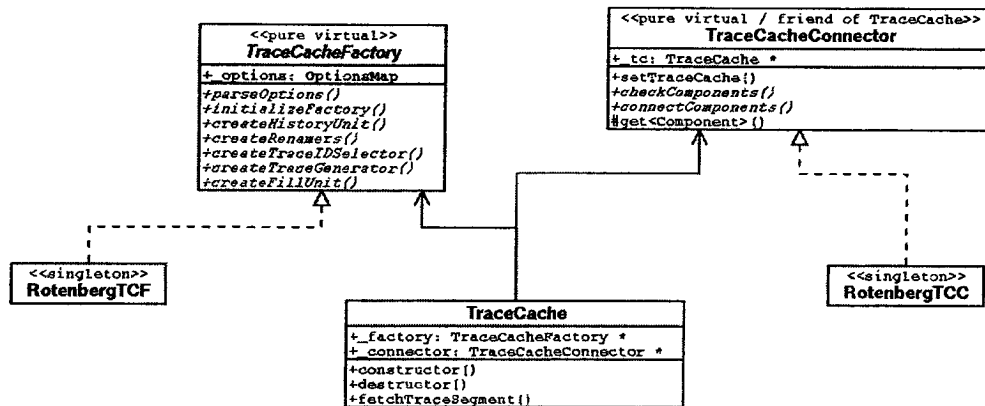


Figure 4.3: UML Diagram of the Creational Level Classes

4.5 Helper Level Classes

The final level of the object-oriented design comprises helper classes. Several data types that form the basis of a trace cache implementation are encapsulated in class form, all of which derive from the common `TraceCacheData` interface. These include `Instruction`, `Address`, `Trace` and derivatives of `PredictionToken`. These classes convey data between components and act as primitives for storage structures within the concrete component implementations. Classes such as `Instruction` provide useful methods for classifying and identifying the data (i.e. `isDirectJump`, or `promoteBranch`). Others simply assume a role similar to the *Memento* pattern [34] by externalizing state information that is carried down the pipeline (`PredictionToken`). Classes that are also included as

helpers include exception classes (InvalidOptionException, InstructionNotAddedException, etc.) and basic Interfaces (Outputable, Printable). These help establish a strongly typed object-oriented design.

Chapter 5

Implementation and Results

This section analyzes the results from simulations that examine various aspects of trace cache organizations. A new fill unit scheme, the Sliding Window Fill Mechanism, is presented. This method exploits trace continuity and identifies probable start regions to improve trace cache hit rate. A 7% hit rate increase was observed over the Rotenberg fill mechanism. Together with Branch Promotion, trace cache hit rates experienced a 19% average increase along with a 17% average rise in fetch bandwidth.

5.1 Simulation Strategy

A progressive simulation strategy was used for investigating trace cache as an extension to wide-issue superscalar processors. Many of the individual simulations build upon results and conclusions drawn from prior trials. To begin, the benchmark set was run on the *sim-outorder* simulator as exists in the SimpleScalar distribution. This established baseline performance results to which subsequent trials were compared. Specifically, all speedup metrics are expressed as:

$$\text{Speedup} = \frac{\text{IPC of Target Scheme}}{\text{IPC of Base Simulator}}$$

The *sim-outorder* parameters that were used in the simulation are given below in Table 5.1. Parameters that are not listed (such as latencies and penalties) were assigned default values by *sim-outorder*. The sizes of the IFQ, RUU/LSQ, the widths of the decode, issue and commit units, and number of EUs are generous. These large values generally alleviate the problem of bottlenecks within portions of the pipeline that are not directly related to fetch bandwidth. Also, the *sim*

outorder default for the I-Cache line size was doubled to help in approximating (albeit, not perfectly) the behavior of an interleaved cache as described in Revision 1 of Section 2.5.

Investigation of trace cache schemes was preceded by an examination of multiple branch prediction performance. Variations of two different methods were studied, as will be discussed later. Simulation of trace cache schemes proceeded as follows:

- Rotenberg Trace Cache [10]
- Trace Cache Associativity
- Reassociation as an Instruction Pre-processing Technique [16]
- Branch Promotion [15]
- An Alternate Fill Method [20]

A new approach to the trace cache fill unit, the Sliding Window Fill Mechanism, is proposed and simulated. To conclude the simulations, a combination of schemes that yielded favorable results was examined.

<i>Simulation Parameter</i>	<i>Value</i>
Decode/Issue/Commit Widths	16
IFQ Entries	128
RUU/LSQ Entries	512/256
Integer ALUs/Multiplication Units	16/4
FP ALUs/Multiplication Units	16/4
L1 I-Cache	
Size (lines)	1024
Line size	32
Associativity	Direct Mapped
Branch Predictor	
PAT entries	1024
PAT width (bits)	10
PHT entries	1024
BTB	
Entries	512
Associativity	4-way
RAS depth	8

Table 5.1: Base *sim-outorder* parameters

5.2 Benchmarks Used

The benchmarks chosen represent a variety of modern computational problems while possessing the benefits of relatively compact sizes and short run-times. The latter aspect was important to this study, as several trials were run to gain a perspective on a variety of schemes and organizations. To do so within a reasonable amount of time using available resources, it was desired that a simulation run complete within the span of a few hours. Ten programs constituted the benchmark set. Table 5.2 presents a description of the benchmarks, the input sets, the respective type of computation that each is characteristic of and the percentage of time that each took to complete on average. This last statistic is indicative of the size of each program. Appendix B contains source code listings of the first five programs. The source for *gzip* is freely available under the gnu license, and is included in larger integer benchmark suites such as SPECInt. The two JPEG programs and the input image were taken from the MediaBench suite [30]. The benchmark performing the Fast Fourier Transform was taken from a suite (MiBench) designed to benchmark embedded-type applications [31]. Appendix C provides further details on the input sets and parameters used.

Before continuing on to present and analyze various result related to trace cache, this section concludes with a basic statistic for each of the benchmarks. Table 5.3 presents the average basic block size for each program, as measured using the unmodified *sim-outorder* simulator.

<i>Benchmark Name</i>	<i>Description</i>	<i>Input Set</i>	<i>Computation Type</i>	<i>Run-time Fraction</i>
roots	Finds root of a nonlinear equation via Bisection, Secant and Newton's methods.	internal	Scientific Kernel	1.1%
solve	Computes the root approximation for a polynomial function using the bisection method.	solve.in	Scientific Kernel	0.8%
integ	Finds the approximate value for the definite integral of $f(x)$ using Simpson's rule and the Trapezoidal rule.	integ.in	Scientific Kernel	1.0%
lag	Lagrange interpolation of a set of data points, and returns the value of function at x .	lag.in	Scientific Kernel	0.3%
matrix	Matrix multiplication	md1/md2	Scientific Kernel	0.6%
gzip	File compression using Lempel-Ziv coding	Thesis Proposal.doc	Integer	26.8%
djpeg	JPEG image decompression	testing.jpg	Multimedia	17.6%
cjpeg	JPEG image compression	testing.ppm	Multimedia	24.4%
fft	Performs Fast Fourier Transform on n random sinusoids sampling m times.	$n=4, m=128$	Embedded/DSP	10.1%
inv_fft	Performs the inverse FFT	$n=4, m=128$	Embedded/DSP	17.3%

Table 5.2: Benchmark Set Utilized

Benchmark	Average Basic Block Size
roots	7.12
solve	6.83
integ	5.00
lag	5.66
matrix	5.05
gzip	5.79
djpeg	19.65
cjpeg	11.50
fft	6.19
inv fft	6.24

Table 5.3: Average Basic Block Sizes

5.3 Multiple Branch Predictors

As discussed previously, a trace cache that is based on conventional branch prediction (versus next-trace prediction; see Section 3.3) requires a predictor that can supply up to m branch predictions per cycle. Multiple branch predictors are often extensions of established single branch prediction schemes. This is true for the two varieties that are examined in this thesis: the MGAg and MPAg predictors. The MGAg predictor is an extension of the GAg global predictor that accesses the pattern history table in an iterative fashion to generate m predictions (Section 2.5). The MPAg predictor is a straightforward derivative of the PAg per-address scheme, where m pattern histories are stored per PHT entry (Section 3.13). Both schemes were implemented using non-speculative updates and pattern history tables (PHT) of 1024 entries. The MGAg scheme utilized a 10-bit wide global history register, while the MPAg scheme used a 1024 entry per-address table (PAT) with 10-bit wide entries.

In a design, the branch predictor needs to be considered with regard to the rest of the architecture. Bandwidth, update policy, table sizes and entry widths constitute a few of these considerations, though one important issue made itself

apparent while forming the simulator extension. This issue entails how direct jumps are handled in the context of the branch predictor. The two options that exist are to include direct jumps as part of the prediction history, or to exclude them. The decision relies on two aspects of the fetch subsystem, one involving the core fetch mechanism, the other regarding the trace cache. Discussion of the core fetch unit aspect will be covered first.

For direct jumps to be excluded from the predictor history, they must be explicitly recognized during instruction fetch. To achieve this jump detection, use of pre-decode bits is needed, or a “jump bit” for each BTB entry is required. This allows the next fetch address to be set to the corresponding BTB target so that instruction fetch continues from this point the following cycle (since there is no doubt that an unconditional direct jump will always be taken, and will always have the same target address). Inclusion of the jump instruction in the predictor history is not required, as the predictor is not consulted when a direct jump is identified.

In the case where direct jumps are not explicitly recognized by the core fetch unit, the predictor must include jumps in its history. A BTB lookup will convey the fact that the instruction is a control transfer operation that has evaluated taken in the past, but not the type of control operation (conditional branch or direct jump). Therefore the branch direction returned by the predictor is simply associated with the instruction. This implies that direct branches can be mispredicted (assigned a not-taken direction). Over the course of execution, the direct jump will establish a strongly taken behavior that should be reflected in the pattern history. Unfortunately interference, or aliasing, within the predictor could defeat this expected behavior.

The second aspect to deciding whether direct jumps are included in predictor history is how the trace cache is implemented – with or without branch

promotion. For the standard Rotenberg scheme, direct jumps are treated as conditional branch instructions and are included as part of the branch mask and branch flag fields within a trace. When the trace cache is accessed during instruction fetch, both the fetch address and the predictor results are utilized in determining a hit or miss. Specifically, the multiple branch prediction is compared with the branch flags (which indicate direct jumps as well as conditional branches). Therefore the predictor must include jumps as part of the branch history for this reason. If branch promotion is utilized, direct jump instructions can always be considered strongly taken and can automatically (without consulting the branch bias table) be promoted. This alleviates the predictor from storing direct jumps.

To summarize the above, the multiple branch predictor stores history information for direct jumps if: 1) pre-decode bits (or the equivalent) are utilized to identify direct jumps explicitly, and 2) the trace cache utilizes branch promotion. In a machine that does not make use of trace cache, only the first requirement needs to be satisfied.

With the above discussion in mind, four predictor configurations were evaluated: MGAg including direct jumps, MGAg excluding direct jumps, MPAg including direct jumps and MPAg excluding direct jumps. The predictors were assessed in the context of the Revision 1 core fetch unit of Section 2.5. The parameters for PAT and PHT sizes are listed in Table 5.1. These simulations *do not* incorporate trace cache, but rather establish the best multiple branch prediction scheme that will be coupled with the trace cache in later trials. Figure 5.1 shows predictor accuracy (percentage of correctly predicted branches) for each of the configurations.

Accuracy Percentages for MGAg and MPAg Multiple Branch Prediction Schemes

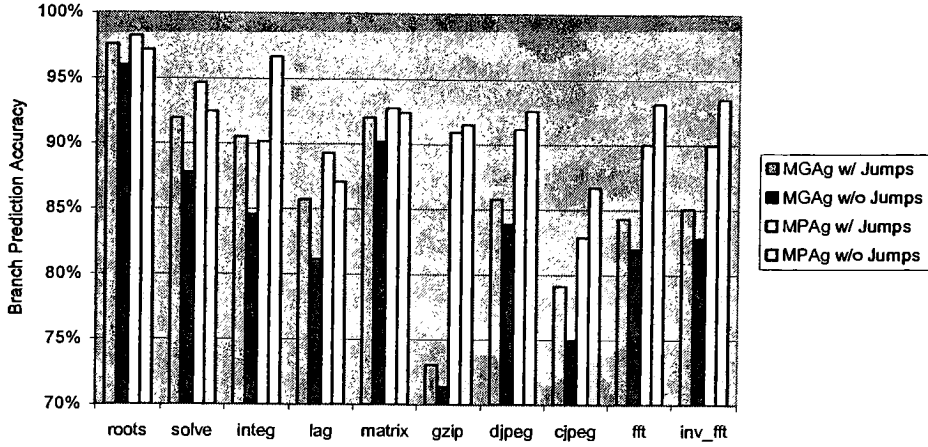


Figure 5.1: Multiple Branch Predictor Accuracy

From this graph it is evident that the MPAg scheme generally outperforms the MGAg predictor. These results are consistent with a similar trend seen for the single branch predictor counterparts. For subsequent simulations, the appropriate MPAg configuration (including or excluding direct jumps) will be used. As a final note concerning the MGAg Predictor: It may not excel from an accuracy standpoint, though the MGAg Predictor's utilization of resources is superior to the MPAg version, as the table size requirements are constant as prediction bandwidth increases. This provides a scalable alternative for larger values of m that may be desirable in implementations that are space constrained.

To conclude this section concerning multiple branch prediction, two graphs are presented that demonstrate an observable trend as the maximum bandwidth value m increases. First, fetch bandwidth improves, as the set of instructions fetched can span increasing numbers of multiple basic blocks. At the same time, prediction accuracy decreases, as "deeper" branch levels are inherently less accurate since they are dependent on the accuracy of the preceding branch. These opposing trends are shown in Figure 5.2 and 5.3. One can expect these

trends to be more pronounced with the introduction of trace cache, as instruction fetch over multiple basic blocks is no longer constrained to not-taken branches (as is the case with the Revision 1 core fetch unit.)

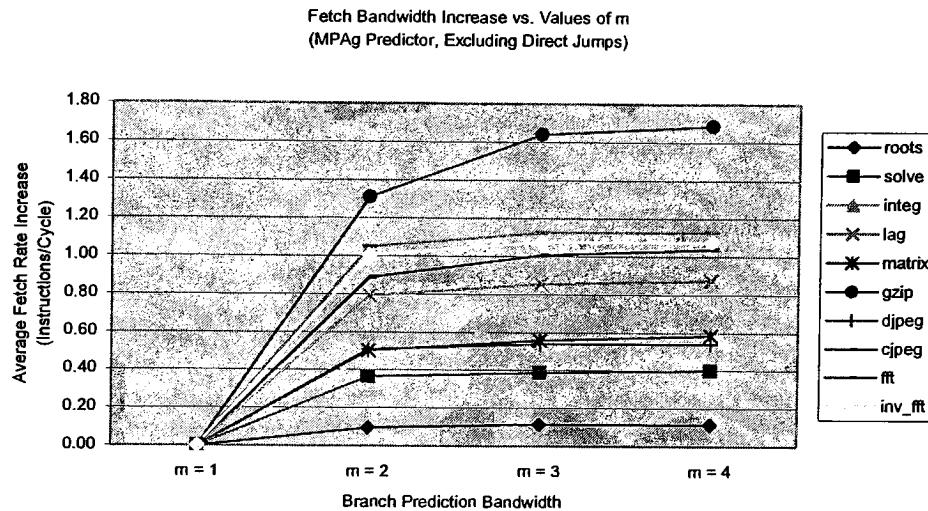


Figure 5.2: Fetch Bandwidth Increase vs. m

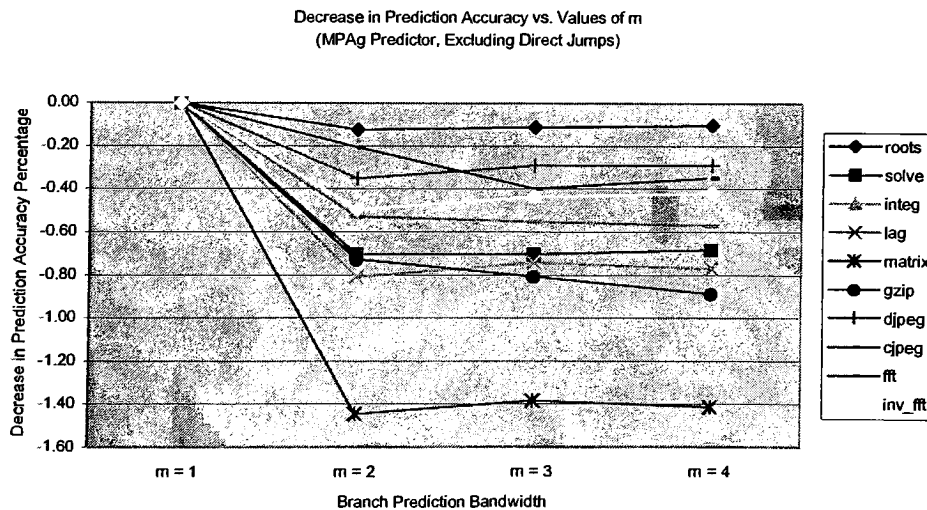


Figure 5.3: Prediction Accuracy vs. m

5.4 Rotenberg Trace Cache

With the multiple branch prediction scheme established, simulation proceeded with an investigation of the basic Rotenberg et al trace cache as proposed in [10] and described in Section 3.2. The trace cache was sized to be roughly equivalent to that of the L1 I-Cache when using a trace segment length of 16 instructions. Three configurations were examined by varying the values of n (maximum number of instructions per trace) and m (maximum number of branches/predictor bandwidth). Values for each are as follows:

- $n = 16, m = 3$
- $n = 20, m = 3$
- $n = 20, m = 4$

These configurations will hereafter be identified as 16/3, 20/3 and 20/4 respectively. For the later two, the decode, issue and retire widths of the pipeline were increased to 32 to accommodate the larger maximum value of n . Several aspects of the trace cache were drawn from the simulations, the first being the average length of segments added (Figure 5.4).

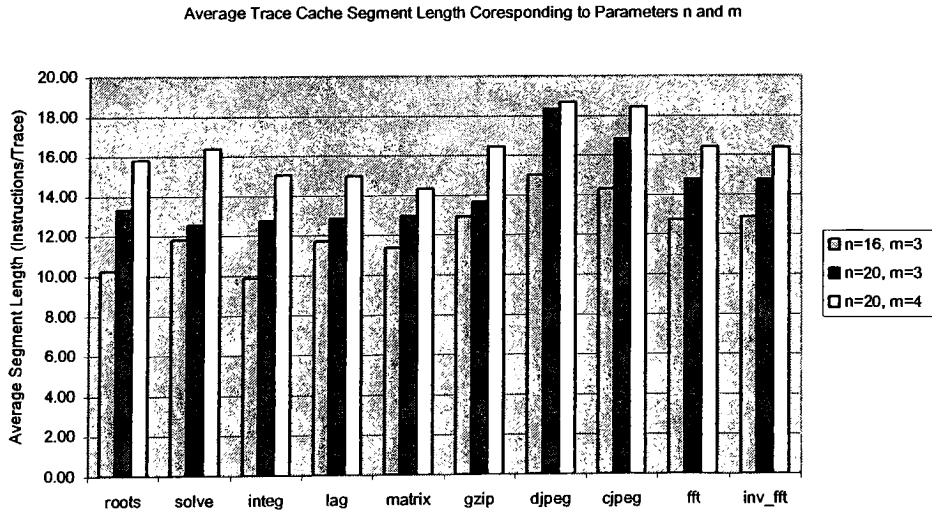


Figure 5.4: Average Trace Cache Segment Length

From the above, we can see that the trace cache is capable of supplying well over 10 instructions per cycle. Figure 5.5 shows the change in the average fetch bandwidth experienced when combining the trace cache with the core fetch unit.

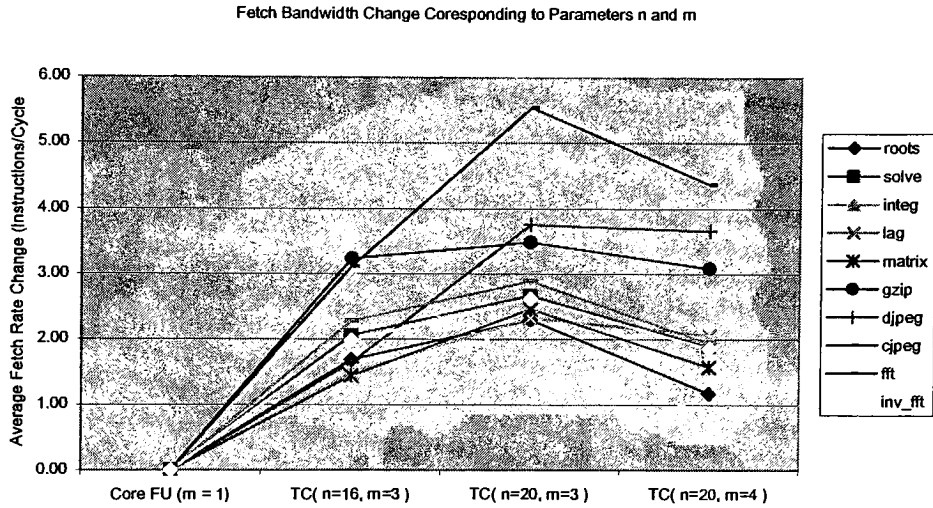


Figure 5.5: Fetch Bandwidth after Incorporating Trace Cache

In general, fetch bandwidth increases with greater segment lengths, though decreases when the prediction bandwidth is changed from three to four branches per cycle. Examining the trace cache hit rate (Figure 5.6) can provide insight into this fetch rate decrease. For a trace cache hit to occur in the Rotenberg scheme, the current fetch address (PC) must match the first instruction (trace ID) of the indexed segment and the branch flags must match the generated prediction. Without storing individual trace target and fall-through addresses for each basic block within a trace, partial matching cannot be implemented; an “all or nothing” situation exists for trace cache hits. When the maximum number of branch instructions able to be included in a trace segment increases, the likelihood of a full match between the branch flags and generated prediction decreases. The result of a reduced trace cache hit rate is that the core fetch mechanism supplies a higher percentage of total instructions at an inherently lower bandwidth.

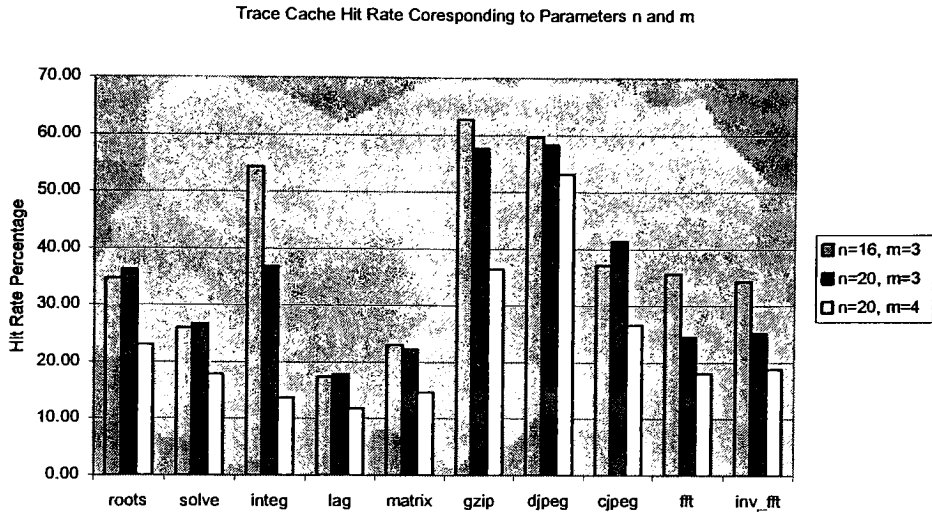


Figure 5.6: Trace Cache Hit Rate

Figure 5.7 shows a speedup comparison (over the base *simoutorder* IPC) of the three configurations tested.

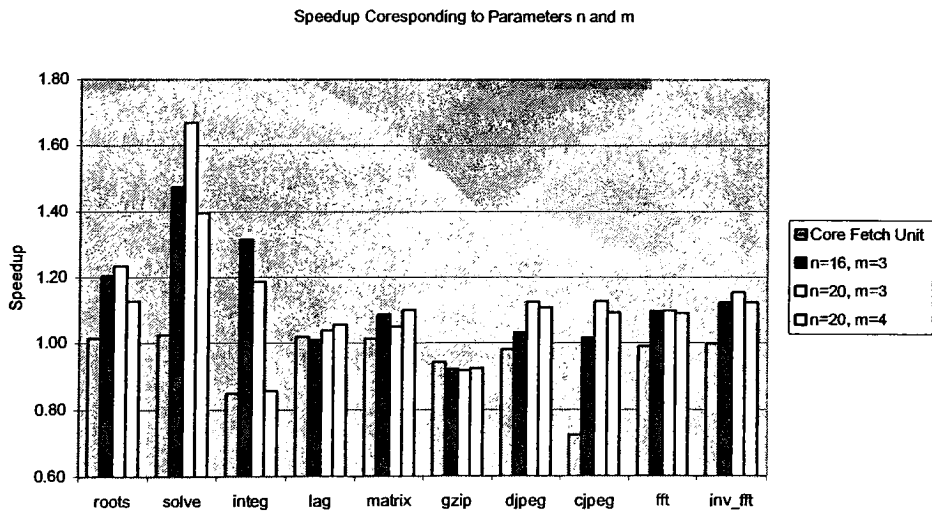


Figure 5.7: Trace Cache Speedup

As expected from the fetch bandwidth results, 20/3 provides the best overall performance in most instances, though requires a 25% increase in trace cache size

over 16/3. To further elaborate on Figure 5.7, it is helpful to look at some statistics involving the **trace-terminating condition**. Three conditions have the potential to finalize a trace in the fill buffer and force a new segment to be written to the trace cache:

- The maximum number of instructions has been reached (n-constrained)
- The maximum number of branches has been reached (m-constrained)
- An indirect jump has been encountered

Figure 5.8 shows the distribution of these three conditions for each of the benchmarks.

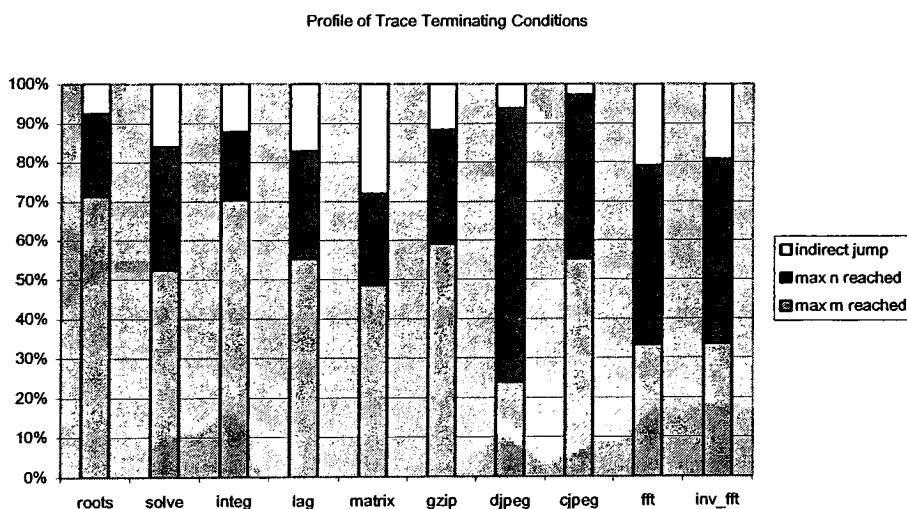


Figure 5.8: Distribution of Trace-Terminating Conditions

Comparing Figure 5.8 with Figure 5.7 shows that programs that exhibit a high percentage of n-constrained traces (such as the two JPEG benchmarks) will benefit from longer trace cache segments.

5.5 Trace Cache Associativity

The Rotenberg scheme implements a direct mapped trace cache. Aside from the usual effects of aliasing, a drawback is that only one trace with a particular Trace

ID can be stored at the same time. Consider a set of dynamic paths that can consist of basic blocks A, B, C, D and E. Traces “ABC” and “ADE” are both frequently encountered during program execution and are equally valid. A directly mapped trace cache would displace “ABC” as soon as “ADE” was finalized and moved from the fill buffer. The next occurrence of “ABC” would similarly displace “ADE”. This trace thrashing can hinder the trace cache hit rate, which will affect the fetch bandwidth. By making the trace cache set associative, multiple paths with the same Trace ID can be stored. This comes at the cost of implementing a set-associative cache structure, which requires additional selection, fill and replace logic, along with greater power consumption.

In the simulations, both a two-way and a four-way set associative trace cache were simulated. The replacement policy used was least recently used (LRU). Figure 5.9 compares the change in hit rate among the different associativities.

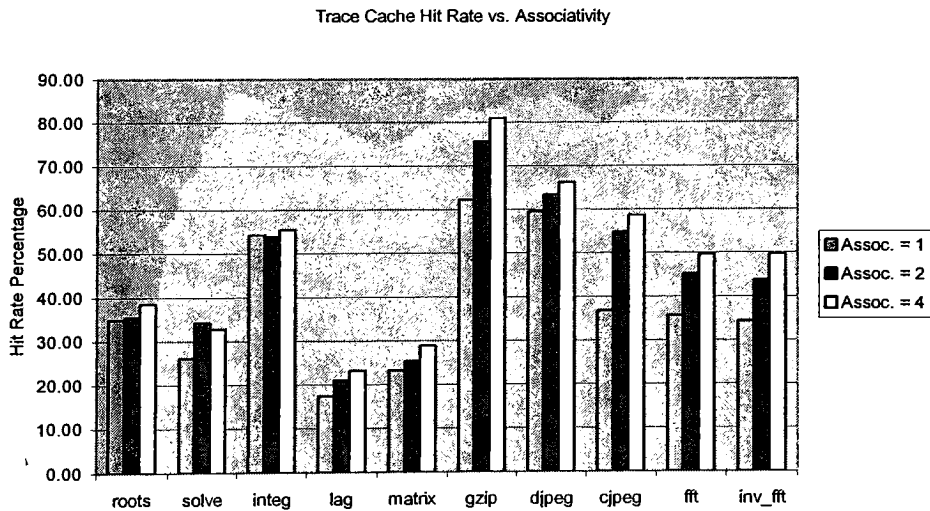


Figure 5.9: Hit Rate vs. Associativity

Trace cache utilization also tends to improve with increases in associativity (Figure 5.10). Utilization is measured by the number of traces that remain invalid at the conclusion of a simulation.

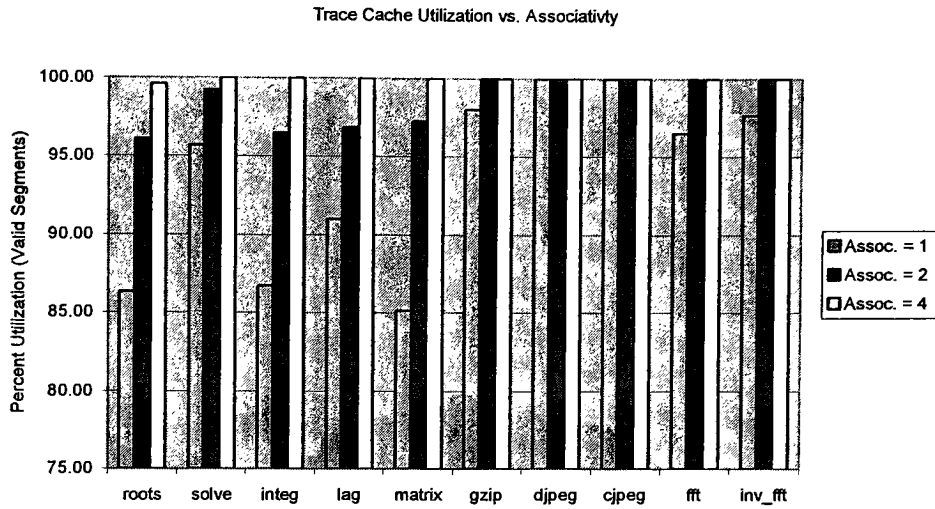


Figure 5.10: Utilization vs. Associativity

The final speedup results using the set associative trace cache organizations is shown in Figure 5.11.

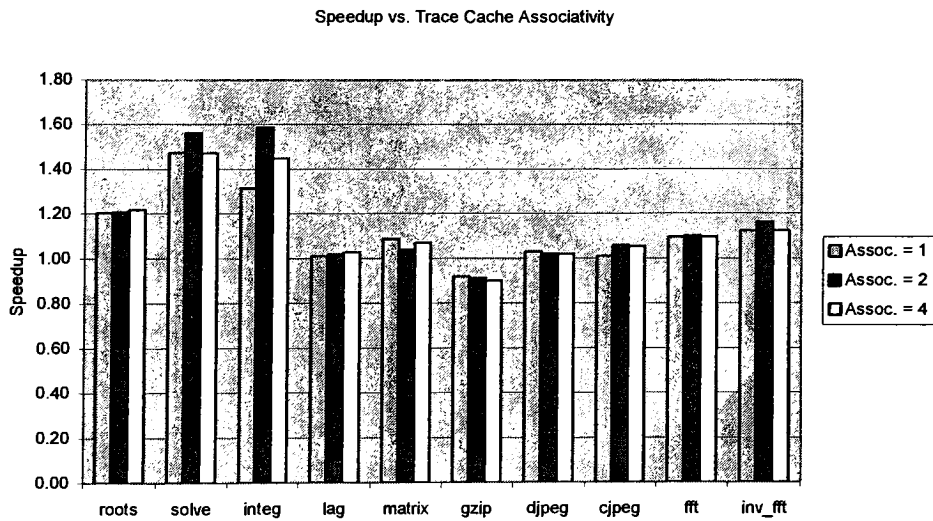


Figure 5.11: Speedup vs. Associativity

This graph presents speedup data that is not as conclusive as one would expect. Results varied, as several prediction accuracies *decreased* along with the trace cache

hit rate and fetch bandwidth increase. In general, it seems that trace cache can benefit from multiple associativity. Though subsequent simulations will utilize the basic direct mapped organization due to the mediocre speedup results of Figure 5.11 coupled with the increased complexity of implementing a set-associative structure in hardware.

5.6 Reassociation

As described in Section 3.11, reassociation [16] is one of the dynamic instruction pre-processing techniques that can be incorporated in a processor that utilizes trace cache. Since the fill unit is outside the critical path of the main pipeline, adding logic to perform these optimizations has a negligible effect on performance. At the same time, such optimizations have the potential to improve both fetch and execution bandwidth.

Instructions that were considered for reassociation in the simulation included ADDI and ADDUI operations that followed each other sequentially (see code example of Section 3.11) or were separated by a direct jump or conditional branch. It was found that the number of occurrences of reassociable instructions within the dynamic instruction stream was minimal. Table 5.4 shows both the number of instructions encountered by the fill unit that were reassociated, as well as the number of reassociated instructions fetched from the trace cache. For all benchmarks, the effect of incorporating reassociation was barely, if at all, reflected in the IPC. Therefore, future simulations did not incorporate the pre-processing technique, as the fill unit overhead does not justify the insignificant performance gains.

Benchmark	Number of Instruction Reassoicated	Number of Reassociated Instructions Fetched
roots	12	7
solve	30	15
integ	13	0
lag	9	0
matrix	0	0
gzip	2	1
djpeg	20	10
cjpeg	8	0
fft	513	155
inv fft	1025	5

Table 5.4: Reassociation Statistics

5.7 Branch Promotion

The trace terminating condition m has the potential to limit traces to a length that falls short of the maximum number of instructions able to be accommodated by a segment. This restricts the fetch bandwidth and results in trace cache storage inefficiency. One technique that aims to alleviate this problem is branch promotion [15]. Discussed in Section 3.4, branch promotion regards branches with strong taken, or not-taken behaviors as statically predicted, leaving the hardware predictor to only consider branches that exhibit reliance on dynamic path history. The branch bias table (BBT) is the structure that records individual bias counts (how many times a branch has been taken or not-taken). Upon reaching a bias threshold, a branch is promoted.

The first simulation set of this section is concerned with establishing an optimal branch bias threshold. The initial bias value was set at 8, and was increased by powers of two for each trial, ending at 256. The next two figures (5.12, 5.13)

shows the resultant fetch bandwidth and speedup (over the base *simoutorder* IPC) corresponding to each of the 6 branch bias thresholds.

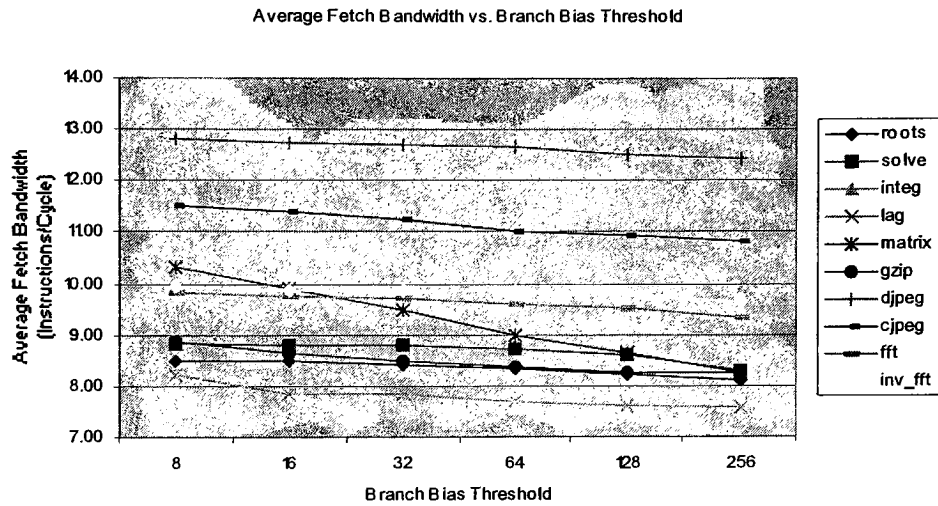


Figure 5.12: Average Fetch Bandwidth vs. Branch Bias Threshold

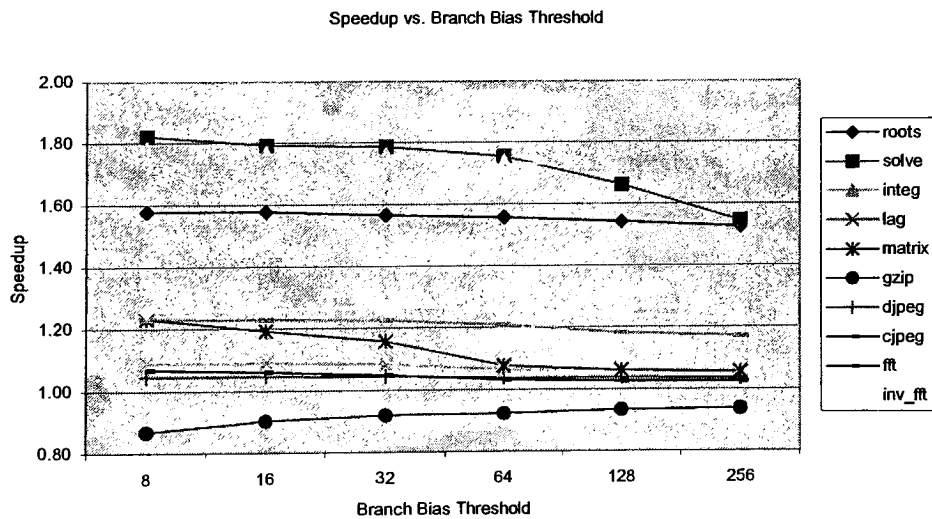


Figure 5.13: Speedup vs. Branch Bias Threshold

As expected, fetch bandwidth increases with more aggressive branch promotion (lower bias thresholds). Setting the bias threshold too low will cause frequent promoted branch mispredictions. This will have the effect of “confusing” the dynamic predictor, as branches are constantly being promoted and demoted, in turn polluting the history tables. Examining Figure 5.13, one can observe that threshold values between 16 and 64 tend to result in the best performance. A branch bias threshold value of 32 was chosen for subsequent simulations.

The second set of simulations related to branch promotion investigates the organization of the branch bias table. As described in [15], the branch bias table is a separate structure in hardware that stores the bias count and the direction of the previous branch outcome. This straightforward implementation of the BBT will be referred to as a **full branch bias table**.

One will note that the BBT is similar to the BTB in that entries are indexed via branch address and utilize tag checking to ensure a hit. This observation introduces an alternate BBT organization: The **Merged BTB/BBT**. By combining the BTB and BBT, hardware space can be conserved. This requires that the merged BTB/BBT provide multiple accesses per cycle, as branch target addresses must be retrieved in the fetch stage, branch bias entries must be supplied to the fill unit, and both BTB and BBT information must be updated when instructions retire in the commit stage. The disadvantage to this organization is that both taken and non-taken branches must be stored. The traditional BTB stores only branches that have previously evaluated taken, as storing not-taken targets is redundant. This is no longer valid with the introduction of the combined BTB/BBT structure, as any particular branch can be strongly biased in either direction and requires a table entry.

The final variation on branch bias table organizations is the **Cost-reduced BBT**. Similar to the merged BTB/BBT, both tables are implemented as one hardware structure. The difference entails reintroduction of the stipulation that

only taken branches are stored. This excludes branches that are strongly not-taken from being promoted, as no branch bias information is recorded for not-taken branches. This drawback is compensated by improved BTB performance, which mirror that provided by a stand-alone BTB.

Figure 5.14 shows the resultant speedups from each of the three branch bias table organizations. As expected, the full BTB performs the best, as both tables are implemented as separate structures. Between the merged BTB/BBT and cost-reduced schemes, it seems that the later provides better performance since BTB hit rates are not sacrificed via redundant storage of not-taken branches.

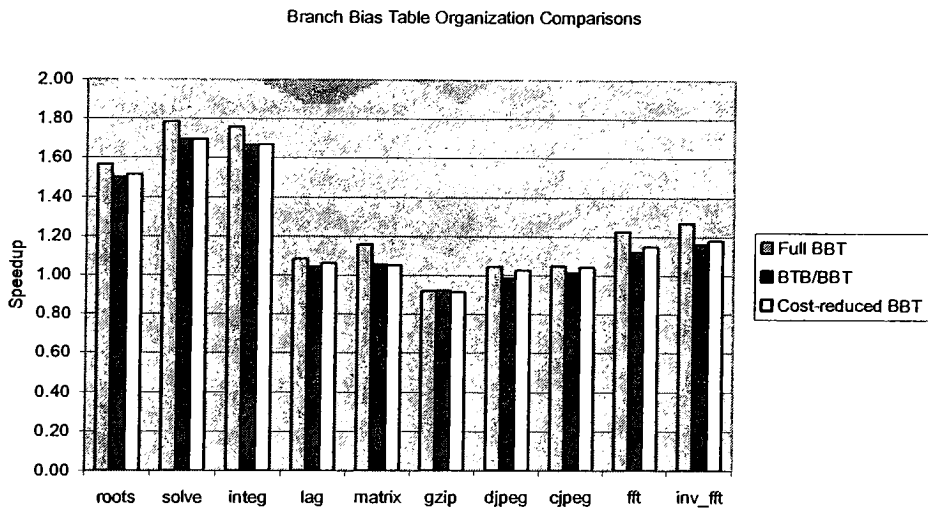


Figure 5.14: Speedup Comparison of BBT Organizations

With perspectives on branch bias threshold and bias table organization, we can now compare the use of branch promotion with the results obtained in Section 5.4 for the Rotenberg trace cache. Figure 5.15 shows the substantial speedup obtained using branch promotion with a full branch bias table and a branch bias threshold value of 32. The first three benchmarks (*roots*, *solve* and *integ*) along with the FFT related computations benefited the most from branch promotion, yielding a maximum speedup of 79%. The average speedup among the ten benchmarks used was 29%.

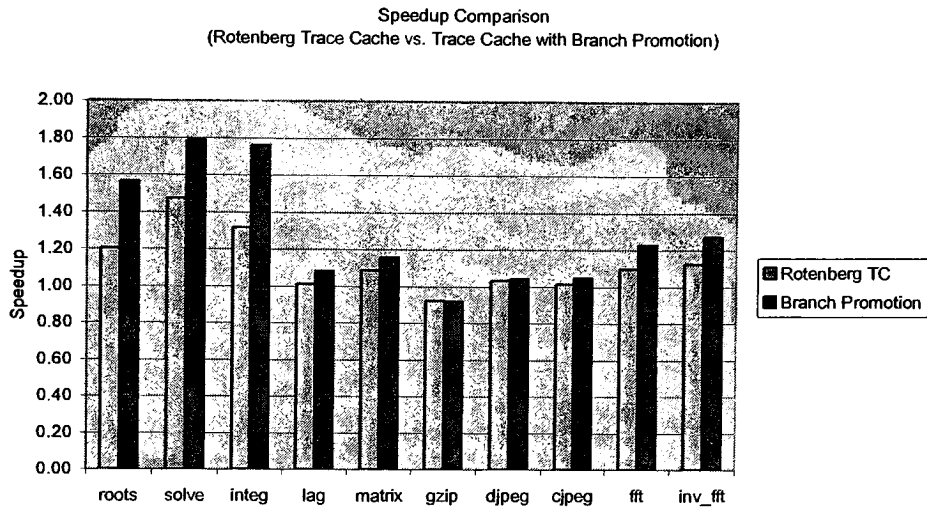


Figure 5.15: Speedup using Branch Promotion

5.8 Alternate Fill Mechanism

The aim of the techniques presented in the next few sections is to improve trace cache hit rates by modifying the way traces are created and filled to the cache. The first of these techniques is the Alternate Fill Scheme that is explained in Section 3.12. This method is best described as a “truncate-and-shift” approach to the fill unit. Instead of completely flushing the fill buffer after a new trace is committed to the trace cache, the front-most basic block is eliminated, and the trace is shifted. Newly retired instructions are added to the end of the partially filled buffer, and the Trace ID is set to that of the instruction following the truncated block.

Figure 5.16 shows the change in trace cache hit rate experienced when the alternate fill scheme is employed. These results are seemingly contrary to the above stated goal: To *improve* the trace cache hit rate. A comparison of the number of unique traces added (Figure 5.17) provides a partial explanation.

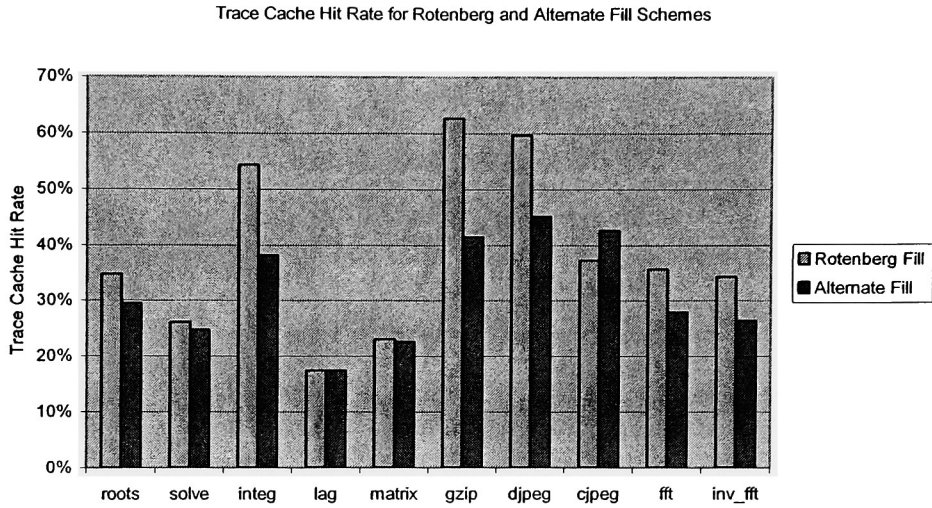


Figure 5.16: Hit Rate for Alternate Fill Scheme

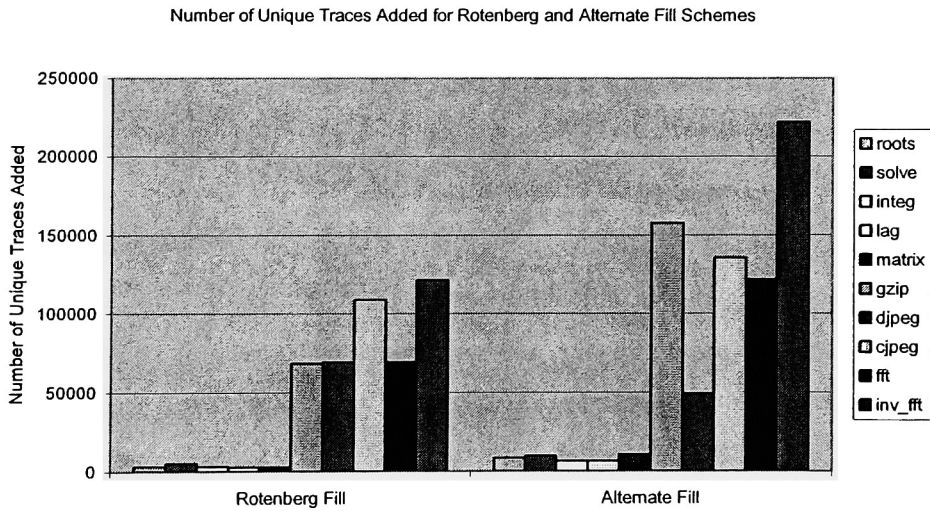


Figure 5.17: Unique Traces Added

Since a unique trace is added for virtually every basic block encountered, the number of traces committed is almost double that of the Rotenberg approach. Despite the reduced hit rate, the performance is largely equivalent to the scheme evaluated in Section 5.4 (Figure 5.18).

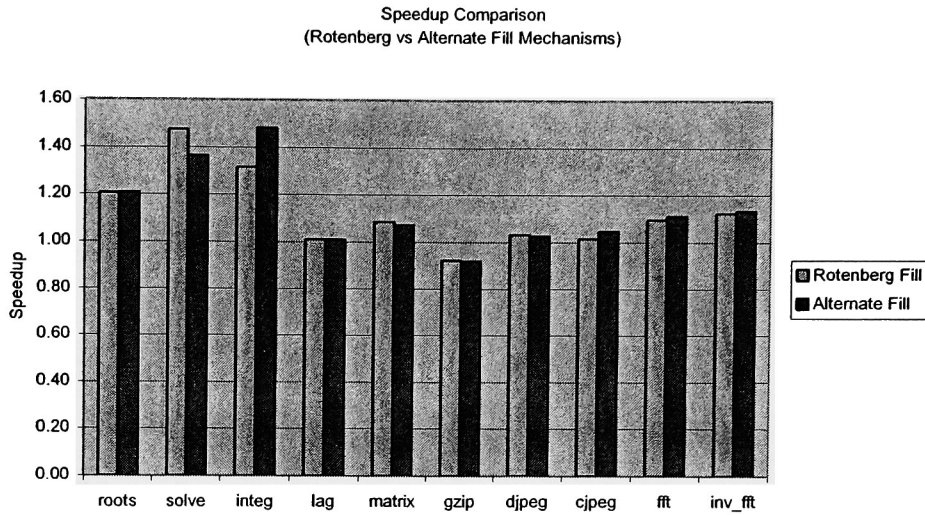


Figure 5.18: Speedup Using Alternate Fill Scheme

5.9 Trace Continuity and Probable Entry Points

This section serves as a segue to the next by explaining two attributes of traces segments and trace cache: **Trace Continuity** and **Probable Entry Points**. The issue of trace continuity is best explained by way of example. Consider a set of basic blocks (A, B, C, D and E) that constitute a dynamic path free of indirect jumps. Each block is sized such that two blocks can occupy the fill buffer without exceeding either the n or m constraint. Three blocks will not fit in the buffer though, as the number of instructions occupying the buffer will reach n before the entire third block can be added. This will result in a finalized trace that ends with a sequential instruction (opposed to a control instruction). If such a trace begins with block A, it will be followed by block B and the *start* of block C (identified as “C1”). Similarly, if the remainder of block C appears independent of C1, it will be identified as “C2”. Figure 5.19 illustrates the fill process for both the Rotenberg and Alternate fill schemes.

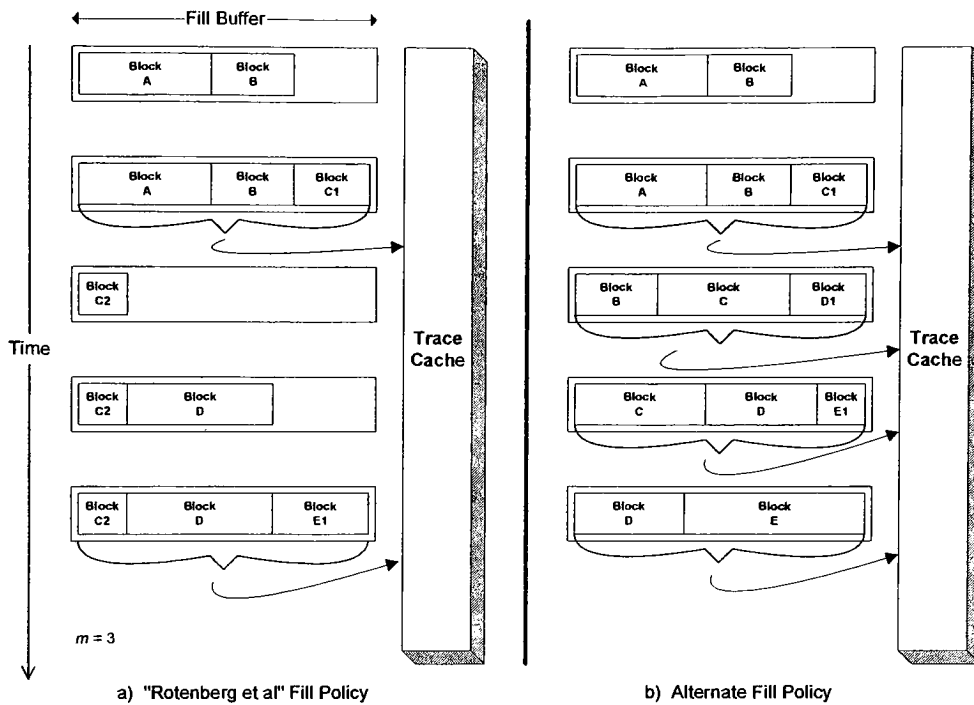


Figure 5.19: Fill Unit Example

Table 5.5 lists the resulting traces generated for each scheme above.

Rotenberg Fill Policy	Alternate Fill Policy
A-B-C1	A-B-C1
C2-D-E1	B-C-D1
	C-D-E1
	D-E

Table 5.5: Resulting Example Traces

The issue of trace continuity becomes evident when the PC returns to the start of Block A. In either scheme, the trace A-B-C1 is fetched from the trace cache and passed down the pipeline. The PC will then be set to the address of C2. The Rotenberg scheme will generate another hit (C2-D-E1). The trace cache implementing the alternate fill scheme will experience a miss, as it does not store

traces beginning with instructions that dynamically follow sequential instructions. The Rotenberg method therefore provides better trace continuity. It should also be clear that trace continuity is applicable only when dealing with *n*-constrained traces. If a trace is terminated by way of the *m*-constraint or indirect jump, the Trace ID of the subsequent trace will start a basic block, which does not pose a problem for either fill approach. Revisiting Table 5.16 and 5.18, we see that benchmarks consisting of traces that are largely *n*-constrained (*djpeg* is a good example; See Figure 5.8) are subject to decreased performance using the alternate fill scheme. This is a result of poor trace continuity.

Despite its deficiency in maintaining trace continuity, the alternate fill scheme does excel at generating traces at probably entry points, or region start points. As discussed in the context of fetch preconstruction in Section 3.7, these points begin regions of code that will be encountered later in the course of normal execution. Probable entry points always start on basic block boundaries; also the manner in which the alternate fill scheme generates traces.

5.10 The Sliding Window Fill Mechanism & Fill Selection Table

The previous section identified two trace properties that the fill unit should take advantage of. The first is to maintain trace continuity when faced with a series of one or more *n*-constrained segments. The second is to identify probable entry points and generate traces based on these fetch addresses. A solution proposed in this work that satisfies this dual requirement involves a new scheme that incorporates a Sliding Window Fill Mechanism (SWFM) and a Fill Selection Table (FST). This section proceeds with a description of the FST followed by a description of the SWFM. A presentation and comparison of results is presented afterwards.

The concept of the fill selection table is fairly straightforward. Every cycle, the PC address that the core fetch mechanism encounters (following a trace cache miss) is identified as a probable entry point. These addresses are stored in the FST. An FST entry consists of an address tag, a valid bit and a counter. Each time a fetch address is encountered by the core fetch unit, the count value of the associated entry is incremented. The fill unit will also allocate or increment the count of a FST entry when an n-constrained trace is constructed and added to the trace cache.

The Sliding Window Fill Mechanism that is paired with the FST is an extension of the alternate fill scheme examined in Section 5.8. The difference is that instead of “truncating-and-shifting” an entire basic block, single instructions are trimmed one at a time. The SWFM is implemented as a circular buffer. Pointers are used to mark the current start of a potential trace segment (*trace_head*) the final instruction of a potential trace segment (*trace_tail*) and the point at which retired instructions are added to the fill buffer (*next_instruction*).

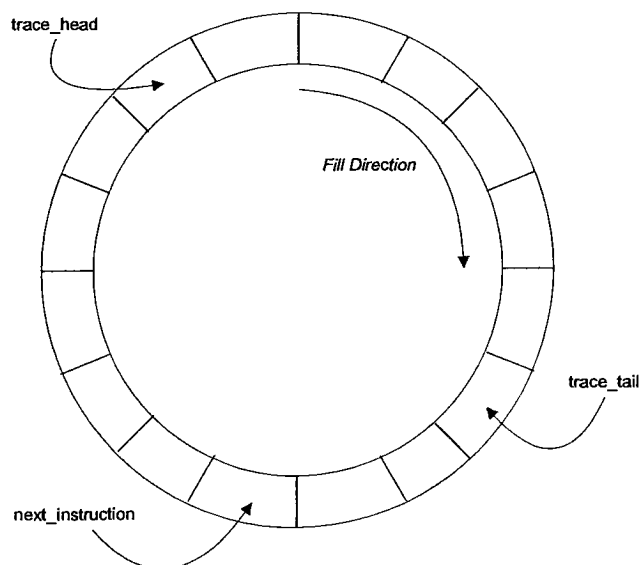


Figure 5.20: The Sliding Window Fill Mechanism

When a retired instruction is added to the fill buffer the *next_instruction* pointer is incremented. At the same time, the potential trace segment bounded by the *trace_head* and *trace_tail* pointers is considered for addition to the trace cache (as described shortly). The *trace_tail* pointer is then adjusted according to the following table.

Fill Buffer Condition	Resultant trace_tail Value
IsControlInstruction(trace_tail) == FALSE	Next-n-instr
(IsConditionalBranch(trace_tail) IsDirectJump(trace_tail)) && (IsConditionalBranch(trace_head) IsDirectJump(trace_head))	Next-m-instr
Trace-tail == trace_head	Next-m-instr
(otherwise)	trace_tail (no change)

Table 5.6: Conditions for Adjusting the *trace_tail* Pointer

Next-n-instr is expressed as the following conditional statement:

```
trace_tail = ( trace_tail - trace_head ) < n - 1 ) ?
              trace_tail + 1 : trace_tail
```

Next-m-instr can be described with the following algorithm:

```
while ( ( trace_tail - trace_head ) < n - 1 ) {
    trace_tail++;

    if ( trace_tail == next_instruction )
        break;

    if ( isControlInstruction( trace_tail ) )
        break;
}
```

The last step in updating the state of the fill buffer is to increment the *trace_head* pointer.

From an implementation standpoint, this may seem to be a difficult task; independently the SWFM would generate as many traces as there are retired

instructions! The value of the SWFM is realized when paired with the FST. Before filling a segment to the trace cache, a FST lookup using the trace ID is performed. If a corresponding entry exists, the count is compared with a defined threshold value. If this count meets the threshold, then the segment is added to the trace cache, and the FST entry is cleared. Otherwise, the state of the fill buffer is updated as described in the previous paragraph, effectively discarding the lead instruction. One final task of the fill buffer is to allocate or increment the count of an FST entry for any address that follows a n-constrained segment that gets added to the trace cache. In summary, the SWFM/FST combination identifies, constructs and fills traces starting with fetch IDs that have been previously encountered or those that immediately follow n-constrained traces.

To observe how the sliding window fill mechanism with FST affects the trace cache hit rate, a set of simulations were performed in which the fill selection threshold value was adjusted. The threshold value of the FST has a substantial effect on the number of unique traces added. Table 5.7 compares the number of traces added for each of the six FST threshold values. Figure 5.21 shows the best trace cache hit rate is obtained using a value of two or three (two was chosen for the remainder of the simulations

Benchmark	T=1	T=2	T=3	T=4	T=8	T=16
roots	25,725	4,828	2,121	1,828	837	327
solve	19,308	2,856	1,579	1,058	444	237
integ	11,480	1,968	1,013	708	303	144
lag	16,269	2,609	1,149	800	360	172
matrix	6,829	1,622	803	567	326	238
gzip	56,020	16,274	11,214	7,784	4,728	2,496
djpeg	327,289	25,265	15,770	11,488	5,297	2,344
cjpeg	303,101	39,720	24,436	18,703	8,666	4,500
fft	334,094	48,395	23,438	17,495	6,313	2,228
inv_fft	653,229	82,096	37590	26,158	10,167	3,068

Table 5.7: Number of Unique Traces Added using SWFM/FST

Trace Cache Hit Rate Using the Sliding Window Fill Scheme and the FST

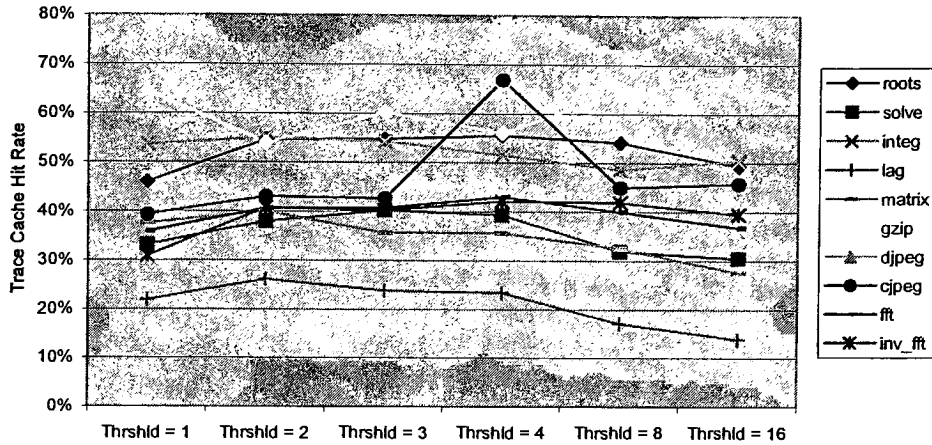


Figure 5.21: Hit Rate for SWFM using various FST Thresholds

Examining Figure 5.22, one can observe that the average speedup value for a processor utilizing trace cache and the SWFM offers notable improvement. Among the benchmarks, the average speedup increase over the Rotenberg fill mechanism was 4% with an average trace cache hit rate increase of 7%.

Speedup Comparison
(Rotenberg vs. Sliding Window Fill Mechanism)

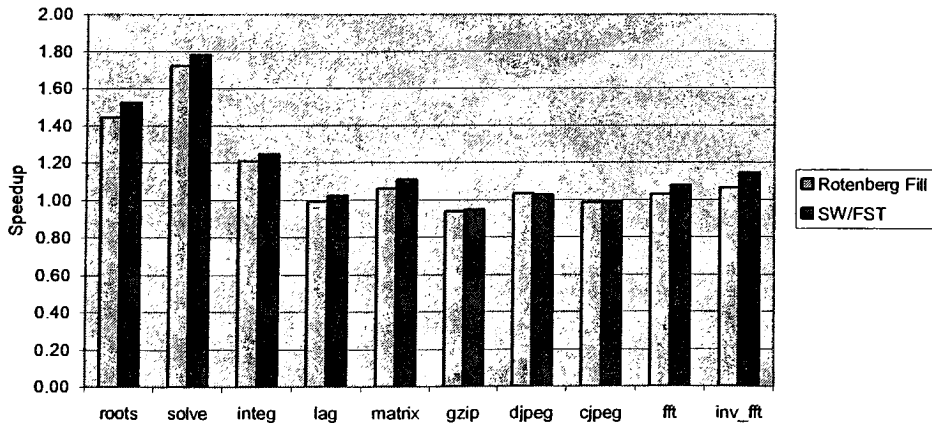


Figure 5.22: Speedup using the SWFM/FST

5.11 Combining Trace Cache Schemes

To conclude the presentation of results, two trace cache schemes that provided favorable results were combined: Branch Promotion and the Sliding Window Fill Mechanism. Intuitively, these schemes seem to compliment each other, as the SWFM excels at generating relevant traces, while branch promotion increases trace segment utilization by reducing the number of traces that are prematurely terminated by the m -constraint. The expectation that the merged scheme would increase trace cache hit rate along with the fetch held true, as figures 5.23 and 5.24 illustrate. For both metrics, the combined scheme outperformed branch promotion and the SWFM when implemented separately. The average hit rate increase over the Rotenberg scheme for this combined scheme was 19%. The fetch bandwidth improved, on average, 17%. over the Rotenberg scheme.

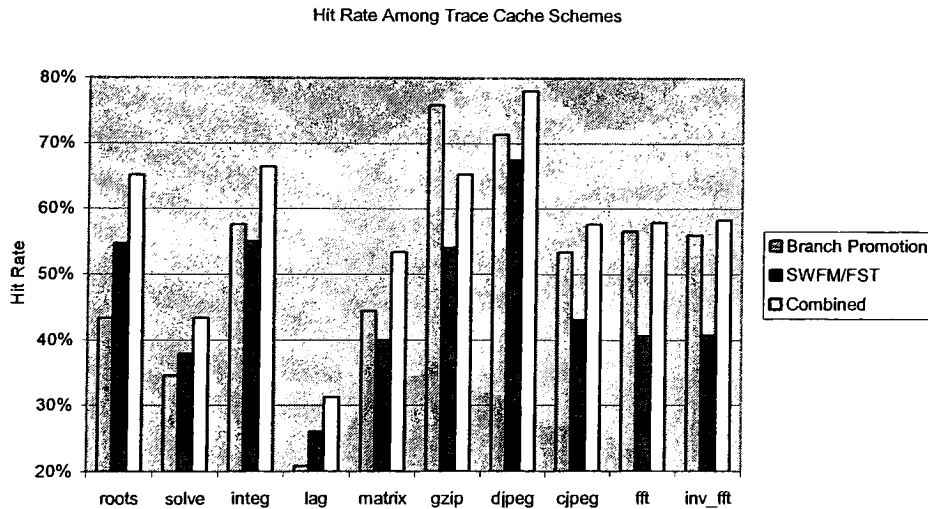


Figure 5.23: Hit Rate Increase for Combined Scheme

Judging from these favorable results, one would expect the speedup to follow a similar trend. The average number of instructions that retired per cycle was hindered though by a reduction in branch prediction accuracy. As Figure 5.25 shows, the prediction rate diminished significantly for several benchmarks with

Fetch Bandwidth Increase Among Trace Cache Schemes

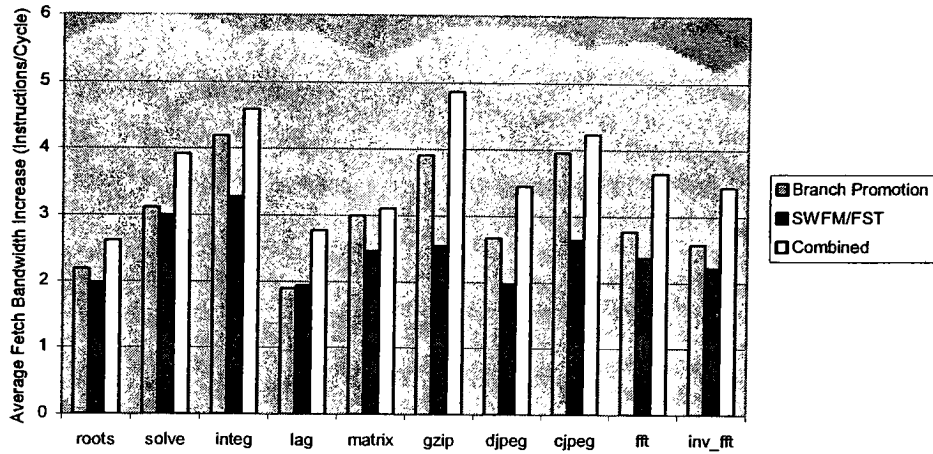


Figure 5.24: Fetch Bandwidth Increase for Combined Scheme

Branch Prediction Accuracy Among Trace Cache Schemes

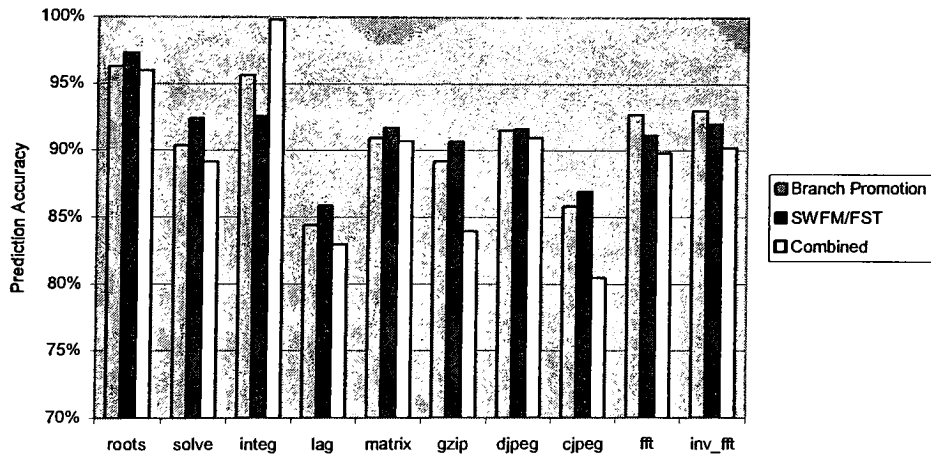


Figure 5.25: Decrease in Branch Prediction Accuracy for Combined Scheme

the dual introduction of branch promotion and the SWFM. This resulted in IPC values that showed no improvement over the Rotenberg trace cache using branch promotion. Were a more aggressive branch predictor utilized to resolve this

decrease in branch prediction, the substantial improvement in fetch bandwidth would be reflected in the overall processor speedup.

5.12 Other Simulated Schemes

A few trace cache schemes were simulated that produced less-than-favorable results. Three are discussed here, along with reasoning that explains the mediocre outcomes. The first of these schemes was an attempt to implement simple fetch address renaming (Section 3.5). This involved storing traces by combining (XOR) the fetch address and the branch flags field to form the trace ID. Trace lookup was accomplished by combining the next fetch address with the m results from the multiple branch predictor. Compared with other schemes investigated, the resultant hit rate was extremely low (under 10%). The justification for this poor hit rate involves the difference in computing the index when traces are stored versus when they are retrieved in the fetch stage. When a trace is committed from the fill buffer, only the branch flags that are valid (i.e. those that correspond to the branch count) are combined with the fetch address to form the trace ID. Since the fetch unit has no information about the number of forthcoming branches in a trace, the trace ID is formed by combining all m prediction bits with the PC. These extraneous bits that were not present when forming the trace tend to throw off the fetch stage when indexing the trace table.

The second scheme implemented that did not perform well was Selective Trace Storage (Section 3.10). This was a simple addition to the fill unit that discarded traces that did not contain taken branches rather than filling them to the trace cache. As described in [18], this scheme is based on the assumption that the core fetch unit is as effective at supplying sequential basic blocks from the L1 I-Cache (fetching beyond non-taken branches) as the trace cache. Hence, storing these traces that duplicate the static program order is redundant. It was found that this

reduced the average trace cache hit rate to below 10%. This exposed some shortcomings in the L1 I-Cache utilized in the simulations. Despite doubling the line size, the cache was not truly interleaved as specified in [18]. This, along with miss penalties associated with the I-Cache resulted in a core fetch unit that *not equal* in performance to the trace cache when fetching beyond not-taken branches. This performance skew was evident in the speedup results.

The last scheme that was investigated but did not perform well when simulated was Path-Based Next Trace Prediction (Section 3.3). This failed the using the simulator for a couple of reasons. First, the scheme as described in [13] states that the “history register is updated speculatively with each new prediction.” This requires that mispredictions be resolved in the commit stage by restoring the state of the history register prior to the prediction and appending it with the hashed trace ID of the correct trace. Considering the trace cache independently, this seems straightforward. Yet when combined with core fetch unit, some difficulties arise. When fetching from the core fetch unit, there is no way to generate a trace ID speculatively, since no trace yet exists generate one. Therefore in this situation, appending the history register must be deferred until the commit stage when a trace has been assembled. In the meantime, instructions may be fetched from the trace cache resulting in speculative updates that will be appended immediately to the history register. A recovery mechanism would seemingly be very complex in this situation. The solution to this problem when implementing path prediction was to defer all updates until the commit stage, which may inherently decrease the accuracy of the scheme. After implementing this modified version of the scheme, the hit rate observed was poor. Further investigation of this method is warranted, especially as an alternative prediction method for trace cache utilizing the SWFM.

5.13 Side Effects Associated with Trace Cache

Aside from the difficulties encountered with the implementation of three schemes as explained in the above section (fetch address renaming, selective trace storage and next-trace prediction), a few other caveats deserve mention. Introduction of the trace cache seems to affect the performance of some of the fetch unit components in a negative fashion. The multiple branch predictor and return address stack are two examples. As described in Section 5.11, the observed hit rate and fetch bandwidth increase were also accompanied by a decrease in multiple branch prediction accuracy. One can claim that the trace cache places more pressure on the multiple branch predictor than the core fetch unit. Since the trace cache has the inherent ability to fetch beyond taken or not-taken conditional branches, a greater number of predictions per cycle are utilized. This implies that more predictions originate closer to the n^{th} level and have higher probabilities of misprediction. The aggregate prediction accuracy is driven down and the performance penalty due to misspeculated paths increases.

The trace cache also has a side effect influencing the RAS. Consider the following situation. A speculative RETURN is fetched (i.e. a preceding branch has not yet evaluated). In the fetch cycle following the RETURN either 1) a hit occurs in the trace cache and the segment is passed down the pipeline, or 2) a miss occurs in the trace cache and the core fetch unit begins fetches the appropriate blocks. The sequence of dynamic instructions is:

```
BRANCH
RETURN
...
...
BRANCH (taken)
...
BRANCH (taken)
...
CALL
...
```

Assume the MBP predicts both branches after the RETURN taken. In the first case, the entire sequence would be fetched over the span of two cycles. The CALL would overwrite the stack location of the speculative RETURN. In the second case it will take four cycles to fetch the same sequence (each predicted taken branch terminates a fetch block). This situation manages to avoid RAS corruption, since the likelihood of the mispredicted branch being detected (in turn flushing the pipeline and adjusting the PC) before the CALL is encountered in the fetch stage is better. This specific example illustrates how the increased fetch bandwidth reduces the RAS accuracy, a trend noticed over the course of simulation.

Chapter 6

Conclusion

Trace Cache has been shown to be an ideal mechanism to sustain the fetch bandwidth required for high issue width processors. The theory, design and results presented as part of this thesis are summarized in this chapter. Some difficulties that were encountered are explained. To conclude, suggestions for future work involving trace cache and the Sliding Window Fill Mechanism are offered.

6.1 Summary of Work

The work performed for this thesis established several results and conclusions based on various trace cache schemes. The first, and most basic, is that trace cache proves to be an optimal low latency, high bandwidth fetch mechanism for wide issue machines. With the exception of one, all benchmarks showed a marked improvement in fetch bandwidth utilizing the methods examined in Sections 5.4 – 5.11. Research which started as an investigation of existing schemes, evolved into a series of fill unit revisions aimed to improve the hit rate of the trace cache. The motivation was to introduce a mechanism that could intelligently produce sequences of traces that started at probable entry points and maintained trace continuity. The product of these successive revisions was the Sliding Window Fill Mechanism, which was tightly coupled with the Fill Select Table. These mechanisms demonstrated an ability to increase the trace cache hit rate by a notable amount. Paired with branch promotion, the hit rate improved by average of 17%, resulting in an average fetch bandwidth increase of 19%.

A note on implementation complexity of the SWFM is warranted. To provide multiple accesses per cycle, the FST can be implemented as an interleaved and/or multi-ported structure. Typically, only a trace per cycle would be committed to the cache. The SWFM has the potential to generate multiple traces per cycle. A queue mechanism would need to be implemented to account for this. Since the fill unit is outside the critical path of the processor, and fill latencies have been shown to have a negligible effect on performance [16] this is acceptable. The complexity of the SWFM is moderate, but compared with the requirements for trace cache in a wide issue machine should prove to be feasible.

6.2 Difficulties Encountered

This project extended over a period of several months, during which a host of challenges were encountered, most of which were overcome. As with any sizable research endeavor, the goals and focus evolved over this period. Comparing the results with the initial proposal provide an indication of this shift in direction. After observing the aspects of the Rotenberg and the Alternate fill mechanism, emphasis was placed on developing a new method that combined the benefits of both. This proved, in the author's opinion, to be of greater value than simply researching existing methods as was proposed initially.

Aside from the difficulties and caveats encountered with a few aspects of the implementation as explained in Section 5.12 and 5.13, a few additional points deserve mention. Understanding the organization of SimpleScalar required overcoming an initial learning curve. The majority of the simulator is coded using a strictly procedural approach, which provides a stark contrast to the understandability and maintainability of object-oriented programming. The drawback to implementing the extension using OO techniques was realized during initial simulation runs involving the trace cache. The class structure of the extension increased simulation run times, in some instances over four times the

time required for the base *simoutorder* simulator. In retrospect, some computation intensive programming techniques (RTTI, use of STL structures) could have been avoided to alleviate this issue.

6.3 Recommendations for Future Work

This thesis established a strong foundation for future projects involving microarchitectures that incorporate high bandwidth fetch mechanisms. Some areas that would be worthwhile to pursue have already been hinted at throughout this thesis. A few include:

- Further investigation of the nature of the inverse relationship between trace cache hit rate/fetch bandwidth and multiple prediction accuracy (Section 5.11).
- Incorporating a multiple branch prediction scheme that yields better prediction accuracy. This would benefit the combined scheme described in Section 5.11. Implementing a path-based prediction scheme is an option that was examined (Section 5.13), though certainly warrants further investigation.
- Improving the core fetch unit to support a truly interleaved I-Cache, thereby allowing a valid assessment of schemes such as Selective Trace Storage.
- Streamlining the trace cache extension to improve simulation run times. This would allow a comprehensive assessment of the SWFM and associated techniques using a complete benchmark suite, such as those provided by SPEC.

In conclusion, this thesis examined trace cache from a variety of perspectives. The simulation and analysis of established schemes led to the proposal of a new

fill unit method that exploits trace continuity and identifies probable start regions to improve trace cache hit rate. Through simulation, it has been shown that this scheme increased hit rates by an average of 7% independently, and by an average of 19% when combined with branch promotion. The combined scheme also yielded a 17% increase in fetch bandwidth. These results demonstrate that the Sliding Window Fill Mechanism can contribute to the overall performance of a processor that utilizes an aggressive trace cache scheme.

References

- [1] D. Sima et al. Advanced Computer Architectures: A Design Space Approach. Addison-Wesley, 1997.
- [2] D. A. Patterson and J. L. Hennessy. Computer Architecture: A Quantitative Approach, 2nd ed. Morgan Kaufmann, 1996.
- [3] J. E. Smith and G. S. Sohi. "The Microarchitecture of Superscalar Processors". *Proceedings of the IEEE*, vol. 83, pp. 1609 – 1624, 1995.
- [4] G. S. Sohi. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers". *IEEE Transactions on Computers*, vol. 39, pp. 349 – 359, 1990.
- [5] T. Yeh and Y. N. Patt. "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History". *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 257 – 266, 1993.
- [6] S. Pan et al. "Correlation-based Branch Prediction". *Twelfth Annual International Phoenix Conference on Computers and Communications*, pp. 210 - 216, 1993.
- [7] S. McFarling. "Combining Branch Predictors". *WRL Technical Note TN-36*, Digital Equipment Corporation, Western Research Laboratory, 1993.
- [8] T. Conte et al. "Optimization of instruction fetch mechanisms for high issue rates". *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-344, 1995.
- [9] T. Yeh et al. "Increasing Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache". *Proceedings of the 7th ACM International Conference on Supercomputing*, pp. 67 – 76, 1993.
- [10] E. Rotenberg et al. "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching". *Proceedings of the 29th Annual IEEE / ACM International Symposium on Microarchitecture*, pp. 24 – 34, 1996.

- [11] D. Burger and T. M. Austin. "The SimpleScalar Toolset, Version 2.0". www.simplescalar.com
- [12] T. M. Austin. "SimpleScalar Hacker's Guide". www.simplescalar.com
- [13] Q. Jacobson et al. "Path- Based Next Trace Prediction". *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14 –23, 1997.
- [14] D. H. Friendly et al. "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism". *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 24 –33, 1997.
- [15] S. J. Patel et al. "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing". *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 262 - 271, 1998.
- [16] D. H. Friendly et al. "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors". *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 173 -181, 1998.
- [17] B. Black et al. "The Block-based Trace Cache". *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 196 -207, 1999.
- [18] A. Ramirez et al. "Trace Cache Redundancy: Red & Blue Traces". *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pp. 325 –333, 2000.
- [19] Q. Jacobson and J. E. Smith. "Trace Preconstruction". *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 36 – 46, 2000.
- [20] A. Peleg and U. Weiser. "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line". US Patent number 5,381,533, Intel Corporation, 1994.
- [21] P. W. Lee et al. "Exploring the Trace Cache Design Space". Graduate Project, Stanford University, 2000.
- [22] S. J. Patel et al. "Evaluation of Design Options for the Trace Cache Fetch Mechanism". *IEEE Transactions on Computers*, vol. 48, pp 193 – 204, 1999.
- [23] Y. Xie and M. Wang. "Multi-Branch Prediction". Graduate Project, Carnegie Mellon University. 1999.

- [24] E. Rotenberg et al. "Trace Processors". *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 111 – 120, 1997.
- [25] Q. Jacobson and J. E. Smith. "Instruction Pre-Processing in Trace Processors". *Proceedings of the Fifth IEEE International Symposium on High Performance Computer Architecture*, pp. 125 - 129, 1999.
- [26] Y. Chou and J. P. Shen. "Instruction Path Coprocessors". *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 270 – 281, 2000.
- [27] Y. Chou et al. "PipeRench Implementation of the Instruction Path Coprocessor". *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 147 – 158, 2000.
- [28] S. J. Patel and S. S. Lumemtta. "rePLay: A Hardware Framework for Dynamic Program Optimization". *Technical Report CRHC-99-16, University of Illinois Technical Report*, 1999.
- [29] G. Hinton et al. "The Microarchitecture of the Pentium 4". *Intel Technology Journal*, Q1, 2001.
- [30] C. Lee et al. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems". Presented at Micro 30, 1997.
- [31] M. R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". *IEEE 4th Annual Workshop on Workload Characterization*. 2001.
- [32] D. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0" *University of Wisconsin-Madison Computer Sciences Department Technical Report # 1342*, 1997.
- [33] T. M. Austin. "SimpleScalar Hacker's Guide". www.simplescalar.com.
- [34] E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Appendix A

Trace Cache Extension Header Files

**** Abstract Interfaces ****

Main/Creational Classes

TraceCache
TraceCacheFactory
TraceCacheConnector
CoreFetchUnit

Component Classes

HistoryUnit
Renamer
TraceIDSelector
TraceGenerator
FillUnit
Predictor

Data Classes

HistoryToken
PredictionToken
Trace
Instruction
Address
TraceCacheData

Helper/Exception Classes

TCDefs (#defines)

TCLogger
Outputable
Printable

LookupTraceException
InstructionNotAddedException
InvalidOptionException
TraceCacheException

```
// File:      $Id: TraceCache.h,v 1.19 2002/10/28 19:00:51 ted Exp $
// Author:    Edward Mulrane
// Contributors:
// Description: Top-level trace cache class. Encompasses all the various
//              components.
// Revisions:
//   $Log: TraceCache.h,v $
//   Revision 1.19 2002/10/28 19:00:51 ted
//   *** empty log message ***
//   Revision 1.18 2002/10/14 22:45:30 ted
//   before Instruction::isTBOp() -> Instruction::isControl() change
//   Revision 1.17 2002/08/31 02:14:31 ted
//   Further updates/fixes...
//   Revision 1.16 2002/08/12 16:37:53 ted
//   Path-based related changes...
//   Revision 1.15 2002/08/05 16:40:49 ted
//   Added getPredictionTokens method that simply calls
//   TraceGenerator::getPredictionTokens
//   Revision 1.14 2002/07/16 16:21:48 ted
//   Further updates...
//   Revision 1.13 2002/07/09 00:55:56 ted
//   Added setLogger and associated attribute
//   Revision 1.12 2002/07/08 18:03:32 ted
//   Killed some debug stuff, other minor changes...
//   Revision 1.11 2002/07/03 13:31:23 ted
//   Debug...
//   Revision 1.10 2002/06/30 01:39:00 ted
//   Bug fixes...
//   Revision 1.9 2002/06/27 01:54:16 ted
//   Various minor changes/bug fixes...
//   Revision 1.8 2002/06/25 15:20:50 ted
//   Added dumpOptions and dumpStats
//   Revision 1.7 2002/06/23 23:36:11 ted
//   Added TraceCacheConnector::setTraceCache call to constructor
//   Revision 1.6 2002/06/21 15:45:27 ted
//   Finished first draft of fetchTraceSegment
//   Revision 1.5 2002/06/18 15:34:12 ted
//   Added TC component members and Core Fetch Unit pointer
//   Revision 1.4 2002/05/30 19:52:25 ted
//   Added TraceCacheConnector as a friend class
//   Revision 1.3 2002/05/30 17:39:37 ted
//   Added TraceCacheConnector pointer
//   Revision 1.2 2002/04/26 03:15:28 ted
//   modified interface such that the factory is provided
```

```
// upon construction.
// TraceCache manages factory destruction.
//
// Revision 1.1 2002/04/12 00:44:21 ted
// Initial revision
//
//
// #ifndef _TRACECACHE_H
// #define _TRACECACHE_H
//
// #include "Outputable.h"
//
// #include "TraceCacheFactory.h"
// #include "TraceCacheConnector.h"
//
// #include "CoreFetchUnit.h"
//
// #include "HistoryUnit.h"
// #include "Renamer.h"
// #include "TraceIDSelector.h"
// #include "TraceGenerator.h"
// #include "FillUnit.h"
//
// #include "TCLogger.h"
//
// #include "../machine.h" //Defines md_addr_t
// #include "../sim-tc.h" //Defines fetch_rec
//
// class TraceCache : public Outputable {
//
// friend class TraceCacheConnector;
//
// public:
//
//     // Name: (constructor)
//     // Description: Creates the trace cache and establishes the various
//     //              components based on the concrete factory passed in.
//     // Arguments: A reference to the concrete factory, the connector,
//     //              core fetch unit and the TC trace output logger.
//     // Modifies: Creates the object, and Sets the TraceCache pointer of
//     //              the connector to 'this'
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     TraceCache( TraceCacheFactory * factory,
//                 TraceCacheConnector * connector,
//                 CoreFetchUnit * fu,
//                 TCLogger * tc_log );
//
//     // Name: (destructor)
//     //
//     // Description: Destroys the trace cache and associated trace cache
//     //              factory/connector...
//     // Arguments: (none)
```

```

// Modifies: Destroys the trace cache factory & connector provided
// Returns: to the constructor.
// Arguments: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual ~TraceCache();

// Name: fetchTraceSegment
// Description: Populates the IPFETCH -> DISPATCH queue if there is a
// hit in the trace cache. Called from ruw_fetch.
// Arguments: Lots!!
// Modifies: (none)
// Returns: A boolean value indicatinig a trace cache hit or miss.
// Assertions: (none)
// Exceptions: (none)
//
bool fetchTraceSegment( md_addr_t &
                       fetch_pred_pc,
                       fetch_data,
                       fetch_tail,
                       fetch_num,
                       unsigned &
                       ptrace_seq,

                       const int &
                       const int &
                       const int &
                       ruw_decode_width,
                       fetch_speed,
                       const int &
                       ruw_ifq_size );

// Name: retireInstruction
// Description: Performs back-end trace cache work (line fill,
// instruction stream processing, etc.)
// Arguments: The retired instruction
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void retireInstruction( Instruction & instr );

// Name: drainFillUnit
// Description: Explicitly ends any trace currently being appended too
// in the fill unit (called when a trap/system call
// is encountered during the decode stage)
// Arguments: (none)
// Modifies: Explicitly ends the trace being formed
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void drainFillUnit();

// Name: getBranchCount
// Description: Returns the number of branches encountered in the last
// fetch (for the purpose of deallocating unused

```

```

// Arguments: prediction tokens
// Modifies: (none)
// Returns: (none)
// Assertions: The number of branches in the last fetch group
// Exceptions: (none)
//
int getBranchCount() const;

// Name: getPredictionTokens
// Description: Returns the prediction token vector associated with
// the last fetch
// Arguments: (none)
// Modifies: (none)
// Returns: The token vector for the last tc fetch
// Assertions: (none)
// Exceptions: (none)
//
TokenVector & getPredictionTokens();

// Name: dumpOptions
// Description: Writes options specific information to the output
// stream
// Arguments: The output stream for writing
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void dumpOptions( ostream & out );

// Name: dumpStats
// Description: Writes statistic specific information to the output
// stream
// Arguments: The output stream for writing
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void dumpStats( ostream & out );

private:
TraceCacheFactory * _factory; // The associated TC Factory
TraceCacheConnector * _connector; // The associated TC Connector

CoreFetchUnit * _fu; // The core fetch unit

HistoryUnit * _history_unit; // Components...
RenamerVector * _renamers;
TraceIDSelector * _trace_id_selector;
TraceGenerator * _trace_generator;
FillUnit * _fill_unit;

```

```
TCLogger *   _tc_log;           // TC trace output logger
int          _bcount;           // Last branch count
//
// Statistics
//
long _trace_too_big;

long _wrong_starting_pc;

long _branches_fetched;
long _promoted_fetched;
long _reassoc_fetched;

); // TraceCache
#endif
```

```

// File: $Id: TraceCacheFactory.h,v 1.9 2002/06/24 22:06:11 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Abstract factory providing the interface for inherited
//               classes producing the components for differing trace cache
//               models.
// Revisions:
//   $Log: TraceCacheFactory.h,v $
//   Revision 1.9 2002/06/24 22:06:11 ted
//   Modified while adding/debugging options parsing/passing
//
//   Revision 1.8 2002/06/18 23:34:05 ted
//   Added virtual destructor...
//
//   Revision 1.7 2002/06/17 22:42:22 ted
//   Changed all occurrences of InputGenerator to HistoryUnit
//
//   Revision 1.6 2002/06/17 22:24:25 ted
//   Added TraceGenerator and FillUnit to factory
//
//   Revision 1.5 2002/05/30 17:39:37 ted
//   Changed references returned by 'create' members to pointers
//
//   Revision 1.4 2002/05/29 20:09:10 ted
//   *** empty log message ***
//
//   Revision 1.3 2002/04/26 03:15:28 ted
//   added parseOptions
//
//   Revision 1.2 2002/04/12 05:12:53 ted
//   *** empty log message ***
//
//   Revision 1.1 2002/04/12 00:44:21 ted
//   Initial revision
//
//
// #ifndef _TRACECACHEFACTORY_H
// #define _TRACECACHEFACTORY_H
//
// #include <string>
// #include <map>
//
// #include "HistoryUnit.h"
// #include "Renamer.h"
// #include "TraceIDSelector.h"
// #include "TraceGenerator.h"
// #include "FillUnit.h"
// #include "InvalidOptionException.h"
//
// typedef map< string, string, less< string > > OptionsMap;
//
// class TraceCacheFactory {
// public:
//
//     // Name: (destructor)
//     // Description: Provides a virtual base class destructor for proper
//     //               destruction of derived classes when using polymorphism
//     // Arguments: (none)
//
//     // Modifies: Destroys/deallocates the object
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     virtual ~TraceCacheFactory() { }
//
//     // Name: parseOptions (pure virtual)
//     // Description: Parses the option string passed in from sim-tc
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     virtual void parseOptions( const char * options )
//         throw ( InvalidOptionException ) = 0;
//
//     // Name: initializeFactory (pure virtual)
//     // Description: Performs any inits. prior to component creations.
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     virtual void initializeFactory()
//         throw ( InvalidOptionException ) = 0;
//
//     // Name: createHistoryUnit (pure virtual)
//     // Description: Returns a pointer to the History Unit
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: A pointer to the History Unit
//     // Assertions: (none)
//     // Exceptions: (none)
//     virtual HistoryUnit * createHistoryUnit()
//         throw ( InvalidOptionException ) = 0;
//
//     // Name: createRenamers (pure virtual)
//     // Description: Returns a pointer to the Renamer Array.
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: A pointer to the Renamer Array.
//     // Assertions: (none)
//     // Exceptions: (none)
//     virtual RenamerVector * createRenamers()
//         throw ( InvalidOptionException ) = 0;
//
//     // Name: createTraceIDSelector (pure virtual)

```

```
//
// Description: Returns a pointer to the Trace ID Selector
// Arguments: (none)
// Modifies: (none)
// Returns: A pointer to the Trace ID Selector
// Assertions: (none)
// Exceptions: (none)
//
virtual TraceIDSelector * createTraceIDSelector()
    throw ( InvalidOptionException ) = 0;

//
// Name: createTraceGenerator (pure virtual)
//
// Description: Returns a pointer to the Trace Generator
// Arguments: (none)
// Modifies: (none)
// Returns: A pointer to the Trace ID Generator
// Assertions: (none)
// Exceptions: (none)
//
virtual TraceGenerator * createTraceGenerator()
    throw ( InvalidOptionException ) = 0;

//
// Name: createFillUnit (pure virtual)
//
// Description: Returns a pointer to the Trace Fill Unit
// Arguments: (none)
// Modifies: (none)
// Returns: A pointer to the Trace Fill Unit
// Assertions: (none)
// Exceptions: (none)
//
virtual FillUnit * createFillUnit()
    throw ( InvalidOptionException ) = 0;

protected:
    OptionsMap _options;
}; // TraceCacheFactory

#endif
```

```

// File: $Id: TraceCacheConnector.h,v 1.5 2002/07/23 16:57:11 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Friend class of TraceCache whos subclasses will handle
//              establishing the interconnection between components.
// Revisions:
// $Log: TraceCacheConnector.h,v $
// Revision 1.5 2002/07/23 16:57:11 ted
// Modified creation scheme such that no 'set' methods
// exist in the base component classes.
//
// Revision 1.4 2002/06/24 22:06:11 ted
// Modified while adding/debugging options parsing/passing
//
// Revision 1.3 2002/06/18 15:34:12 ted
// Added protected accessor members for the Trace Cache
// data members for use by concrete children
//
// Revision 1.2 2002/05/30 18:00:38 ted
// The real initial revision
//
// Revision 1.1 2002/05/30 17:39:37 ted
// Initial revision
//
//
// #ifndef TRACECACHECONNECTOR_H
// #define TRACECACHECONNECTOR_H
//
// #include "CoreFetchUnit.h"
//
// #include "HistoryUnit.h"
// #include "Renamer.h"
// #include "TraceIDSelector.h"
// #include "TraceGenerator.h"
// #include "FillUnit.h"
//
// class TraceCache;
//
// class TraceCacheConnector {
// public:
//
//     // Name: (constructor)
//
//     // Description: Initializes the TraceCache pointer to null.
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//
//     TraceCacheConnector();
//
//     // Name: setTraceCache
//
//     // Description: Sets the TraceCache pointer to the TC whos components
//     //              require connections.
//     // Arguments: The TraceCache pointer
//     // Modifies: (none)

```

```

// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
void setTraceCache( TraceCache * tc );

//
// Name: checkComponents (pure virtual)
//
// Description: Performs pre-condition checks to verify consistency
//              between the Trace Cache and the rest of the system
// Arguments: (none)
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// virtual void checkComponents() = 0;

//
// Name: connectComponents (pure virtual)
//
// Description: Connect the components of the TraceCache depending
//              on the concrete implementation.
// Arguments: (none)
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// virtual void connectComponents() = 0;

protected:
//
// These helper accessors provide concrete connectors
// access to the TraceCache data members
//
// CoreFetchUnit * getCoreFetchUnit();

private:
//
// TraceCache * _tc;
//
// TraceCacheConnector
//
#endif

```

```

// File: Std: CoreFetchUnit.h, v 1.15 2002/10/28 19:00:51 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: The core fetch unit, as described in Rotenberg et al. The
//              branch predictor is "owned" by this unit.
// Revisions:
// $Log: CoreFetchUnit.h,v $
// Revision 1.15 2002/10/28 19:00:51 ted
// *** empty log message ***
//
// Revision 1.14 2002/10/16 16:33:27 ted
// Fixed branch promotion mechanism...
//
// Revision 1.13 2002/10/14 22:43:50 ted
// before Instruction::isBtOp() -> Instruction::isControl() change
//
// Revision 1.12 2002/08/31 02:14:31 ted
// Further updates/fixes...
//
// Revision 1.11 2002/07/16 16:21:48 ted
// Further updates...
//
// Revision 1.10 2002/07/09 00:55:56 ted
// Added setLogger and associated attribute
//
// Revision 1.9 2002/07/08 18:03:32 ted
// Added TC Trace output stream attribute
//
// Revision 1.8 2002/06/28 17:16:15 ted
// Debugging updates...
//
// Revision 1.7 2002/06/27 01:54:16 ted
// Various minor changes/bug fixes...
//
// Revision 1.6 2002/06/25 15:20:50 ted
// Added dumpOptions and dumpStats
//
// Revision 1.5 2002/06/24 17:14:41 ted
// Removed static members and added btb_size parameter to constructor
//
// Revision 1.4 2002/06/21 15:45:27 ted
// Misc changes to accompany work done with TraceCache
//
// Revision 1.3 2002/06/20 16:50:25 ted
// Continued work in progress...
//
// Revision 1.2 2002/06/19 17:12:02 ted
// Work in progress...
//
// Revision 1.1 2002/06/18 15:34:12 ted
// Initial revision
//
//
// #ifndef _COREFETCHUNIT_H
// #define _COREFETCHUNIT_H
//
// #include <map>
// #include "Outputable.h"
// #include "Predictor.h"
// #include "Instruction.h"

```

```

#include "TCLogger.h"

#include ".../machine.h"
#include ".../sim-tc.h"
#include ".../cache.h"
#include ".../ptrace.h"

class CoreFetchUnit : public Outputable {
public:
    //
    // Name: (constructor)
    //
    // Description: Creates and initializes the core fetch unit with the
    //              predictor given
    // Arguments: Pointers to the multiple and single branch predictors
    //              and the TC trace logger stream
    // Modifies: (none)
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    CoreFetchUnit( Predictor * mpred, bpred_t * bpred, TCLogger * tc_log,
                  BiasScheme bias_scheme, int btb_size, int bt_assoc,
                  int bias_threshold );

    //
    // Name: (destructor)
    //
    // Description: Destroys the fetch unit AND the associated
    //              multiple branch predictor
    // Arguments: (none)
    // Modifies: (none)
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    virtual ~CoreFetchUnit();

    //
    // Name: getPredictor
    //
    // Description: Returns the multiple branch predictor
    // Arguments: (none)
    // Modifies: (none)
    // Returns: The multiple branch predictor
    // Assertions: (none)
    // Exceptions: (none)
    //
    Predictor * getPredictor();

    //
    // Name: fetchInstructions
    //
    // Description: Populates the IFETCH -> DISPATCH queue via the core
    //              fetch mechanism after a miss in the trace cache.
    //              Called from ruu_fetch.
    // Arguments: The references to the current PC, the IFETCH ->
    //              DISPATCH queue, and the head, tail and size of the

```



```

// queue.
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void fetchInstructions( md_addr_t &
    mem_t *
    cache_t *
    cache_t *
    unsigned &
    fetch_rec *
    int &
    int &
    int &
    int &
    int &
    int &
    unsigned &
    const int &
    const int &
    const int &
    const md_addr_t & ld_text_base,
    const unsigned & ld_text_size,
    const int & cache_ill_iat,
    const tick_t & sim_cycle );

// Name: getBranchTarget
//
// Description: Accesses the SimpleScalar BTB along with
// accessing/updating the RAS
// Arguments: The instruction, dir_update_ptr and stack_recover_idx
// Modifies: (none)
// Returns: A valid branch target
// Assertions: (none)
// Exceptions: (none)
//
Address getBranchTarget( Instruction & inst,
    bpred_update_t * dir_update_ptr,
    int * stack_recover_idx );

// Name: retireInstruction
//
// Description: Performs back-end core fetch unit work (updating the
// multiple branch predictor )
// Arguments: The retired instruction
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void retireInstruction( Instruction & instr );

// Name: getBranchCount
//
// Description: Returns the number of branches encountered in the last
// fetch (for the purpose of deallocating unused
// prediction tokens)
//
// Arguments: (none)
// Modifies: (none)
// Returns: The number of branches in the last fetch group
// Assertions: (none)
// Exceptions: (none)
//
int getBranchCount() const;

// Name: biasTableLookup
//
// Description: Returns branch bias from either simplesim BTB bias
// table or full bias table depending on the
// _enable_full_bias_table option.
// Arguments: The instruction whose address is to be looked-up
// Modifies: (none)
// Returns: The branch bias for the current address
// Assertions: (none)
// Exceptions: (none)
//
bool biasTableLookup( Instruction & instr );

// Name: updateBiasTable
//
// Description: Updates the branch bias table
// Arguments: (none)
// Modifies: The branch bias table state
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void updateBiasTable( Instruction & instr );

// Name: dumpOptions
//
// Description: Writes options specific information to the output
// stream
// Arguments: The output stream for writing
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void dumpOptions( ostream & out );

// Name: dumpStats
//
// Description: Writes statistic specific information to the output
// stream
// Arguments: The output stream for writing
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void dumpStats( ostream & out );

private:

```

```

class BiasTableEntry {
public:
    Address      _tag;
    int          _bias_count;
    bool         _bias_dir;
    bool         _bias_miss;
    BiasTableEntry *_newer;
    BiasTableEntry *_older;
};

typedef map< Address, BiasTableEntry *, less< Address > > BiasTable;

Predictor *_mpred;
bpred_t *_bpred;
int _bcount;           // Multiple branch predictor
                        // Last branch count

BiasScheme _bias_scheme; // Use full bias table or the
                        // one associated with the BTB?
int _bt_size;          // branch bias table size
int _bt_assoc;          // branch bias table assoc.
int _bias_threshold;    // branch bias threshold value

BiasTable _bt;          // Full branch bias table

TCLogger *_tc_log;      // TC trace output logger

//
// Statistics
//

long _btb_ops_retired;   // BTB ops retired
long _indir_ops_retired; // Indirect ops retired

}; // CoreFetchUnit

#endif

```

```

// File: $Id: HistoryUnit.h,v 1.6 2002/08/12 16:37:53 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Pure virtual class defining the interface for the history unit
// Revisions:
//   $Log: HistoryUnit.h,v $
// Revision 1.6 2002/08/12 16:37:53 ted
// Path-based related changes...
// Revision 1.5 2002/07/23 16:57:11 ted
// Removed pure virtual 'set' methods
// Revision 1.4 2002/07/09 00:55:56 ted
// Added setLogger and associated attribute
// Revision 1.3 2002/06/25 15:20:50 ted
// Added dumpOptions and dumpStats
// Revision 1.2 2002/06/21 15:45:27 ted
// Modified generateRenamerInput to return a dynamic pointer
// Revision 1.1 2002/06/17 22:42:22 ted
// Initial revision
//
// --As InputGenerator.h,v--
// Revision 1.4 2002/06/14 16:17:33 ted
// Modified addToHistory and changed removeFromHistory -> updateHistory
// Revision 1.3 2002/06/13 16:24:57 ted
// Added destructor
// Revision 1.2 2002/05/30 19:52:25 ted
// Revised generateRenamerInput to return a const reference
// Added addToHistory and removeFromHistory methods
// Revision 1.1 2002/05/29 20:09:10 ted
// Initial revision
//
//
// #ifndef _HISTORYUNIT_H
// #define _HISTORYUNIT_H
//
// #include "Outputable.h"
// #include "TraceCacheData.h"
// #include "Address.h"
// #include "HistoryToken.h"
// #include "Renamer.h"
// #include "TCLogger.h"
//
// class HistoryUnit public Outputable {
// public:
//
//     // Name: (destructor)
//
//     // Description: Provides a virtual base class destructor for proper
//     // destruction of derived classes when using polymorphism
//
//     // Arguments: (none)
//     // Modifies: Destroys/deallocates the object
//     // Returns: (none)

```

```

// Assertions: (none)
// Exceptions: (none)
// virtual ~HistoryUnit() { }
//
// // Name: generateRenamerInput (pure virtual)
//
// // Description: Returns a NEW TraceCacheData object based on either the
// // fetch address, previous history, or a combination of
// // the two.
// // Arguments: The current fetch address
// // Modifies: (none)
// // Returns: A pointer to the renamer input (dynamically allocated)
// // Assertions: (none)
// // Exceptions: (none)
//
// virtual const TraceCacheData * generateRenamerInput(
//     const Address & fetch_addr, const Renamer * ren_in ) = 0;
//
// // Name: setLogger
//
// // Description: Sets the TC trace (log file) output stream
// // Arguments: A reference to the output stream
// // Modifies: (none)
// // Returns: (none)
// // Assertions: (none)
// // Exceptions: (none)
//
// void setLogger( TCLogger * tc_log ) { _tc_log = tc_log; }
//
// protected:
//
//     TCLogger * _tc_log; // The TC trace output logger
// }; // HistoryUnit
//
// #endif

```

```
// File: $Id: Renamer.h,v 1.10 2002/07/23 16:57:11 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Pure virtual class defining the interface for trace cache
//               renamers.
// Revisions:
//   $Log: Renamer.h,v $
//   Revision 1.10 2002/07/23 16:57:11 ted
//   Removed pure virtual 'set' methods
//
//   Revision 1.9 2002/07/09 00:55:56 ted
//   Added setLogger and associated attribute
//
//   Revision 1.8 2002/06/27 15:22:33 ted
//   Minor method signature changes...
//
//   Revision 1.7 2002/06/25 15:20:50 ted
//   Added dumpOptions and dumpStats
//
//   Revision 1.6 2002/06/21 15:45:27 ted
//   Modified setCurrentTraceID to accept a pointer
//
//   Revision 1.5 2002/06/13 16:24:57 ted
//   Added destructor
//
//   Revision 1.4 2002/06/10 23:10:23 ted
//   Replaced TraceID with TraceCacheData
//
//   Revision 1.3 2002/05/29 20:09:10 ted
//   *** empty log message ***
//
//   Revision 1.2 2002/04/12 05:12:53 ted
//   *** empty log message ***
//
//   Revision 1.1 2002/04/12 00:44:21 ted
//   Initial revision
//
//
// #ifndef _RENAMER_H
// #define _RENAMER_H
//
// #include "Outputable.h"
// #include "TraceCacheData.h"
// #include "TCLogger.h"
//
// class Renamer public Outputable {
// public:
//
//     // Name: (destructor)
//
//     // Description: Provides a virtual base class destructor for proper
//     //               destruction of derived classes when using polymorphism
//
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//
//     virtual ~Renamer() { }
```

```
// Name: setCurrentTraceID (pure virtual)
// Description: Returns the current Trace ID based on the current
//               state
// Arguments: (none)
// Modifies: (none)
// Returns: The current a reference to the TraceID
// Assertions: (none)
// Exceptions: (none)
//
// virtual const TraceCacheData & getCurrentTraceID() = 0;
//
// Name: setCurrentTraceID (pure virtual)
// Description: Sets the current state of the renamer based on the
//               input and deallocates the renamer input.
// Arguments: A dynamic pointer to the input TraceCacheData
// Modifies: Deletes the renamer input
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// virtual void setCurrentTraceID( const TraceCacheData * ren_in ) = 0;
//
// Name: setLogger
// Description: Setss the TC trace (log file) output stream
// Arguments: A reference to the output stream
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// void setLogger( TCLogger * tc_log ) { _tc_log = tc_log; }
```

```
protected:
```

```
TCLogger * _tc_log; // The TC trace output logger
```

```
}; // Renamer
```

```
#endif
```

```

// File: $Id: TraceIDSelector.h,v 1.12 2002/07/23 16:57:11 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Pure virtual class defining the interface for trace cache
//               trace ID selectors.
// Revisions:
//   $Log: TraceIDSelector.h,v $
//   Revision 1.12 2002/07/23 16:57:11 ted
//   Removed pure virtual 'set' methods
//   Revision 1.11 2002/07/09 00:55:56 ted
//   Added setLogger and associated attribute
//   Revision 1.10 2002/06/28 17:16:15 ted
//   Debugging updates
//   Revision 1.9 2002/06/27 15:22:33 ted
//   Minor method signature changes...
//   Revision 1.8 2002/06/25 15:20:50 ted
//   Added dumpOptions and dumpStats
//   Revision 1.7 2002/06/18 15:34:12 ted
//   Changed Renamers references to pointers
//   Revision 1.6 2002/06/13 16:24:57 ted
//   Added destructor
//   Revision 1.5 2002/06/10 23:10:23 ted
//   Made setRenamerVector a virtual function
//   Revision 1.4 2002/05/30 17:39:37 ted
//   Added log tag to RCS header
//
// #ifndef _TRACEIDSELECTOR_H
// #define _TRACEIDSELECTOR_H
// #include <vector>
// #include "TraceCacheData.h"
// #include "Renamer.h"
// #include "TCLogger.h"
// typedef vector< Renamer * > RenamerVector;
// class TraceIDSelector : public Outputable {
// public:
//     // Name: (destructor)
//     // Description: Provides a virtual base class destructor for proper
//     //               destruction of derived classes when using polymorphism
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
// }

```

```

virtual ~TraceIDSelector() { }

// Name: selectTraceID (pure virtual)
// Description: Returns the 'best' Trace ID based from the enrolled
//               Renamers.
// Arguments: (none)
// Modifies: (none)
// Returns: A reference to the TraceID
// Assertions: (none)
// Exceptions: (none)
//
virtual const TraceCacheData & selectTraceID() const = 0;

// Name: setLogger
// Description: Setss the TC trace (log file) output stream
// Arguments: A reference to the output stream
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void setLogger( TCLogger * tc_log ) { _tc_log = tc_log; }

protected:
    TCLogger * _tc_log; // The TC trace output logger
}; // TraceIDSelector
#endif

```

```

// File:      $Id: TraceGenerator.h,v 1.17 2002/11/13 04:26:22 ted Exp ted $
// Author:     Edward Mulrane
// Contributors:
// Description: Pure virtual class defining the interface for trace cache
//               trace generators.
//
// Revisions:
//   $Log: TraceGenerator.h,v $
//   Revision 1.17  2002/11/13 04:26:22 ted
//   *** empty log message ***
//
//   Revision 1.16  2002/08/05 16:40:49 ted
//   Added pure virtual method getPredictionTokens
//
//   Revision 1.15  2002/07/23 16:57:11 ted
//   Removed pure virtual 'set' methods
//
//   Revision 1.14  2002/07/09 00:55:56 ted
//   Added setLogger and associated attribute
//
//   Revision 1.13  2002/07/03 13:31:23 ted
//   Debug...
//
//   Revision 1.12  2002/06/28 17:16:15 ted
//   Debugging updates
//
//   Revision 1.11  2002/06/27 01:54:16 ted
//   Various minor changes/bug fixes...
//
//   Revision 1.10  2002/06/25 15:20:50 ted
//   Added dumpOptions and dumpStats
//
//   Revision 1.9   2002/06/17 16:36:49 ted
//   Changes associated with RotenbergTrace
//
//   Revision 1.8   2002/06/13 22:07:51 ted
//   Deleted removeTrace
//
//   Revision 1.7   2002/06/13 16:24:57 ted
//   Further work in progress...
//
//   Revision 1.6   2002/06/11 16:08:56 ted
//   Work in progress...
//
//   Revision 1.5   2002/06/10 23:10:23 ted
//   Replaced TraceID with TraceCacheData
//
//   Revision 1.4   2002/06/06 19:38:20 ted
//   Minor change
//
//   Revision 1.3   2002/05/30 18:14:10 ted
//   minor change
//
//   Revision 1.2   2002/05/30 17:39:37 ted
//   Added log tag to RCS header
//
//
// #ifndef _TRACEGENERATOR_H
// #define _TRACEGENERATOR_H
//
// #include "Outputable.h"
//
// #include "Trace.h"
// #include "Predictor.h"
// #include "LookupTraceException.h"
// #include "TcLogger.h"
//
// class TraceGenerator : public Outputable {
// public:
//
//     // Name:      (destructor)
//     // Description: Provides a virtual base class destructor for proper
//     //               destruction of derived classes when using polymorphism
//     // Arguments:  (none)
//     // Modifies:   (none)
//     // Returns:    (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//
//     virtual ~TraceGenerator() {}
//
//     // Name:      lookupTrace (pure virtual)
//     // Description: Retrieves a trace based on the Trace ID. Checks for
//     //               hit/miss based on the trace ID and the current state.
//     //               If a miss occurs, an exception is thrown. Otherwise,
//     //               the trace is returned.
//     // Arguments:  The ID of the trace to retrieve.
//     // Modifies:   (none)
//     // Returns:    The specified trace.
//     // Assertions: (none)
//     // Exceptions: A LookupTraceException is thrown on a TC miss.
//
//     virtual Trace & lookupTrace( const TraceCacheData & tr_id )
//     throw ( LookupTraceException ) = 0;
//
//     // Name:      addTrace (pure virtual)
//     // Description: Adds a trace to the generator.
//     // Arguments:  A reference to the trace to add.
//     // Modifies:   (none)
//     // Returns:    Whether or not the trace was added
//     // Assertions: (none)
//     // Exceptions: (none)
//
//     virtual bool addTrace( Trace * tr ) = 0;
//
//     // Name:      getNextFetchAddress (pure virtual)
//     // Description: Returns the next fetch address based on the previous
//     //               call to lookupTrace.
//     // Arguments:  (none)
//     // Modifies:   (none)
//     // Returns:    The next fetch address
//     // Assertions: (none)
//     // Exceptions: (none)
//
//     virtual Address getNextFetchAddress() const = 0;

```

```
//
// Name:         getPredictionTokens (pure virtual)
//
// Description: Returns the PredictionToken vector associated with
// the last call to lookupTrace.
//
// Arguments:    (none)
// Modifies:    (none)
// Returns:     The PredictionToken vector
// Assertions:  (none)
// Exceptions:  (none)
//
virtual TokenVector & getPredictionTokens() = 0;

//
// Name:         setLogger
//
// Description: Sets the TC trace (log file) output stream
// Arguments:   A reference to the output stream
// Modifies:   (none)
// Returns:    (none)
// Assertions: (none)
// Exceptions: (none)
//
void setLogger( TLogger * tc_log ) { _tc_log = tc_log; }

protected:
    TLogger * _tc_log;           // The TC trace output logger
}; // TraceGenerator

#endif
```

```

// File:      $Id: FillUnit.h,v 1.10 2002/08/31 02:14:31 ted Exp $
// Author:    Edward Mulrane
// Contributors:
// Description: Pure virtual class defining the interface for the trace cache
//              fill unit.
//
// Revisions:
//   $Log: FillUnit.h,v $
// Revision 1.10 2002/08/31 02:14:31 ted
// Further updates/fixes...
//
// Revision 1.9 2002/07/23 16:57:11 ted
// Removed pure virtual 'set' methods
//
// Revision 1.8 2002/07/16 16:21:48 ted
// Further updates...
//
// Revision 1.7 2002/07/09 00:55:56 ted
// Added setLogger and associated attribute
//
// Revision 1.6 2002/06/28 17:16:15 ted
// Debugging updates...
//
// Revision 1.5 2002/06/25 15:20:50 ted
// Added dumpOptions and dumpStats
//
// Revision 1.4 2002/06/17 22:42:22 ted
// Changed all occurrences of InputGenerator to HistoryUnit
//
// Revision 1.3 2002/06/17 21:43:53 ted
// Finished first-draft revisions...
//
// Revision 1.2 2002/06/17 16:36:49 ted
// Added TraceGenerator pointer, and other changes
//
// Revision 1.1 2002/06/14 16:17:33 ted
// Initial revision
//
//
// #ifndef _FILLUNIT_H
// #define _FILLUNIT_H
//
// #include "Trace.h"
// #include "Instruction.h"
// #include "Predictor.h"
// #include "HistoryUnit.h"
// #include "TraceGenerator.h"
//
// class FillUnit public Outputable {
// public:
//
//     // Name: (destructor)
//
//     // Description: Provides a virtual base class destructor for proper
//     //              destruction of derived classes when using polymorphism
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)

```

```

// Exceptions: (none)
//
// virtual ~FillUnit() { }
//
// Name: drain (pure virtual)
//
// Description: Explicitly ends any trace currently being appended too
//              in the fill unit
// Arguments: (none)
// Modifies: Explicitly ends the trace being formed
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// virtual void drain() = 0;
//
// Name: enqueueInstruction (pure virtual)
//
// Description: Adds a retired instruction to the instruction
//              stream register. A terminating condition will
//              trigger a compare with the head of the trace FIFO
//              and clear the stream register
// Arguments: A reference to the retired instruction
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// virtual void enqueueInstruction( Instruction & instr ) = 0;
//
// Name: setLogger
//
// Description: Setss the TC trace (log file) output stream
// Arguments: A reference to the output stream
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// void setLogger( TCLogger * tc_log ) { _tc_log = tc_log; }
//
protected:
    TCLogger * _tc_log; // The TC trace output logger
}; // FillUnit
#endif

```



```
// File: $Id: Predictor.h,v 1.17 2002/10/28 19:00:51 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Pure virtual class providing the interface for
//              branch/trace predictors
// Revisions:
//   $Log: Predictor.h,v $
// Revision 1.17 2002/10/28 19:00:51 ted
// *** empty log message ***
//
// Revision 1.16 2002/10/08 19:49:52 ted
// *** empty log message ***
//
// Revision 1.15 2002/09/02 21:34:07 ted
// prior to removing token_queue
//
// Revision 1.14 2002/08/31 02:14:31 ted
// Further updates/fixes...
//
// Revision 1.13 2002/07/08 18:03:32 ted
// Added newToken method
//
// Revision 1.12 2002/07/03 13:30:39 ted
// Debug...
//
// Revision 1.11 2002/06/28 17:16:15 ted
// Debugging updates
//
// Revision 1.10 2002/06/25 15:20:50 ted
// Added dumpOptions and dumpStats
//
// Revision 1.9 2002/06/24 16:50:03 ted
// Removed static members and added size parameter to constructor
//
// Revision 1.8 2002/06/18 23:34:05 ted
// Added virtual destructor...
//
// Revision 1.7 2002/06/17 21:43:53 ted
// Added branch direction attribute/associated members
//
// Revision 1.6 2002/06/13 16:24:57 ted
// Revised lookup, added speculativeUpdate and TokenVector member
//
// Revision 1.5 2002/06/10 16:48:40 ted
// Changed token reference in update to pointer type
//
// Revision 1.4 2002/06/07 18:28:50 ted
// Further updates...
//
// Revision 1.3 2002/06/06 19:36:22 ted
// Started filling out body of class
//
// Revision 1.2 2002/05/30 17:39:37 ted
// Added log tag to RCS header
//
//
// #ifndef _PREDICTOR_H
// #define _PREDICTOR_H
//
// include <vector>

```

```
#include "Outputable.h"
#include "Printable.h"
#include "Address.h"
#include "PredictionToken.h"

typedef vector< PredictionToken * > TokenVector;

class Predictor . public Outputable,
                 public Printable {
public:
    // Name: NewTokenType (enum)
    enum NewTokenType { BTB_MISS, DIR_CHANGE, PROMOTED_BRANCH,
                       PROMOTED_JUMP };

    // Name: (constructor)
    // Description: Initializes the predictor width (basic block count)
    // Arguments: The predictor width
    // Modifies: Initializes the predictor width
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    Predictor( int width );

    // Name: (destructor)
    // Description: Provides a virtual base class destructor for proper
    // destruction of derived classes when using polymorphism
    // Arguments: (none)
    // Modifies: Destroys/deallocates the object
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    virtual ~Predictor() {}

    // Name: getWidth
    // Description: Gets the predictor bandwidth (number of branches
    // predicted per cycle).
    // Arguments: (none)
    // Modifies: (none)
    // Returns: The predictor width
    // Assertions: (none)
    // Exceptions: (none)
    int getWidth() const;

    // Name: lookup (pure virtual)
    // Description: Fills the PredictionToken vector (of established
    // width) with the current multiple branch prediction and
    // predictor update/restore information
    //
}

```

```

// Arguments: The branch address
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void lookup( const Address & baddr ) = 0;

//
// Name: getPredictionTokens (pure virtual)
//
// Description: Returns the PredictionToken vector (of established
// width). It is the consumers responsibility to
// deallocate the dynamically allocated space for all
// tokens
// Arguments: (none)
// Modifies: (none)
// Returns: The PredictionToken vector
// Assertions: (none)
// Exceptions: (none)
//
virtual TokenVector & getPredictionTokens() = 0;

//
// Name: newToken (pure virtual)
//
// Description: Generates a newly allocated token for the specified
// branch address
// Arguments: The branch address and direction
// If clone_token is true, then the non-directional state
// information for the specified token (clone_num) in the
// current vector is used in the new token.
// Modifies: (none)
// Returns: A new prediction token.
// Assertions: (none)
//
virtual PredictionToken * newToken( const Address & baddr,
                                   bool bdir,
                                   NewTokenType tok_type,
                                   bool clone_token = false,
                                   int clone_num = 0 ) = 0;

//
// Name: speculativeUpdate (pure virtual)
//
// Description: Updates the predictor state before the branch outcome
// is known
// Arguments: The PredictionToken
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void speculativeUpdate( PredictionToken * ptok ) = 0;

//
// Name: update (pure virtual)
//
// Description: Updates the predictor state once the branch outcome
// is known
// Arguments: The PredictionToken and the actual branch outcome

```

```

// Modifies: Deallocates the prediction token
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void update( PredictionToken * ptok, bool bdir ) = 0;

//
// Name: mispredictRecover (pure virtual)
//
// Description: Updates the predictor state when a misprediction has
// been detected.
// Arguments: PredictionToken of mis-speculated instruction
// Modifies: Internal predictor state
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
virtual void mispredictRecover( PredictionToken * ptok ) = 0;

protected:

    int _width; // predictor bandwidth

}; // Predictor

#endif

```

```
File: $Id: HistoryToken.h,v 1.1 2002/06/14 16:17:33 ted Exp $
Author: Edward Mulrane
Contributors:
Description: Virtual class providing the interface for history tokens.
Revisions:
    $Log: HistoryToken.h,v $
    Revision 1.1 2002/06/14 16:17:33 ted
    Initial revision

#ifdef _HISTORYTOKEN_H
#define _HISTORYTOKEN_H
#include "TraceCacheData.h"

class HistoryToken public TraceCacheData {
public:
    virtual const TraceCacheData & getHistoryData() const = 0;
}; // HistoryToken

#endif
```

```

// File: $Id: PredictionToken.h,v 1.9 2002/11/13 04:36:22 ted Exp ted $
// Author: Edward Mulrane
// Contributors:
// Description: Virtual class providing the interface for
//              branch prediction tokens.
// Revisions:
// $Log: PredictionToken.h,v $
// Revision 1.9 2002/11/13 04:26:22 ted
// *** empty log message ***
//
// Revision 1.8 2002/10/28 19:00:51 ted
// *** empty log message ***
//
// Revision 1.7 2002/08/05 16:38:51 ted
// Moved _baddr attribute from PredictionToken to MQAgToken
//
// Revision 1.6 2002/07/03 13:30:39 ted
// Debug...
//
// Revision 1.5 2002/06/28 17:16:15 ted
// Debugging updates
//
// Revision 1.4 2002/06/17 21:43:53 ted
// Added branch direction attribute/associated members
//
// Revision 1.3 2002/06/10 21:45:41 ted
// Finished first-draft revisions...
//
// Revision 1.2 2002/06/10 16:48:40 ted
// Changed token reference in update to pointer type
//
// Revision 1.1 2002/06/07 18:28:50 ted
// Initial revision
//
//
// #ifndef _PREDICTIONTOKEN_H
// #define _PREDICTIONTOKEN_H
//
// #include "TraceCacheData.h"
// #include "Address.h"
//
// class PredictionToken public TraceCacheData {
// public:
//
//     // Name: (destructor)
//     //
//     // Description: Provides a virtual base class destructor for proper
//     //              destruction of derived classes when using polymorphism
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     virtual ~PredictionToken() { }
//
//     //
//     // Name: getPrediction
//     // Description: Returns the prediction

```

```

// Arguments: (none)
// Modifies: (none)
// Returns: Returns the prediction
// Assertions: (none)
// Exceptions: (none)
//
// virtual bool getPrediction() const;
//
// Name: asLong
//
// Description: Returns a long representation of the PredictionToken
// Arguments: (none)
// Modifies: (none)
// Returns: The data represented as a long
// Assertions: (none)
// Exceptions: (none)
//
// virtual long asLong() const;
//
// protected:
//
// enum PredictionSource { PHT, BTB_MISS, DIR_CHANGE, PROMOTED_BRANCH,
//                         PROMOTED_JUMP };
//
// bool _pred; //Branch prediction
//
// }; // PredictionToken
//
// #endif

```

```

// File:      $Id: Trace.h,v 1.13 2002/10/08 19:49:52 ted Exp $
// Author:    Edward Mulrane
// Contributors:
// Description: Pure virtual class providing the interface for the trace
//              data representation.
// Revisions:
//   $Log: Trace.h,v $
//   Revision 1.13 2002/10/08 19:49:52 ted
//   *** empty log message ***
//
//   Revision 1.12 2002/08/12 16:37:53 ted
//   Path-based related changes...
//
//   Revision 1.11 2002/07/24 20:49:25 ted
//   Added getSequenceID virtual member
//
//   Revision 1.10 2002/07/08 18:03:32 ted
//   Minor changes...
//
//   Revision 1.9 2002/07/03 13:31:23 ted
//   Debug...
//
//   Revision 1.8 2002/06/28 17:16:15 ted
//   Debugging updates
//
//   Revision 1.7 2002/06/25 15:20:50 ted
//   Added dumpOptions and dumpStats
//
//   Revision 1.6 2002/06/21 15:45:27 ted
//   Added pure virtual method getTraceVector
//
//   Revision 1.5 2002/06/10 23:10:23 ted
//   Replaced TraceID with TraceCacheData
//
//   Revision 1.4 2002/06/03 16:32:39 ted
//   Added getTraceID member
//
//   Revision 1.3 2002/05/30 17:39:37 ted
//   Added log tag to RCS header
//
//
// #ifndef TRACE_H
// #define TRACE_H
//
// #include <deque>
// #include "TraceCacheData.h"
// #include "Address.h"
// #include "Instruction.h"
// #include "Outputable.h"
// #include "Printable.h"
// #include "../machine.h"
//
// typedef deque< Instruction > TraceVector;
//
// class Trace
// {
//     public Outputable,
//     public Printable {
//
//     public:
//
//         //

```

```

// Name:      getTraceID (pure virtual)
//
// Description: Returns the TraceID of the current Trace
// Arguments:  (none)
// Modifies:  (none)
// Returns:    The current TraceID
// Assertions: (none)
// Exceptions: (none)
//
// virtual TraceCacheData & getTraceID() = 0;
//
// Name:      getTraceVector (pure virtual)
//
// Description: Returns the vector of instructions comprising the
//              trace
// Arguments:  (none)
// Modifies:  (none)
// Returns:    The trace vector
// Assertions: (none)
// Exceptions: (none)
//
// virtual TraceVector & getTraceVector() = 0;
//
// Name:      getStartingPC (pure virtual)
//
// Description: Returns the starting PC address of the trace
// Arguments:  (none)
// Modifies:  (none)
// Returns:    The starting PC address
// Assertions: (none)
// Exceptions: (none)
//
// virtual Address getStartingPC() const = 0;
//
// Name:      getBranchCount (pure virtual)
//
// Description: Returns the branch count for the segment
// Arguments:  (none)
// Modifies:  (none)
// Returns:    The branch count
// Assertions: (none)
// Exceptions: (none)
//
// virtual int getBranchCount() const = 0;
//
// Name:      getSequenceID (pure virtual)
//
// Description: Returns the sequence number for the segment
// Arguments:  (none)
// Modifies:  (none)
// Returns:    The sequence ID
// Assertions: (none)
// Exceptions: (none)
//
// virtual unsigned getSequenceID() const = 0;

```

); // Trace

endif

Trace.h

Page 2 of 2

```

// File: $Id: Instruction.h,v 1.20 2002/10/28 19:00:51 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: Class encapsulating a single instruction
// Revisions:
// $Log: Instruction.h,v $
// Revision 1.20 2002/10/28 19:00:51 ted
// *** empty log message ***
//
// Revision 1.19 2002/10/16 16:33:27 ted
// Fixed branch promotion mechanism...
//
// Revision 1.18 2002/10/14 22:43:50 ted
// before Instruction::isBtOp() -> Instruction::isControl() change
//
// Revision 1.17 2002/10/08 19:49:52 ted
// *** empty log message ***
//
// Revision 1.16 2002/07/24 20:49:25 ted
// Added _fetch_source and _tc_seq attributes.
//
// Revision 1.15 2002/07/16 16:21:48 ted
// Further updates...
//
// Revision 1.14 2002/07/09 17:37:47 ted
// Added disassemble method
//
// Revision 1.13 2002/07/08 18:03:32 ted
// Added operator<<
//
// Revision 1.12 2002/07/03 13:30:39 ted
// Debug...
//
// Revision 1.11 2002/06/30 01:39:00 ted
// Added isBranch member function
//
// Revision 1.10 2002/06/28 17:16:15 ted
// Debugging updates...
//
// Revision 1.9 2002/06/27 01:54:16 ted
// Various minor changes/bug fixes...
//
// Revision 1.8 2002/06/21 15:45:27 ted
// Added operator==
//
// Revision 1.7 2002/06/20 16:50:25 ted
// Minor revisions...
//
// Revision 1.6 2002/06/19 17:12:02 ted
// Added a few predicate members and target attribute
//
// Revision 1.5 2002/06/17 21:43:53 ted
// Added PredictionToken attribute/associated members and
// an isControl predicate member
//
// Revision 1.4 2002/06/17 16:36:49 ted
// Added Address attribute and associated changes
//
// Revision 1.3 2002/06/10 16:48:40 ted
// Changed declarations from unsigned -> int
//
//
// Revision 1.2 2002/06/03 16:32:39 ted
// Filled out class to mirror (with addition of isNOP)
// the Address class
//
// Revision 1.1 2002/05/31 20:26:34 ted
// Initial revision
//
//
// #ifndef _INSTRUCTION_H
// #define _INSTRUCTION_H
//
// #include <ostream.h>
// #include <string>
//
// #include "Address.h"
// #include "PredictionToken.h"
// #include "TCDefs.h"
//
// #include "../machine.h"
//
// class Instruction : public TraceCacheData {
// public:
//
//     // Name: dataSize (static)
//     // Description: Returns the size (in bytes) of an instruction
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: The size of an instruction
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     static int dataSize();
//
//     // Name: (constructor)
//     // Description: Creates an instruction container
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     Instruction();
//
//     // Name: (constructor)
//     // Description: Creates/initializes an instruction container
//     // Arguments: Values for the instruction, address, and predetion
//     // token
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     Instruction( const md_inst_t & inst, const Address & addr );

```

```

// Name: operator()
// Description: Functor returning the md_inst_t value
// Arguments: (none)
// Modifies: (none)
// Returns: The md_inst_t value (const)
// Assertions: (none)
// Exceptions: (none)
//
const md_inst_t & operator()();

// Name: operator==
// Description: Compares two instructions for equality (includes PC
// address value)
// Arguments: The object to compare the current with
// Modifies: (none)
// Returns: The result of the equality comparison
// Assertions: (none)
// Exceptions: (none)
//
bool operator==( const Instruction & other ) const;

// Name: getAddress
// Description: Returns the address associated with the instruction
// Arguments: (none)
// Modifies: (none)
// Returns: The address associated with the instruction
// Assertions: (none)
// Exceptions: (none)
//
const Address & getAddress() const;

// Name: getTarget
// Description: Returns the target address of the instruction
// Arguments: (none)
// Modifies: (none)
// Returns: The target address associated with the instruction
// Assertions: (none)
// Exceptions: (none)
//
const Address & getTarget() const;

// Name: setBranchDirection
// Description: Sets the branch direction (actual outcome)
// Arguments: The branch direction
// Modifies: The branch direction
// Returns: (none)
// Assertions: Instruction must be a control op
// Exceptions: (none)
//
void setBranchDirection( bool bdir );

```

```

// Name: getBranchDirection
// Description: Returns the branch direction (actual outcome)
// Arguments: (none)
// Modifies: (none)
// Returns: Returns the branch direction
// Assertions: (none)
// Exceptions: (none)
//
bool getBranchDirection() const;

// Name: setPredictionToken
// Description: Sets the prediction token pointer associated with the
// instruction
// Arguments: A pointer to the prediction token
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void setPredictionToken( PredictionToken * ptok );

// Name: getPredictionToken
// Description: Returns a pointer to the prediction token associated
// with the instruction
// Arguments: (none)
// Modifies: (none)
// Returns: A pointer to the prediction token
// Assertions: (none)
// Exceptions: (none)
//
PredictionToken * getPredictionToken();

// Name: setFetchSource
// Description: Sets the instruction fetch source
// Arguments: The fetch source
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void setFetchSource( FetchSource fetch_source );

// Name: getFetchSource
// Description: Returns the instruction fetch source
// Arguments: (none)
// Modifies: (none)
// Returns: The instruction fetch source
// Assertions: (none)
// Exceptions: (none)
//
FetchSource getFetchSource() const;

```



```

// Name: setTraceCacheSequence
// Description: Sets the trace cache sequence number associated with
// the instruction
// Arguments: The trace cache sequence number
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void setTraceCacheSequence( unsigned tc_seq );

// Name: getTraceCacheSequence
// Description: Returns the trace cache sequence number associated with
// the instruction
// Arguments: (none)
// Modifies: (none)
// Returns: The trace cache sequence number
// Assertions: (none)
// Exceptions: (none)
//
unsigned getTraceCacheSequence() const;

// Name: promoteBranch
// Description: Sets the promotion bit of the branch instruction to true
// Arguments: (none)
// Modifies: The promotion bit
// Returns: (none)
// Assertions: (none)
// Exceptions: Valid only for branches (fails otherwise)
//
void promoteBranch();

// Name: demoteBranch
// Description: Sets the promotion bit of the branch instruction to
// false
// Arguments: (none)
// Modifies: The promotion bit
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void demoteBranch();

// Name: getPromotionFlag
// Description: Returns the value of the branch promotion bit
// Arguments: (none)
// Modifies: The branch promotion bit (false for non-branches)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
bool Instruction::getPromotionFlag() const;

// Name: setReassocFlag
// Description: Sets the reassociation flag for this instruction
// Arguments: (none)
// Modifies: The reassociation bit
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void Instruction::setReassocFlag( bool reassoc );

// Name: getReassocFlag
// Description: Gets the reassociation flag for this instruction
// Arguments: (none)
// Modifies: (none)
// Returns: The value of the reassociation flag
// Assertions: (none)
// Exceptions: (none)
//
bool Instruction::getReassocFlag() const;

// Name: setTarget
// Description: Sets the target instruction for an indirect jump
// (non-determinable)
// Arguments: The target address
// Modifies: (none)
// Returns: (none)
// Assertions: Valid only for Indirect Jump ops (fails otherwise)
// Exceptions: (none)
//
void setTarget( const Address & targ );

// Name: isNOP
// Description: Returns true if the instruction is a no-op
// Arguments: (none)
// Modifies: (none)
// Returns: True if the instruction is a no-op
// Assertions: (none)
// Exceptions: (none)
//
bool isNOP() const;

// Name: isControl
// Description: Returns true if the instruction is a control instr.
// Arguments: (none)
// Modifies: (none)
// Returns: True if the instruction is a control instruction
// Assertions: (none)
// Exceptions: (none)
//
bool isControl() const;

// Name: isDirectJump

```

```

// Description: Returns true if the instruction is a direct jump
// Arguments: (none)
// Modifies: (none)
// Returns: True if the instruction is a direct jump
// Assertions: (none)
// Exceptions: (none)
//
bool isDirectJump() const;

//
// Name: isIndirectJump
//
// Description: Returns true if the instruction is an indirect jump
// Arguments: (none)
// Modifies: (none)
// Returns: True if the instruction is an indirect jump
// Assertions: (none)
// Exceptions: (none)
//
bool isIndirectJump() const;

//
// Name: isBranch
//
// Description: Returns true if the instruction is an explicit branch
// instruction or a direct jump (Applicable for the BTB)
// Arguments: (none)
// Modifies: (none)
// Returns: True if the above conditions are true
// Assertions: (none)
// Exceptions: (none)
//
bool isBranch() const;

//
// Name: isBTBOp
//
// Description: Returns true if the instruction is an explicit branch
// instruction or a direct jump (Applicable for the BTB)
// Arguments: (none)
// Modifies: (none)
// Returns: True if the above conditions are true
// Assertions: (none)
// Exceptions: (none)
//
bool isBTBOp() const;

//
// Name: isTrap
//
// Description: Returns true if the instruction is a TRAP or SYSCALL
// Arguments: (none)
// Modifies: (none)
// Returns: True if the above conditions are true
// Assertions: (none)
// Exceptions: (none)
//
bool isTrap() const;

```

```

// Name: isCall
//
// Description: Returns true if the instruction is a subroutine call
// Arguments: (none)
// Modifies: (none)
// Returns: True if the above conditions are true
// Assertions: (none)
// Exceptions: (none)
//
bool isCall() const;

//
// Name: isLargeBackwardDisp
//
// Description: Returns true if the instruction is conditional branch
// with a large (>= 32) backward displacement
// Arguments: (none)
// Modifies: (none)
// Returns: True if the above conditions are true
// Assertions: (none)
// Exceptions: (none)
//
bool isLargeBackwardDisp() const;

//
// Name: asLong
//
// Description: Returns a long representation of the Instruction
// Arguments: (none)
// Modifies: (none)
// Returns: The data represented as a long (_inst.a)
// Assertions: (none)
// Exceptions: (none)
//
virtual long asLong() const;

//
// Name: disassemble
//
// Description: Returns the string representation of the instruction
// and its operands (modified from md_print_insn() )
// Arguments: (none)
// Modifies: (none)
// Returns: The instruction as an assembly op
// Assertions: (none)
// Exceptions: (none)
//
string disassemble() const;

//
// Name: friend operator<<
//
// Description: Prints the instruction to the output stream
// Arguments: The ostream object and the instruction object
// Modifies: Writes the instruction to the output stream
// Returns: The output stream
// Assertions: (none)
// Exceptions: (none)
//
friend ostream & operator<< ( ostream & os, const Instruction & inst );

```

```
private:
```

```
md_inst_t
Address
Address
    _inst;
    _addr;
    _targ;

    bool
    PredictionToken * _ptok;
    FetchSource
    unsigned
    md_opcode

}; //Instruction

#endif
```

// the "raw" instruction data
// the instruction address
// the branch target

// actual branch outcome
// the associated pred. token
// true -> tc, false -> fu
// trace cache sequence num.
// the associated opcode enum

```
// File: $Id: Address.h,v 1.11 2002/07/08 18:01:17 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: An container object for the md_addr_t type
// Revisions:
// $Log: Address.h,v $
// Revision 1.11 2002/07/08 18:01:17 ted
// Added operator<<
// Revision 1.10 2002/07/03 13:30:39 ted
// Debug...
// Revision 1.9 2002/06/21 15:45:27 ted
// Added operator++
// Revision 1.8 2002/06/20 16:50:25 ted
// Minor revisions
// Revision 1.7 2002/06/19 17:12:02 ted
// Added valid flag
// Revision 1.6 2002/06/13 16:24:57 ted
// Added operator< and getIndex methods
// Revision 1.5 2002/06/11 16:08:56 ted
// Added operator==
// Revision 1.4 2002/06/10 23:10:23 ted
// Replaced TraceID with TraceCacheData
// Revision 1.3 2002/06/10 16:48:40 ted
// Changed declarations from unsigned -> int
// Revision 1.2 2002/06/03 16:32:39 ted
// Changed parent class from TraceCacheData to TraceID
// Revision 1.1 2002/05/29 20:09:10 ted
// Initial revision
//
// #ifndef _ADDRESS_H
// #define _ADDRESS_H
// #include <ostream.h>
// #include "TraceCacheData.h"
// #include "../machine.h"
// class Address public TraceCacheData {
// public:
//     // Name: dataSize (static)
//     //
//     // Description: Returns the size (in bytes) of an address
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: The size of an address
//     // Assertions: (none)
//     // Exceptions: (none)
// }
```

```
// static int dataSize();
//
// Name: (constructor)
//
// Description: Creates an address container (valid = false)
// Arguments: (none)
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// Address();
//
// Name: (constructor)
//
// Description: Creates/initializes an address container (valid = true)
// Arguments: Initial value
// Modifies: (none)
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// Address( const md_addr_t & address );
//
// Name: operator=
//
// Description: Assignment operator
// Arguments: md_addr_t to copy
// Modifies: (none)
// Returns: reference to the newly assigned container
// Assertions: (none)
// Exceptions: (none)
//
// const Address & operator=( const md_addr_t & address );
//
// Name: operator==
//
// Description: Compares two Addresses for equality
// Arguments: (ignores valid status)
// Modifies: The object to compare the current with
// Returns: (none)
// Assertions: The result of the equality comparison
// Exceptions: (none)
//
// bool operator==( const Address & other ) const;
//
// Name: operator<
//
// Description: Checks if one object is less than another
// Arguments: The object to compare the current with
// Modifies: (none)
// Returns: The result of the less-than comparison
// Assertions: (none)
// Exceptions: (none)
```

```

// bool operator<( const Address & other ) const;
//
// Name: operator++
//
// Description: Increments the current address by one
// Arguments: (none)
// Modifies: The current address
// Returns: The incremented address
// Assertions: (none)
// Exceptions: (none)
//
const Address & operator++();
//
// Name: operator()
//
// Description: Functor returning the md_addr_t value
// Arguments: (none)
// Modifies: (none)
// Returns: The md_addr_t value (const)
// Assertions: (none)
// Exceptions: (none)
//
const md_addr_t & operator() () const;
//
// Name: asLong
//
// Description: Returns a long representation of the PredictionToken
// Arguments: (none)
// Modifies: (none)
// Returns: The data represented as a long
// Assertions: (none)
// Exceptions: (none)
//
virtual long asLong() const;
//
// Name: getIndex
//
// Description: Returns an index given the width in bits
// Arguments: The width of the index in bits
// Modifies: (none)
// Returns: The index for the current address
// Assertions: (none)
// Exceptions: (none)
//
Address getIndex( int width ) const;
//
// Name: isValid
//
// Description: Returns the valid status of the address
// Arguments: (none)
// Modifies: (none)
// Returns: Returns the valid status of the address
// Assertions: (none)
// Exceptions: (none)
//
bool isValid() const;
//
// Name: validate
//
// Description: Sets the status of the address to valid
// Arguments: (none)
// Modifies: The status of the address
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void validate();
//
// Name: invalidate
//
// Description: Sets the status of the address to invalid
// Arguments: (none)
// Modifies: The status of the address
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
void invalidate();
//
// Name: friend operator<<
//
// Description: Prints the address (in hex) to the output stream
// Arguments: The ostream object and the Address object
// Modifies: Writes the address to the output stream
// Returns: The output stream
// Assertions: (none)
// Exceptions: (none)
//
friend ostream & operator<<( ostream & os, const Address & addr );
//
private:
    md_addr_t _address;
    bool _valid;
}; //Address
#endif

```

```

// File:      $Id: TraceCacheData.h,v 1.4 2002/07/03 13:31:23 ted Exp $
// Author:     Edward Mulrane
// Contributors:
// Description: An empty interface for all trace cache data representations
//              (provides a common super-class).
// Revisions:
//   $Log: TraceCacheData.h,v $
//   Revision 1.4  2002/07/03 13:31:23  ted
//   Debug...
//
//   Revision 1.3  2002/06/11 18:51:10  ted
//   Minor revision...
//
//   Revision 1.2  2002/06/11 16:08:56  ted
//   Added virtual destructor
//
//   Revision 1.1  2002/05/29 20:09:10  ted
//   Initial revision
//
//
// #ifndef _TRACECACHEDATA_H
// #define _TRACECACHEDATA_H
//
// class TraceCacheData {
// public:
//     // Name:      (destructor)
//     //
//     // Description: Provides a virtual base class destructor for proper
//     //              destruction of derived classes when using polymorphism
//     // Arguments:  (none)
//     // Modifies:   Destroys/deallocates the object
//     // Returns:    (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     virtual ~TraceCacheData() { }
//
//     // Name:      asLong (pure virtual)
//     //
//     // Description: Returns a primitive (long) representation of the Data
//     // Arguments:  (none)
//     // Modifies:   (none)
//     // Returns:    The data represented as a long
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     virtual long asLong() const = 0;
// };

```

```

// #endif

```

```

// File:          $Id: TCDefs.h,v 1.5 2002/10/14 22:44:54 ted Exp $
// Author:        Edward Mulrane
// Contributors:
// Description:    Contains definitions for the Trace Cache extension
// Revisions:
//   $Log: TCDefs.h,v $
//   Revision 1.5  2002/10/14 22:44:54 ted
//   before Instruction::isBtOp() -> Instruction::isControl() change
//   Revision 1.4  2002/07/24 20:49:25 ted
//   Changed IFQFetchInfo to FetchSource
//   Revision 1.3  2002/07/16 16:21:48 ted
//   Further updates...
//   Revision 1.2  2002/07/10 18:32:50 ted
//   Added IFQFetchInfo enum
//   Revision 1.1  2002/05/31 20:45:35 ted
//   Initial revision
//
#ifndef _TCDEFS_H
#define _TCDEFS_H

#define TAKEN true
#define NOT_TAKEN false

#include <cmath>

enum BiasScheme {
    FULL_BIAS_TABLE,
    BTB_BIAS_TABLE,
    COST_REDUCED_BIAS_TABLE,
    NO_BIAS_TABLE
};

enum FetchSource { FU = 0, TC_BEGIN, TC_MID, TC_END, TC };

int mylog2( int x );

#endif

```

```

// (logger or bit bucket)
// Arguments: (none)
// Modifies: The current state
// Returns: (none)
// Assertions: (none)
// Exceptions: (none)
//
// void turnOn();
// void turnOff();

private:
    ostream * _tct_out;
    ofstream _tct_fout;
    ofstream _nul_fout;
    bool _onoff;

}; // Class

endif

```



```

// File:      $Id: Outputable.h,v 1.1 2002/06/25 15:20:50 ted Exp $
// Author:    Edward Mulrane
// Contributors:
// Description: An Interface defining options/statistic specific output
//              members.
// Revisions:
//   $Log: Outputable.h,v $
//   Revision 1.1  2002/06/25 15:20:50 ted
//   Initial revision
//

#ifndef _OUTPUTABLE_H
#define _OUTPUTABLE_H

#include <ostream.h>

class Outputable {
public:
    // Name:      dumpOptions (pure virtual)
    // Description: Writes options specific information to the output
    //              stream
    // Arguments:  The output stream for writing
    // Modifies:  (none)
    // Returns:    (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    virtual void dumpOptions( ostream & out ) = 0;

    // Name:      dumpStats (pure virtual)
    // Description: Writes statistic specific information to the output
    //              stream
    // Arguments:  The output stream for writing
    // Modifies:  (none)
    // Returns:    (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    virtual void dumpStats( ostream & out ) = 0;
}; // Outputable

#endif

```

```
// File: $Id: Printable.h,v 1.1 2002/07/08 18:03:32 ted Exp $
// Author: Edward Mulrane
```

```
// File:      $Id: LookupTraceException.h,v 1.2 2002/07/09 17:37:47 ted Exp $
// Author:    Edward Mulrane
// Contributors:
// Description: A class derived from TraceCacheException thrown when
//              TraceGenerator::lookupTrace fails to return a valid Trace.
// Revisions:
//   $Log: LookupTraceException.h,v $
//   Revision 1.2 2002/07/09 17:37:47 ted
//   Accept string argument rather than char *
//
//   Revision 1.1 2002/06/28 17:16:15 ted
//   Initial revision
//
//   --As GetTraceException--
//   Revision 1.1 2002/06/11 16:08:56 ted
//   Initial revision
//
//
// #ifndef _LOOKUPTRACEEXCEPTION_H
// #define _LOOKUPTRACEEXCEPTION_H
//
// #include "TraceCacheException.h"
//
// class LookupTraceException : public TraceCacheException {
// public:
//     // Name:      (constructor)
//     //
//     // Description: Creates the exception object
//     // Arguments:  Pointer to the message string
//     // Modifies:   (none)
//     // Returns:    (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     LookupTraceException( const string & message );
// }; // LookupTraceException
//
// #endif
```

```
// File: $Id: InstructionNotAddedException.h,v 1.3 2002/07/09 17:37:47 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: A class derived from TraceCacheException thrown when an
//              instruction is unable to be added to a trace segment.
// Revisions:
//   $Log: InstructionNotAddedException.h,v $
//   Revision 1.3 2002/07/09 17:37:47 ted
//   Accept string argument rather than char *
//
//   Revision 1.2 2002/06/21 15:45:27 ted
//   Killed const from Instruction references.
//
//   Revision 1.1 2002/06/17 16:36:49 ted
//   Initial revision
//
//
// #ifndef _INSTRUCTIONNOTADDEXCEPTION_H
// #define _INSTRUCTIONNOTADDEXCEPTION_H
//
// #include "TraceCacheException.h"
// #include "Instruction.h"
//
// class InstructionNotAddedException : public TraceCacheException {
// public:
//     // Name: (constructor)
//     //
//     // Description: Creates the exception object
//     // Arguments: Pointer to the message string, instruction that was
//                 not added
//     // Modifies: (none)
//     // Returns: (none)
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     InstructionNotAddedException( const string & message,
//                                   Instruction & instr );
//
//     // Name: getInstruction
//     //
//     // Description: Returns the instruction not added
//     // Arguments: (none)
//     // Modifies: (none)
//     // Returns: The instruction not added
//     // Assertions: (none)
//     // Exceptions: (none)
//     //
//     Instruction & getInstruction();
//
// private:
//     Instruction _instr;
// }; // InstructionNotAddedException
//
// #endif
```

```
#endif
```

```
// File: $Id: TraceCacheException.h,v 1.2 2002/07/09 17:37:47 ted Exp $
// Author: Edward Mulrane
// Contributors:
// Description: A class derived from exception providing a message string
//              (used for further extension)
// Revisions:
// $Log: TraceCacheException.h,v $
// Revision 1.2 2002/07/09 17:37:47 ted
// Accept string argument rather than char *
//
// Revision 1.1 2002/04/26 03:15:28 ted
// Initial revision
//
//
#ifndef _TRACECACHEEXCEPTION_H
#define _TRACECACHEEXCEPTION_H

#include <exception>
#include <string>

class TraceCacheException public exception {
public:
    // Name: (constructor)
    //
    // Description: Creates the exception object
    // Arguments: Pointer to the message string
    // Modifies: (none)
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    TraceCacheException( const string & message );

    // Name: what
    //
    // Description: Returns the appropriate exception message
    // Arguments: (none)
    // Modifies: (none)
    // Returns: (none)
    // Assertions: (none)
    // Exceptions: (none)
    //
    virtual const char * what() const;

private:
    string _message;
}; // TraceCacheException

#endif
```

Appendix B

Kernel Benchmark Source Files

```

/*
MODULE NAME:      roots.c
AUTHOR:          Edward Mulrane
CLASS:           Scientific Applications Programming: ICSP-319, Section 2
MODULE INTENT:    Finds the approximate values of a root for an equation using
                  the Bisection, Secant, and Newton's methods.
INPUT DATA:      None
OUTPUT DATA:     The value for a root of the equation
                   $35x^6 + 36x^5 + 23x^4 - 13x^3 - 12x^2 + x + 1 = 0$ 
                  using the three methods, as well as the number of iterations
                  for each, and the method with the fewest iterations.
*/

/*-----
Include Section
-----*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*-----
Preprocessor Section
-----*/

#define EPS      1e-6
#define L_END    0.4
#define R_END    1.0
#define INIT_1   1.0
#define INIT_2   2.0

/*-----
Structure Declarations
-----*/

enum RootMethod{ BISECTION_M, SECANT_M, NEWTONS_M };

/*-----
Function Prototypes
-----*/

void bisect( double left, double right, double *approx, int *iterations,
             double epsilon );

void secant( double x1, double x2, double *approx, int *iterations,
             double epsilon );

void newton( double x1, double *approx, int *iterations, double epsilon );

double f( double x );

double df( double x );

/*-----
Global Variables
-----*/

/*----- main program -----*/

int main( )

```

```

{
    int i;
    double n_approx;
    int num_iter, small_iter;
    enum RootMethod small_method;

    printf( " Finding An Approximate Root Of A Nonlinear Equation\n" );
    for ( i = 0; i < 80; i++ )
        putchar( '-' );
    putchar( '\n' );

    bisect( L_END, R_END, &n_approx, &num_iter, EPS );
    small_method = BISECTION_M;
    small_iter = num_iter;
    printf( " Bisection Method Root = %f Number of iterations = %d\n",
            n_approx, num_iter );

    secant( INIT_1, INIT_2, &n_approx, &num_iter, EPS );
    if ( num_iter <= small_iter ) {
        small_method = SECANT_M;
        small_iter = num_iter;
    }
    printf( " Secant Method Root = %f Number of iterations = %d\n",
            n_approx, num_iter );

    newton( R_END, &n_approx, &num_iter, EPS );
    if ( num_iter <= small_iter ) {
        small_method = NEWTONS_M;
        small_iter = num_iter;
    }
    printf( " Newton's Method Root = %f Number of iterations = %d\n",
            n_approx, num_iter );

    for ( i = 0; i < 80; i++ )
        putchar( '-' );
    putchar( '\n' );
    printf( "\n Method Using The Smallest Number of Iterations is: " );
    switch ( small_method ) {
    case BISECTION_M:
        printf( "Bisection Method\n" );
        break;
    case SECANT_M:
        printf( "Secant Method\n" );
        break;
    case NEWTONS_M:
        printf( "Newton's Method\n" );
        break;
    }

    return EXIT_SUCCESS;
}

/*----- function -----
NAME:      bisect
PURPOSE:   Computes an approximate root value for f(x) using the Bisection
           Method.
ARGUMENTS:
IN:        left: The left x value for the interval

```



```

right: The right x value for the interval
epsilon: The value of epsilon to use in the approximation
approx: The final root approximation
iterations: The number of iterations in finding the value
of approx.
CALLS:
    fabs (math.h), f
void bisect( double left, double right, double *approx, int *iterations,
            double epsilon )
{
    double x1, xr, xm;
    int count;

    for ( x1 = left, xr = xm = right, count = 0; fabs( f( xm ) ) > epsilon;
          count++ ) {
        if ( f( x1 ) * f( xm ) < 0 )
            xr = xm;
        else
            x1 = xm;
        xm = ( x1 + xr ) / 2;
    }

    *approx = xm;
    *iterations = count;

    return;
}

/*----- function -----*/
NAME:      secant
PURPOSE:   Computes an approximate root value for f(x) using the Secant
            Method.
ARGUMENTS:
    IN:
        x1: The first initial approximation
        x2: The second initial approximation
    epsilon: The value of epsilon to use in the approximation
    approx: The final root approximation
    iterations: The number of iterations in finding the value
               of approx.
    CALLS:
        fabs (math.h), f
void secant( double x1, double x2, double *approx, int *iterations,
            double epsilon )
{
    double xn1, xn2, xtemp;
    int count;

    for ( xn1 = x1, xn2 = x2, count = 0; fabs( f( xn2 ) ) > epsilon;
          count++ ) {
        xtemp = xn2 - ( f( xn2 ) * ( xn2 - xn1 ) ) / ( f( xn2 ) -
            f( xn1 ) );
        xn1 = xn2;
        xn2 = xtemp;
    }

    *approx = xn2;
    *iterations = count;

    return;
}

```

```

/*----- function -----*/
NAME:      newton
PURPOSE:   Computes an approximate root value for f(x) using Newton's
            Method.
ARGUMENTS:
    IN:
        x1: The initial approximation
    epsilon: The value of epsilon to use in the approximation
    approx: The final root approximation
    iterations: The number of iterations in finding the value
               of approx.
    CALLS:
        fabs (math.h), f, df
void newton( double x1, double *approx, int *iterations, double epsilon )
{
    double xn;
    int count;

    for ( xn = x1, count = 0; fabs( f( xn ) ) > epsilon; count++ )
        xn = xn - ( f( xn ) / df( xn ) );

    *approx = xn;
    *iterations = count;

    return;
}

/*----- function -----*/
NAME:      f
ARGUMENTS:
    IN:
        x: the value of x to compute the function for
    RETURNS:
        The result of the function
    CALLS:
        pow (math.h)
double f( double x )
{
    return 36 * pow( x, 6 ) + 36 * pow( x, 5 ) + 23 * pow( x, 4 ) -
        13 * pow( x, 3 ) - 12 * pow( x, 2 ) + x + 1;
}

/*----- function -----*/
NAME:      df
ARGUMENTS:
    IN:
        x: the value of x to compute the first derivative of the
            function for
    RETURNS:
        The result of the first derivative
    CALLS:
        pow (math.h)
double df( double x )
{
    return 216 * pow( x, 5 ) + 180 * pow( x, 4 ) + 92 * pow( x, 3 ) -
        39 * pow( x, 2 ) - 24 * x + 1;
}

```

```

/*
MODULE NAME: solve.c
AUTHOR: Edward Mulrane
CLASS: Scientific Applications Programming: ICSP-319, Section 2
MODULE INTENT: Reads the coefficient and exponents values for a polynomial
               function, along with interval values, from a file and
               computes the root approximation for the function using
               the bisection method.
INPUT DATA: Name of the data file as a command line argument. File
              contains the intervals for the root approximation, the
              number of coefficients/exponents, and the coefficient/exponent
              lists for the function.
OUTPUT DATA: The interval for the root approximation, the polynomial
              function printed in conventional format, and the approximate
              value of the root (or any appropriate error message.)
*/

/*-----
Include Section
-----*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/*-----
Preprocessor Section
-----*/

#define EPS 1e-6

/*-----
Structure Declarations
-----*/

struct BISECT_INFO {
    float endpoints[ 2 ];
    int num;
    float coeffs[ 10 ];
    int exponents[ 10 ];
};

/*-----
Function Prototypes
-----*/

void print_function( const struct BISECT_INFO *data );
double bisect( const struct BISECT_INFO *data, double epsilon );
double f( const struct BISECT_INFO *data, double x );

/*----- main program -----*/

int main( int argc, char *argv[] )
{
    FILE *input_file;
    struct BISECT_INFO data_read;

    if ( argc != 2 ) {
        fprintf( stderr, "\nUsage: %s <inputfile>\n", argv );
        fprintf( stderr, "\n**** Program terminated ****\n" );
    }

```

```

    exit( EXIT_FAILURE );
}
else {
    int count = 0;
    char c;
    float *coeff_ptr = data_read.coeffs;
    int *exp_ptr = data_read.exponents;

    input_file = fopen( *( argv + 1 ), "r" );

    if ( input_file == NULL ) {
        fprintf( stderr, "\nNo such file or directory\n" );
        fprintf( stderr, "\n**** Program terminated ****\n" );
        exit( EXIT_FAILURE );
    }

    fscanf( input_file, "%f", &data_read.endpoints,
            &data_read.endpoints + 1 );
    fscanf( input_file, "%d", &data_read.num );

    if ( data_read.num <= 2 || data_read.num >= 10 ) {
        fprintf( stderr, "\nNumber of coefficients must " );
        fprintf( stderr, "be between 3 and 9 inclusive\n" );
        fprintf( stderr, "\n**** Program terminated ****\n" );
        exit( EXIT_FAILURE );
    }

    while (getc( input_file ) != '\n' )
        ;

    while ( c = getc( input_file ) ) {
        if ( c == '\n' )
            break;
        else if ( isspace( c ) )
            continue;
        else {
            ungetc( c, input_file );
            fscanf( input_file, "%f", &coeff_ptr++ );
            count++;
        }
    }

    if ( count != data_read.num ) {
        fprintf( stderr, "\nIncorrect number of coefficient " );
        fprintf( stderr, "values\n" );
        fprintf( stderr, "\n**** Program terminated ****\n" );
        exit( EXIT_FAILURE );
    }

    count = 0;
    while ( c = getc( input_file ) ) {
        if ( c == '\n' )
            break;
        else if ( isspace( c ) )
            continue;
        else {
            ungetc( c, input_file );
            fscanf( input_file, "%d", &exp_ptr++ );
            count++;
        }
    }
}

```

```

if ( count != data_read.num ) {
    fprintf( stderr, "\nincorrect number of exponent " );
    fprintf( stderr, "values\n" );
    fprintf( stderr, "\n*** Program terminated ***\n" );
    exit( EXIT_FAILURE );
}

fclose( input_file );

printf( "\nThe given interval is [%2f , %2f]\n",
        data_read.endpoints[ 0 ], data_read.endpoints[ 1 ] );
printf( "\nThe given equation is:\n" );
print_function( &data_read );
printf( "\nApproximate value of the root is: %f\n",
        bisection( &data_read, EPS ) );
printf( "\n*** Program terminated ***\n" );
}

return EXIT_SUCCESS;
}

/*----- function -----
NAME:      print_function
PURPOSE:   Prints, in a conventional form, the polynomial function
           given in the structure parameter.
ARGUMENTS:
IN:        data: A pointer to the BISECT_INFO structure which
           contains the coefficients and exponents of the
           polynomial function
CALLS:     printf, putchar (stdio.h)
-----*/
void print_function( const struct BISECT_INFO *data )
{
    int i;

    for ( i = 0; i < data->num; i++ ) {
        if ( data->coeffs[ i ] != 0.0 ) {
            if ( i > 0 && data->coeffs[ i ] > 0.0 )
                printf( " + " );
            if ( data->exponents[ i ] > 1 )
                printf( "%1.f x^%d ", data->coeffs[ i ],
                        data->exponents[ i ] );
            else if ( data->exponents[ i ] == 1 )
                printf( "%1.f x ", data->coeffs[ i ] );
            else
                printf( "%1.f ", data->coeffs[ i ] );
        }
    }
    putchar( '\n' );
}

return;
}

/*----- function -----
NAME:      bisection
PURPOSE:   Computes an approximate root value for f(x) using the
           Bisection Method.
ARGUMENTS:
IN:        data: A pointer to the structure where the left and right

```

```

           endpoint information is stored.
           epsilon: The value of epsilon to use in the approximation
           The root approximation for the function.
           fabs (math.h), f
CALLS:
-----*/
double bisection( const struct BISECT_INFO *data, double epsilon )
{
    double xl, xr, xm;
    int count;

    xl = data->endpoints[ 0 ];
    xr = xm = data->endpoints[ 1 ];

    while ( fabs( f( data, xm ) ) > epsilon ) {
        if ( f( data, xl ) * f( data, xm ) < 0 )
            xr = xm;
        else
            xl = xm;

        xm = ( xl + xr ) / 2;
    }

    return xm;
}

/*----- function -----
NAME:      f
PURPOSE:   Computes the value of the function f stored in the structure
           parameter at the given x-value.
ARGUMENTS:
IN:        data: A pointer to the BISECT_INFO structure where the
           function information is stored.
           x: The x-value to compute the function at.
           The value of the function.
-----*/
double f( const struct BISECT_INFO *data, double x )
{
    int i;
    double sum = 0.0;

    for ( i = 0; i < data->num; i++ )
        sum += data->coeffs[ i ] * pow( x, data->exponents[ i ] );

    return sum;
}

```

```

/*
MODULE NAME:      integ.c
AUTHOR:          Edward Mulrane
CLASS:           Scientific Applications Programming: ICSP-319, Section 2
MODULE INTENT:   Finds the approximate value for the definite integral of f(x)
                  using Simpson's rule and the Trapezoidal rule.
INPUT DATA:     The number of subintervals for Simpson's rule and the
                  Trapezoidal rule, and whether or not to continue.
OUTPUT DATA:    The approximate values for the definite integral of f(x)
                  computed by both Simpson's rule and the Trapezoidal rule.
*/

/*-----
Include Section
-----*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

/*-----
Preprocessor Section
-----*/

#define LEFT_EP 0
#define RIGHT_EP 1

/*-----
Function Prototypes
-----*/

void simp( double left, double right, unsigned points, double *approx );
void trap( double left, double right, unsigned points, double *approx );
double f( double x );

/*----- main program -----*/

int main( )
{
    unsigned s_intervals, t_intervals;
    double s_result, t_result;

    printf( "Another approximation for the given integral? " );
    while ( getchar() == 'y' ) {
        while ( getchar() != '\n' )
            continue;

        printf( "\nEnter the number of intervals for Simpson's rule: " );
        scanf( "%u", &s_intervals );
        printf( "\nEnter the number of intervals for Trapezoidal " );
        printf( "rule: " );
        scanf( "%u", &t_intervals );

        while ( getchar() != '\n' )
            continue;

        simp( LEFT_EP, RIGHT_EP, s_intervals, &s_result );
        trap( LEFT_EP, RIGHT_EP, t_intervals, &t_result );

        printf( "\n\nSimpson's Rule\t\tValue: %f\n", s_result );
    }
}

```

```

        printf( "Trapezoidal Rule\t\tValue: %f\n", t_result );
        printf( "\n\nAnother approximation for the given integral? " );
    }

    printf( "\n\n***** Program terminated *****\n" );
    return EXIT_SUCCESS;
}

/*----- function -----
NAME:      simp
PURPOSE:   Computes the approximate value of the definite integral f( x )
            using Simpson's rule.
PRE:       points must be an even positive integer
ARGUMENTS:
IN:        left: Left endpoint of the interval (lower bounds of definite
            integral.)
            right: Right endpoint of the interval (upper bounds of definite
            integral.)
            points: Number of subintervals to compute the approximation
            with.
OUT:       approx: A pointer to where the final approximation value is to
            be copied to.
CALLS:     f, assert (assert.h)
void simp( double left, double right, unsigned points, double *approx )
{
    double h = ( right - left ) / points;
    double xn = left;
    int i;
    double sm = 0.0;

    assert( points % 2 == 0 );

    for ( i = 0; i <= points; i++ ) {
        if ( i == 0 || i == points )
            sm += f( xn );
        else if ( i % 2 )
            sm += 4 * f( xn );
        else
            sm += 2 * f( xn );

        xn += h;
    }

    sm *= h / 3;

    *approx = sm;
    return;
}

/*----- function -----
NAME:      trap
PURPOSE:   Computes the approximate value of the definite integral f( x )
            using the Trapezoidal rule.
ARGUMENTS:
IN:        left: Left endpoint of the interval (lower bounds of definite
            integral.)
            right: Right endpoint of the interval (upper bounds of definite
            integral.)

```

```

points: Number of subintervals to compute the approximation
with.
approx: A pointer to where the final approximation value is to
be copied to.
CALLS: f
-----*/
void trap( double left, double right, unsigned points, double *approx )
(
    double h = ( right - left ) / points;
    double xn = left;
    int i;
    double tn = 0.0;
    for ( i = 0; i <= points; i++ ) (
        if ( i == 0 || i == points )
            tn += 0.5 * f( xn );
        else
            tn += f( xn );
        xn += h;
    )
    tn *= h;
    *approx = tn;
    return;
}

/*----- function -----
NAME: f
PURPOSE: Returns the value of the function f(x) = 1 / ( 1 + x^2 ) at
the given x value
ARGUMENTS:
IN: x: x value for where the function is to be evaluated.
CALLS: pow (math.h)
-----*/
double f( double x )
(
    return 1.0 / ( 1 + pow( x, 2 ) );
)

```

```

/*
MODULE NAME: lag.c
AUTHOR: Edward Mulrane
CLASS: Scientific Applications Programming: ICSP-319, Section 2
MODULE INTENT: Performs lagrange interpolation on a set of given data points,
and returns the value of this approximating function at an x
value provided.
INPUT DATA: The number of points provided, these points as ordered pairs,
and the x value where the approximating lagrange polynomial
is to be evaluated.
OUTPUT DATA: The value of the approximating polynomial at the x value
provided.
*/

/*-----
Include Section
-----*/
#include <stdio.h>
#include <stdlib.h>

/*-----
Preprocessor Section
-----*/

#define ARRAY_MAX 20

/*-----
Function Prototypes
-----*/

void lagrange( int number, double x[], double y[], double point,
double *pvalue );

/*----- main program -----*/

int main( )
{
    int num_entered;
    double x_entered[ ARRAY_MAX ];
    double y_entered[ ARRAY_MAX ];
    double eval_point;
    double result;
    int i;

    printf( "Do you want to continue? " );
    while ( getchar() == 'y' ) {
        while ( getchar() != '\n' )
            ;

        printf( "\nNumber of data points that will be entered? " );
        scanf( "%d", &num_entered );
        printf( "\nEnter each data point on a new line in the " );
        printf( "format:\n" );
        printf( "x_value y_value\n" );

        for ( i = 0; i < num_entered; i++ )
            scanf( "%lf%lf", x_entered + i, y_entered + i );

        printf( "\nEnter the point at which the polynomial is to be " );
        printf( "evaluated? " );
        scanf( "%lf", &eval_point );

```

```

        while ( getchar() != '\n' )
            ;

        lagrange( num_entered, x_entered, y_entered, eval_point,
            &result );
        printf( "\nThe value of p(%f) is %f\n", eval_point, result );
        printf( "\nDo you want to continue? " );
    }

    return EXIT_SUCCESS;
}

/*----- function -----*/
NAME: lagrange
PURPOSE: Computes the lagrange approximating polynomial, and evaluates
it at a x value given.
ARGUMENTS:
IN:
    number: The number of interpolation points.
    x[]: An array of every x point provided.
    y[]: An array of every corresponding y value provided.
    point: The x value to evaluate the approx. function at.
    pvalue: A pointer to the location to place the result of
the evaluated function.
OUT:
    the evaluated function.

void lagrange( int number, double x[], double y[], double point,
double *pvalue )
{
    int i, j;
    double p = 0.0;

    for ( i = 0; i < number; i++ ) {
        double L = 1.0;

        for ( j = 0; j < number; j++ ) {
            if ( j == i )
                continue;
            L *= ( point - x[ j ] ) / ( x[ i ] - x[ j ] );
        }
        p += L * y[ i ];
    }
    *pvalue = p;
    return;
}

```

```

/*
MODULE NAME: matrix.c
AUTHOR: Brad Virkler
CLASS: sap section 2
MODULE INTENT: To find the product of two given matrices. The information
will be stored in two files. The three matrices will be dynamically
allocated. The program will output the two matrices to be
multiplied and the product of them. The columns of the first matrix must equal
the rows of the second matrix. If this condition is not met then the program
will terminate. If the one of the files are not found then the program will
terminate.
INPUT DATA: Two files containing the matrices to be multiplied
OUTPUT DATA: the two matrices and their product.
*/

/*-----
Include Section
-----*/
#include <stdio.h>
#include <stdlib.h>

/*-----
Function Prototypes
void mult( *int, *int, *int, int, int );
-----*/

/*----- function -----
NAME: mult
PURPOSE: to create the product of two matrices
PRE:
POST:
ARGUMENTS: a pointer to the first matrix, pointer to the second, a pointer
to return the final matrix and integers for the rows and columns of
the two matrices to be multiplied.
IN:
INOUT:
OUT:
RETURNS: none
CALLS:
-----*/
void mult( int *m1, int *m2, int *answer, int m1row, int m1col, int m2col ) {
    int i, j, k;
    for( i = 0; i < m1row; i++ ) {
        for( j = 0; j < m2col; j++ ) {
            for( k = 0; k < m1col; k++ ) {
                *( answer + i*m2col + j ) += *( m1 + i*m1col + k ) *
                    *( m2 + k*m2col + j );
            }
        }
    }

    /*----- main program -----
calls: mult, printf

```

```

-----*/
int main( int argc, char *argv[] ) {
    FILE *file1, *file2;
    int *matrix1, *matrix2, *answer, m1row, m1col, m2row, m2col, ansrow;
    int anscol, i;

    /* check and make sure that there are two arguments. If there aren't then
    print out the message and terminate. */
    if( argc != 3 ) {
        printf( "\nYou must supply two files for the matrices\n\n" );
        printf( "**** Program Terminated ****\n\n" );
        return -1;
    }

    /* open up the two files. If either one is not found then print out
    the message and terminate. */
    if( ( file1 = fopen( argv[1], "r" ) ) == NULL ) {
        printf( "\n%s: ", argv[1] );
        printf( "No such file of directory\n\n" );
        printf( "**** Program Terminated ****\n\n" );
        return -1;
    }

    if( ( file2 = fopen( argv[2], "r" ) ) == NULL ) {
        printf( "\n%s: ", argv[2] );
        printf( "No such file of directory\n\n" );
        printf( "**** Program Terminated ****\n\n" );
        return -1;
    }

    /* get the rows and columns for the two matrices to be multiplied. */
    fscanf( file1, "%d", &m1row, &m1col );
    fscanf( file2, "%d", &m2row, &m2col );
    ansrow = m1row;
    anscol = m2col;

    /* if the columns of the first matrix doesn't equal the rows of the second
    then print out the message and terminate. */
    if( m1col != m2row ) {
        printf( "\n%s%s", "Matrices are: ", m1row, " x ", m1col );
        printf( "%s%s", " and ", m2row, " x ", m2col );
        printf( " and cannot be multiplied\n\n" );
        printf( "**** Program Terminated ****\n\n" );
        return -1;
    }

    /* dynamically create the three matrices. */
    matrix1 = (int*)malloc( m1row*m1col*sizeof(int) );
    matrix2 = (int*)malloc( m2row*m2col*sizeof(int) );
    answer = (int*)malloc( m1row*m2col*sizeof(int) );

```

```

/* read the first matrix from the files. */
for( i = 0; i < m1row*m1col; i++ ) {
    fscanf( file1, "%d", matrix1 + i );
}

/* read the second matrix from the file. */
for( i = 0; i < m2row*m2col; i++ ) {
    fscanf( file2, "%d", matrix2 + i );
}

mult(matrix1, matrix2, answer, m1row, m1col, m2col );

/* print out the matrices */
printf( "\nthe matrices are: \n\n" );
for( i = 0; i < m1row*m1col; i++ ) {
    printf( "%5d ", *(matrix1 + i) );
    if( m1row == m1col || m1row < m1col ) {
        if( ( i + 1 ) % m1col == 0 ) {
            printf( "\n" );
        }
        else {
            if( ( i - 1 ) % m1col == 0 ) {
                printf( "\n" );
            }
        }
    }
}

printf( "\n\n*** Program Terminated ***\n\n" );
return (EXIT_SUCCESS);
}

}

/* read the first matrix from the files. */
printf( "\ntheir product is:\n\n" );
for( i = 0; i < ansrow*anscol; i++ ) {
    printf( "%5d ", *(answer + i) );
    if( ansrow == anscol || ansrow < anscol ) {
        if( ( i + 1 ) % anscol == 0 ) {
            printf( "\n" );
        }
        else {
            if( ( i - 1 ) % anscol == 0 ) {
                printf( "\n" );
            }
        }
    }
}

printf( "\n\n*** Program Terminated ***\n\n" );
return (EXIT_SUCCESS);
}

```


Appendix C

Benchmark Input Sets

G.4 1.0

36.0 36.0 23.0 -13.0 -12.0 1.0 1.0

6 5 4 3 2 1 0

integ.in

Y
10
10
Y
100
100
Y
1000
1000

lag.in

y	7	0.0000	-3.0000
		0.3000	-0.7420
		0.6000	2.1430
		0.9000	6.4520
		1.2000	14.5790
		1.5000	31.4800
		1.8000	65.6280
		1.09	
y	5	0.000	0.000
		1.000	0.569
		2.000	0.791
		3.800	0.224
		5.000	-0.185
		4.30	
n			

Appendix D

Example Simulator Output File

```

sim-tc: SimpleScalar/PISA Tool Set version 3.0 of November, 2000.
Copyright (c) 1994-2000 by Todd M. Austin. All Rights Reserved.
This version of SimpleScalar is licensed for academic non-commercial use only.

sim: command line: sim-tc -config advanced.cfg ../telecomm/FFT/fft 4 128

sim: simulation started @ Tue Nov 5 21:20:35 2002, options follow:

sim-outerorder: This simulator implements a very detailed out-of-order issue
superscalar processor with a two-level memory system and speculative
execution support. This simulator is a performance simulator, tracking the
latency of all pipeline operations.

# -config          # load configuration from a file
# -dumpconfig      # dump configuration to a file
# -h              # print help message
# -v             # verbose operation
# -d             # enable debug message
# -i             # start in Diite debugger
# -seed          1 # random number generator seed (0 for timer seed)
# -q            # initialize and terminate immediately
# -chkpt        <null> # restore EIO trace execution from <fname>
# -redir:sim     <null> # redirect simulator output to file (non-interactive only)
# -redir:prog    <null> # redirect simulated program output to file
# -nice          0 # simulator scheduling priority
# -max:inst      0 # maximum number of inst's to execute
# -fastfwd      0 # number of insts skipped before timing starts
# -ptrace       <null> # generate pipetrace, i.e., <fname>|stdout|stderr> <range>
# -fetch:ifsize 128 # instruction fetch queue size (in insts)
# -fetch:implat 3 # extra branch mis-prediction latency
# -fetch:speed  1 # speed of front-end of machine relative to execution co
re
# -bpred         MBPpred # branch predictor type (nottaken|taken|perfect|bimod|2l
ev|comb)
# -bpred:bimod  2048 # bimodal predictor config (<table size>)
# -bpred:2lev   1 1024 8 0 # 2-level predictor config (<llsize> <l2size> <hist_size>
<xor>)
# -bpred:comb   1024 # combining predictor config (<meta_table_size>)
# -bpred:ras    8 # return address stack size (0 for no return stack)
# -bpred:btb    512 4 # BTB config (<num_sets> <associativity>)
# -bpred:spec_update <null> # speculative predictors update in (ID|WB) (default
non-spec)
# -decode:width 16 # instruction decode B/W (insts/cycle)
# -issue:width  16 # instruction issue B/W (insts/cycle)
# -issue:inorder false # run pipeline with in-order issue
# -issue:wrongpath true # issue instructions down wrong execution paths
# -commit:width 16 # instruction commit B/W (insts/cycle)
# -ruu:size     512 # register update unit (RUU) size
# -lsq:size     256 # load/store queue (LSQ) size
# -cache:dll    d11:128:32:4:1 # 11 data cache config, i.e., (<config>|none)
# -cache:d1llat 1 # 11 data cache hit latency (in cycles)
# -cache:d12    u12:1024:64:4:1 # 12 data cache config, i.e., (<config>|none)
# -cache:d12lat 6 # 12 data cache hit latency (in cycles)
# -cache:i11    i11:1024:32:1:1 # 11 inst cache config, i.e., (<config>|d11|d12|none)
# -cache:i1llat 1 # 11 instruction cache hit latency (in cycles)
# -cache:i12    d12 # 12 instruction cache hit latency (in cycles)
# -cache:i12lat 6 # 12 instruction cache hit latency (in cycles)
# -cache:flush  false # flush caches on system calls
# -cache:icompress false # convert 64-bit inst addresses to 32-bit inst equivalen

```

```

ts
-memlat      18 2 # memory access latency (<first_chunk> <inter_chunk>)
-cmb:width   8 # memory access bus width (in bytes)
-itlb:itlb   16b:16:4096:4:1 # instruction TLB config, i.e., (<config>|none)
-tlb:dtlb   16b:32:4096:4:1 # data TLB config, i.e., (<config>|none)
-cf:lat     30 # inst/data TLB miss latency (in cycles)
-res:ialu    16 # total number of integer ALU's available
-res:imult   4 # total number of integer multiplier/dividers available
-res:imemport 8 # total number of memory system ports available (to CPU)
-res:fpalu   16 # total number of floating point ALU's available
-res:fpmult  4 # total number of floating point multiplier/dividers available
-flable
# -pcstat    <null> # profile stat(s) against text addr's (mult uses ok)
# -bugcompat false # operate in backward-compatible bugs mode (for testing
only)
# -tc:scheme advanced # trace cache scheme
# -tc:opts  n:16,m:3,size:256,assoc:1,use_sts:0,use_alterate_fill:0,use_pconside
r:0,use_reassoc:0,use_renam:0,use_fst:1,fst_size:256,fst_assoc:1,fst_bias:2 # trace
cache options (scheme dependent)
# -tc:trace <null> # generate trace cache log, i.e., <fname>|stdout|stderr>
<range>
# -tc:mbp     MPag # multiple branch predictor type
# -tc:mbp_width 3 # multiple branch predictor bandwidth
# -tc:mbp_pat_width 10 # width of first level lookup structure (PAT) in bits
# -tc:mbp_pat_size 1024 # size of first level lookup structure (PAT) in # of ent
ries
# -tc:disable false # disable the trace cache
# -bbt:scheme full # branch bias table scheme
# -bbt:threshold 32 # bias table threshold value
# -bbt:size      128 # bias table size
# -bbt:assoc     4 # bias table associativity

```

Pipetrace range arguments are formatted as follows:

```
((@|<start>):((@|<+>)<end>))
```

Both ends of the range are optional, if neither are specified, the entire execution is traced. Ranges that start with an '@' designate an address range to be traced, those that start with an '#' designate a cycle count range. All other range values represent an instruction count range. The second argument, if specified with a '+', indicates a value relative to the first argument, e.g., 1000:+100 == 1000:1100. Program symbols may be used in all contexts.

Examples:

```

-ptrace FOO.trc #0:#1000
-ptrace BAR.trc #2000:
-ptrace BLAH.trc :1500
-ptrace UXXE.trc :
-ptrace FOOBAR.trc #main:+278

```

Branch predictor configuration examples for 2-level predictor:

```

Configurations: N, M, W, X
N # entries in first level (# of shift register(s))
W # width of shift register(s)
M # entries in 2nd level (# of counters, or other FSM)
X (yes=1/no=0) xor history and address for 2nd level index
Sample predictors:
GAg . 1, W, 2~W, 0
GAP . 1, W, M (M > 2~W), 0
PAG . N, W, 2~W, 0
PAP . N, W, M (M == 2^(N+W)), 0

```

gshare : 1, W, 2^W, 1

Predictor 'comb' combines a bimodal and a 2-level predictor.

The cache config parameter <config> has the following format:

<name>:<nsets>:<bsize>:<assoc>:<repl>

<name> - name of the cache being defined
 <nsets> - number of sets in the cache
 <bsize> - block size of the cache
 <assoc> - associativity of the cache
 <repl> - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-random

Examples: -cache:dl1 dl1:4096:32:1:1
 -dtlb dtlb:128:4096:32:r

Cache levels can be unified by pointing a level of the instruction cache hierarchy at the data cache hierarchy using the "dl1" and "dl2" cache configuration arguments. Most sensible combinations are supported, e.g.,

A unified l2 cache (il2 is pointed at dl2):
 -cache:il1 il1:128:64:1:1 -cache:il2 dl2
 -cache:dl1 dl1:256:32:1:1 -cache:dl2 ul2:1024:64:2:1

Or, a fully unified cache hierarchy (il1 pointed at dl1):
 -cache:il1 dl1
 -cache:dl1 ul1:256:32:1:1 -cache:dl2 ul2:1024:64:2:1

AdvancedTrace:::n

16 # trace size (n)

AdvancedTrace:::m

3 # number of trace blocks (m)

AdvancedTrace:::index_size

8 # trace index size

AdvancedTraceGenerator:::n

256 # trace cache size (in segments)

AdvancedTraceGenerator:::assoc

1 # trace cache assoc.

AdvancedTraceGenerator:::(size)

286208 # trace cache size (in bits)

MPAgPredictor:::width

3 # predictor bandwidth

MPAgPredictor:::pat_size

1024 # pat size

MPAgPredictor:::pat_width

10 # pat width

MPAgPredictor:::(size)

1024 # number of pht entries

sim: ** starting performance simulation **

sim: ** simulation statistics **

sim_num_insn 2691359 # total number of instructions committed
 sim_num_refs 642294 # total number of loads and stores committed
 sim_num_loads 390711 # total number of loads committed
 sim_num_stores 251583 0000 # total number of stores committed
 sim_num_branches 434563 # total number of branches committed
 sim_elapsed_time 393 # total simulation time in seconds

sim_inst_rate	6848.2417	# simulation speed (in insts/sec)
sim_total_insn	6434488	# total number of instructions executed
sim_total_refs	1432874	# total number of loads and stores executed
sim_total_loads	877296	# total number of loads executed
sim_total_stores	555578.0000	# total number of stores executed
sim_total_branches	1012410	# total number of branches executed
sim_cycle	950474	# total simulation time in cycles
sim_ipc	2.8316	# instructions per cycle
sim_cpi	0.3532	# cycles per instruction
sim_exec_BW	6.7698	# total instructions (mjs-spec + committed) per cy
cle		
sim_ipb	6.1933	# instruction per branch
ifq_count	17757200	# cumulative IFQ occupancy
ifq_fcourt	60867	# cumulative IFQ full count
ifq_occupancy	18.6825	# avg IFQ occupancy (insn's)
ifq_rate	6.7698	# avg IFQ dispatch rate (insn/cycle)
ifq_latency	2.7597	# avg IFQ occupant latency (cycle's)
ifq_full	0.0640	# fraction of time (cycle's) IFQ was full
RUU_count	140865033	# cumulative RUU occupancy
RUU_fcourt	84550	# cumulative RUU full count
RUU_occupancy	148.2050	# avg RUU occupancy (insn's)
RUU_rate	6.7698	# avg RUU dispatch rate (insn/cycle)
RUU_latency	21.8922	# avg RUU occupant latency (cycle's)
RUU_full	0.0890	# fraction of time (cycle's) RUU was full
LSQ_count	27243570	# cumulative LSQ occupancy
LSQ_fcourt	612	# cumulative LSQ full count
LSQ_occupancy	28.6631	# avg LSQ occupancy (insn's)
LSQ_rate	6.7698	# avg LSQ dispatch rate (insn/cycle)
LSQ_latency	4.2340	# avg LSQ occupant latency (cycle's)
LSQ_full	0.0006	# fraction of time (cycle's) LSQ was full
sim_slip	90882225	# total number of slip cycles
avg_sim_slip	33.7682	# the average slip between issue and retirement
mbpred_lookups	1133283	# total number of bpred lookups
mbpred_updates	434563	# total number of updates
mbpred_addr_hits	389539	# total number of address-predicted hits
mbpred_dir_hits	402103	# total number of direction-predicted hits (includ
es addr-hits)		
mbpred_misses	32460	# total number of misses
mbpred_jr_hits	36330	# total number of address-predicted hits for JR's
mbpred_jr_seen	48941	# total number of JR's seen
mbpred_jr_non_ras_hits_pp	218	# total number of address-predicted hits for no
n-RAS JR's		
mbpred_jr_non_ras_seen_pp	265	# total number of non-RAS JR's seen
mbpred_bpred_addr_rate	0.8964	# branch address-prediction rate (i.e., addr-hits/upd
ates)		
mbpred_bpred_dir_rate	0.9253	# branch direction-prediction rate (i.e., all-hits/up
dates)		
mbpred_bpred_jr_rate	0.7423	# JR address-prediction rate (i.e., JR addr-hits/JRs
seen)		
mbpred_bpred_jr_non_ras_rate_pp	0.8226	# non-RAS JR addr-pred rate (ie, non-RAS JR
hits/JRs seen)		
mbpred_retstack_pushes	106704	# total number of address pushed onto ret-addr sta
ck		
mbpred_retstack_pops	100145	# total number of address popped off of ret-addr s
tack		
mbpred_used_ras_pp	48676	# total number of RAS predictions used
mbpred_ras_hits_pp	36112	# total number of RAS hits
mbpred_ras_rate_pp	0.7419	# RAS prediction rate (i.e., RAS hits/used RAS)
ill_accesses	1900265	# total number of accesses
ill_hits	1881672	# total number of hits
ill_misses	18593	# total number of misses

```

17665 # total number of replacements
i1l.replacements 0 # total number of writebacks
i1l.writebacks 0 # total number of invalidations
i1l.invalidations 0.0098 # miss rate (i.e., misses/ref)
i1l.miss_rate 0.0093 # replacement rate (i.e., repls/ref)
i1l.repl_rate 0.0000 # writeback rate (i.e., wrbks/ref)
i1l.wb_rate 0.0000 # invalidation rate (i.e., invs/ref)
i1l.inv_rate 727061 # total number of accesses
d1l.accesses 726101 # total number of hits
d1l.hits 960 # total number of misses
d1l.misses 448 # total number of replacements
d1l.replacements 408 # total number of writebacks
d1l.writebacks 0 # total number of invalidations
d1l.invalidations 0.0013 # miss rate (i.e., misses/ref)
d1l.miss_rate 0.0006 # replacement rate (i.e., repls/ref)
d1l.repl_rate 0.0006 # writeback rate (i.e., wrbks/ref)
d1l.wb_rate 0.0000 # invalidation rate (i.e., invs/ref)
d1l.inv_rate 19961 # total number of accesses
u12.accesses 18700 # total number of hits
u12.hits 1261 # total number of misses
u12.misses 0 # total number of replacements
u12.replacements 0 # total number of writebacks
u12.writebacks 0 # total number of invalidations
u12.invalidations 0.0632 # miss rate (i.e., misses/ref)
u12.miss_rate 0.0000 # replacement rate (i.e., repls/ref)
u12.repl_rate 0.0000 # writeback rate (i.e., wrbks/ref)
u12.wb_rate 0.0000 # invalidation rate (i.e., invs/ref)
u12.inv_rate 1900265 # total number of accesses
i1b.accesses 1900245 # total number of hits
i1b.hits 20 # total number of misses
i1b.misses 0 # total number of replacements
i1b.replacements 0 # total number of writebacks
i1b.writebacks 0 # total number of invalidations
i1b.invalidations 0.0000 # miss rate (i.e., misses/ref)
i1b.miss_rate 0.0000 # replacement rate (i.e., repls/ref)
i1b.repl_rate 0.0000 # writeback rate (i.e., wrbks/ref)
i1b.wb_rate 0.0000 # invalidation rate (i.e., invs/ref)
i1b.inv_rate 824376 # total number of accesses
d1b.accesses 824360 # total number of hits
d1b.hits 16 # total number of misses
d1b.misses 0 # total number of replacements
d1b.replacements 0 # total number of writebacks
d1b.writebacks 0 # total number of invalidations
d1b.invalidations 0.0000 # miss rate (i.e., misses/ref)
d1b.miss_rate 0.0000 # replacement rate (i.e., repls/ref)
d1b.repl_rate 0.0000 # writeback rate (i.e., wrbks/ref)
d1b.wb_rate 0.0000 # invalidation rate (i.e., invs/ref)
d1b.inv_rate 0 # total non-speculative bogus addresses seen (debu
sim.invalid_addrs g var)
ld.text_base 0x00400000 # program text (code) segment base
ld.text_size 82560 # program text (code) size in bytes
ld.data_base 0x10000000 # program initialized data segment base
ld.data_size 9300 # program init'ed 'data' and uninit'ed 'bss' siz
e in bytes
ld_stack_base 0x7fffc000 # program stack segment base (highest address in s
tack)
ld_stack_size 16384 # program initial stack size
ld_prog_entry 0x00400140 # program entry point (initial PC)
ld_environ_base 0x7fff8000 # program environment base address
ld_target_big_endian 0 # target executable endian-ness, non-zero if big e
ndian

```

```

mem.page_count 33 # total number of pages allocated
mem.page_mem 132k # total size of memory pages allocated
mem.ptab_misses 696 # total first level page table misses
mem.ptab_accesses 13634879 # total page table accesses
mem.ptab_miss_rate 0.0001 # first level page table miss rate
sent_to_ifq 7203587 # number of inst. sent to the IFQ
fetch_cycle 680513 # num of cycles data was sent to the IFQ
sim.fetch_rate 10.5855 # fetch bandwidth
TraceCache::trace_too_big 33821 # not enough IFQ space
TraceCache::wrong_starting_pc 0 # trace starting pc != fetch pc
TraceCache::branches_fetched 574761 # branches fetched from TC
TraceCache::promoted_fetched 430989 # promoted branches fetched from TC
TraceCache::promoted_percentage 0.749858 # percentage of promoted branches
TraceCache::reassoc_fetched 0 # reassoc'ed insts. fetched from TC
AdvancedTrace::promoted_count 195928 # number of promoted branches encountered
AdvancedTrace::reassoc_count 0 # number of reassoc'ed instructions encountered
AdvancedTraceGenerator::gets 717032 # accesses
AdvancedTraceGenerator::hits 415145 # hits
AdvancedTraceGenerator::hit_rate 0.578977 # trace cache hit rate
AdvancedTraceGenerator::mru_hits 415145 # mru hits within the set
AdvancedTraceGenerator::mid_hits 0 # mid hits within the set
AdvancedTraceGenerator::lru_hits 0 # lru hits within the set
AdvancedTraceGenerator::unique_added 54091 # unique traces added
AdvancedTraceGenerator::alias_count 53835 # number of valid traces replaced
AdvancedTraceGenerator::ends_jmmax 1846 # m count ends trace
AdvancedTraceGenerator::ends_mmax 42141 # n count ends trace
AdvancedTraceGenerator::ends_indir 9351 # indirect jump ends trace
AdvancedTraceGenerator::avg_length 14.3444 # average segment length
AdvancedTraceGenerator::utilized 100 # trace cache percent utilized (valid)
AdvancedFillUnit::new_traces 2450676 # new traces created
AdvancedFillUnit::drains 22515 # explicit fill unit drains
MPAgPredictor::update_count 385353 # number of predictor updates
MPAgPredictor::correct[0] 110774 # correct predictions, level 0
MPAgPredictor::mispred[0] 17201 # mispredictions, level 0

```



```

MPAgPredictor::(pred_rate)[ 0]
0.865591 # prediction rate (dir), level 0
MPAgPredictor::update_count
385353 # number of predictor updates
MPAgPredictor::_correct[ 1]
13495 # correct predictions, level 1
MPAgPredictor::_mispred[ 1]
1984 # mispredictions, level 1
MPAgPredictor::(pred_rate)[ 1]
0.871826 # prediction rate (dir), level 1
MPAgPredictor::update_count
385353 # number of predictor updates
MPAgPredictor::_correct[ 2]
1314 # correct predictions, level 2
MPAgPredictor::_mispred[ 2]
274 # mispredictions, level 2
MPAgPredictor::(pred_rate)[ 2]
0.827456 # prediction rate (dir), level 2
MPAgPredictor::_btb_miss_correct
29378 # correct predictions, btb miss
MPAgPredictor::_btb_miss_mispred
351 # mispredictions, btb miss
MPAgPredictor::(btb_miss_pred_rate)
0.988193 # prediction rate (dir), btb miss
MPAgPredictor::_promoted_correct
128876 # correct predictions, promoted branches
MPAgPredictor::_promoted_mispred
12386 # mispredictions, promoted branches
MPAgPredictor::(promoted_pred_rate)
0.912319 # prediction rate (dir), promoted branches
MPAgPredictor::tot_correct
283837 # total correct predictions
MPAgPredictor::tot_mispred
32196 # total mispredictions
MPAgPredictor::(pred_rate)
0.898125 # total prediction rate (dir)
CoreFetchUnit::_btb_ops_retired
385353 # retired count of btb ops
CoreFetchUnit::_indir_ops_retired
49210 # retired count of indirect jumps

```