UNIVERSITY OF CALIFORNIA, SAN DIEGO

Hardware Optimizations Enabled by a Decoupled Fetch Architecture

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy from the Department of Computer Science and Engineering

by

Glenn Reinman

Committee in charge:

Professor Brad Calder, Chairperson
Professor Todd Austin
Professor Paul Chau
Professor Matt Farrens
Professor Andrew Kahng
Professor Dean Tullsen

2001

Dedications

I dedicate this work to the many friends and family who have made such a difference in my life and who have encouraged me throughout my career.

To my parents, who have always offered me unconditional support through all of my endeavors with their unbridled enthusiasm. They have managed to tolerate my often moody disposition and fiercely independent nature with compassion and understanding. Without them (and Domino), this work would not have been possible.

To the many faculty who have illuminated my path towards the PhD. To my advisor, Dr. Brad Calder, for many years of putting up with me and for pushing me that extra mile or twenty (and for teaching me the meaning of the word *both*). To Dr. Dean Tullsen, for your expert advice and encouragement. To Dr. Todd Austin, for showing me the lighter side of academia. To Dr. Norm Jouppi, for your patience and for a great opportunity. To Dr. Rich Wolski, for numerous sessions of expert advising – even on nonacademic issues. To Dr. Geoff Voelker, for sharing his experience and advice. To Dr. Matt Farrens, for serving on my committee and showing me that Professors can have balance in their lives too. To Dr. Andrew Kahng for filling in on my committee at the last moment.

Thanks to all of my friends in the architecture lab. To Lori, who has been my great friend since we started the PhD program so many years ago. I will always appreciate your laughter, kindness, advice, perspective, and compassion – not to mention your volleyball skills. You are the sister I never had. To Suleyman, for his incredible patience for bad jokes and abuse. Thanks to both you and Sule for BBQs, biking, blading, and hiking. It is rare to find someone so good-natured and honest. To Tim, my co-conspirator, partner in crime, and author of wacky stuff. Thanks for introducing me to hockey – and for countless schemes and Suley-snares. To Chandra (Dr. Krintz that is), for always going first and for her incredible ability and drive. And of course, for passing on her thesis formatting wizardry. To John, for his sense of humor and for hours of hard work administering machines. Remember your martial arts training well. To Barbara, who's compassionate nature and contagious laughter affected the whole lab (and to Hillary for her enthusiastic welcomes). To Beth, master coordinator and networker. To Eric, for asking questions. To Jeff, for bringing laughter to the lab. To Jamison, for his unique perspective and Intel advice. To Erez, for the gift of music. To Wei, for his vast Intel knowledge. To Floria, for her bravery in a new country.

Thanks to the staff at UCSD for countless hours of help and patience. To Julie Conner, who somehow manages to stay sane and amiable while coordinating this department. To CSE support for answering many, many questions. To the administrative staff for their ready smiles and patience.

And, of course, my thanks to my best and closest friend Taraneh. You have been a source of strength and compassion in my life, and have taught me a great deal about myself. The real future work in my life begins with the next chapter that we will write and share together. To your family, for welcoming me into their culture and home, and for many, many trips to Javan.

I met a traveller from an antique land

Who said: 'Two vast and trunkless legs of stone

Stand in the desert. Near them, on the sand,

Half sunk, a shattered visage lies, whose frown,

And wrinkled lip, and sneer of cold command,

Tell that its sculptor well those passions read

Which yet survive, stamped on these lifeless things,

The hand that mocked them and the heart that fed.

And on the pedestal these words appear –

"My name is Ozymandias, king of kings:

Look on my works, ye Mighty, and despair!"

Nothing beside remains. Round the decay

Of that colossal wreck, boundless and bare

The lone and level sands stretch far away.'

- Percy Bysshe Shelley

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Acknowledgments

The text of Chapter VII is in part a reprint of the material as it appears in a summer internship report at COMPAQ's Western Research Laboratory under the direction of Norm Jouppi. The report and model can be obtained at:

http://www.research.digital.com/wrl/people/jouppi/CACTI.html

The dissertation author was the primary researcher and author and the co-author listed on this publication ([74]) directed and supervised the research which forms the basis for Chapter VII.

The text of Chapters VIII, IX, and X are in part reprints of material as it appears in the IEEE Transactions on Computers. The dissertation author was the primary researcher and author and the co-authors listed on this publication ([73]) directed and supervised the research which forms the basis for Chapters VIII, IX, and X

The text of Chapter IX is in part a reprint of the material as it appears in the proceedings of the 26th Annual International Symposium on Computer Architecture. The dissertation author was the primary researcher and author and the co-authors listed on this publication ([70]) directed and supervised the research which forms the basis for Chapter IX.

The text of Chapter X is in part a reprint of the material as it appears in the proceedings of the 32nd International Symposium on Microarchitecture. The dissertation author was the primary researcher and author and the co-authors listed on this publication ([72]) directed and supervised the research which forms the basis for Chapter X.

VITA

September 1, 1974	Born, Miami, Florida
1996	B.S. in Computer Science and Engineering Massachusetts Institute of Technology
1996–1997	Teaching Assistant, University of California, San Diego
1997–2001	Research Assistant, University of California, San Diego
1998	Internship, Intel MRL, Oregon
1999	M.S. in Computer Science University of California, San Diego
2000	Internship, COMPAQ Western Research Lab Palo Alto
2001	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

“Optimizations Enabled by a Decoupled Front-End Architecture.” Authors: Glenn Reinman, Brad Calder, and Todd Austin. IEEE Transactions on Computing. April, 2001.

“A Comparative Survey of Load Speculation Architectures.” Authors: Brad Calder and Glenn Reinman. Journal of Instruction Level Parallelism. May, 2000.

“Fetch Directed Instruction Prefetching.” Authors: Glenn Reinman, Brad Calder, and Todd Austin. 32nd International Symposium on Microarchitecture, November 1999.

“Classifying Load and Store Instructions for Memory Renaming.” Authors: Glenn Reinman, Brad Calder, Dean Tullsen, Gary Tyson, and Todd Austin. ACM International Conference on Supercomputing, June 1999.

“A Scalable Front-End Architecture for Fast Instruction Delivery.” Authors: Glenn Reinman, Todd Austin, and Brad Calder. 26th International Symposium on Computer Architecture, May 1999.

“Selective Value Prediction.” Authors: Brad Calder, Glenn Reinman, and Dean Tullsen. 26th Annual International Symposium on Computer Architecture, May 1999.

"Predictive Techniques for Aggressive Load Speculation." Authors: Glenn Reinman and Brad Calder. 31st International Symposium on Microarchitecture, December 1998.

ABSTRACT OF THE DISSERTATION

Hardware Optimizations Enabled by a
Decoupled Front-End Architecture by

Glenn Reinman
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 2001
Professor Brad Calder, Chair

In the pursuit of instruction-level parallelism, significant demands are placed on a processor's instruction delivery mechanism. In order to provide the performance necessary to meet future processor execution targets, the instruction delivery mechanism must scale with the execution core. Attaining these targets is a challenging task due to I-cache misses, branch mispredictions, and taken branches in the instruction stream. Moreover, there are a number of hardware scaling issues such as wire latency, clock scaling, and energy dissipation that can impact processor design.

To address these issues, this thesis presents a fetch architecture that decouples the branch predictor from the instruction fetch unit. A Fetch Target Queue (FTQ) is inserted between the branch predictor and instruction cache. This allows the branch predictor to run far in advance of the address currently being fetched by the instruction cache. The decoupling enables a number of architectural optimizations including multi-level branch predictor design and fetch directed instruction prefetching.

A multi-level branch predictor design consists of a small first level predictor that can scale well to future technology sizes and larger higher level predictors that can provide capacity for accurate branch prediction.

Fetch directed instruction cache prefetching uses the stream of fetch addresses contained in the FTQ to guide instruction cache prefetching. By following the predicted fetch path, this technique provides more accurate prefetching than simply following a sequential fetch path.

Fetch directed prefetching using a contemporary set-associative instruction cache has some complexity and energy dissipation concerns. Set-associative caches provide a great deal of performance benefit, but dissipate a large amount of energy by blindly driving a number of associative ways. By decoupling the tag and data components of the instruction cache, a complexity effective and energy efficient scheme for fetch directed instruction cache prefetching can be enabled.

This thesis explores the decoupled front-end design and these related optimizations, and suggests future research directions.

Chapter I

Introduction

At a high level, a modern high-performance uniprocessor is composed of two processing engines: the *front-end processor* and the *execution core*. The front-end processor is responsible for fetching and preparing (*e.g.*, decoding, renaming, etc.) instructions for execution. The execution core orchestrates the execution of instructions and the retirement of their register and memory results to non-speculative storage. Typically, these processing engines are connected by a buffering stage of some form, *e.g.*, instruction fetch queues or reservation stations – the front-end acts as a producer, filling the connecting buffers with instructions for consumption by the execution core. This is shown in Figure I.1.

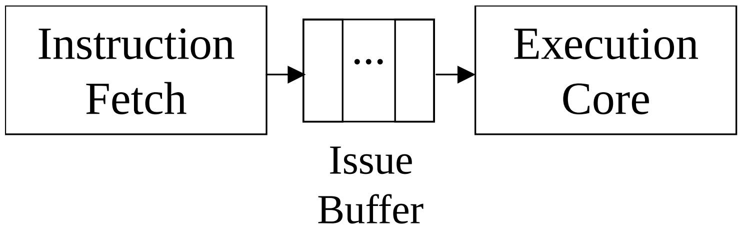


Figure I.1: Two processing engines: the processor pipeline at a high level. The instruction fetch unit prepares and decodes instructions and supplies them to the issue buffer. The execution core consumes instructions from the issue buffer and then orchestrates their execution and retirement. The instruction fetch unit is a fundamental bottleneck in the pipeline: the execution core can only execute instructions as fast as the instruction fetch unit can prepare them.

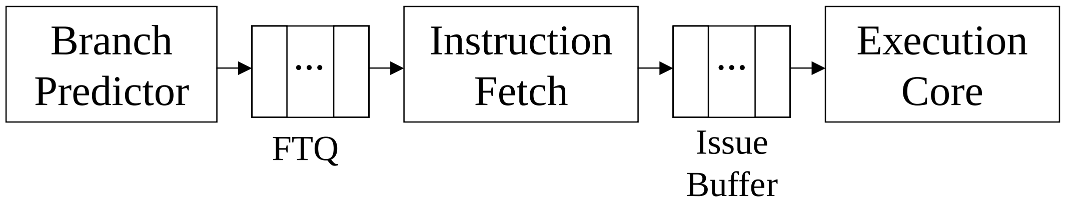


Figure I.2: The decoupled front-end design at a high level.
The fetch target queue (FTQ) buffers fetch addresses produced by the branch predictor. They are queued in the FTQ until they are consumed by the instruction fetch unit, which in turn produces instructions as in an ordinary pipeline. The FTQ allows the branch predictor to continue predicting in the face of an instruction cache miss. It also provides the opportunity for a number of optimizations, including multi-level branch predictor designs and fetch directed cache prefetching.

This producer/consumer relationship between the front-end and execution core creates a fundamental bottleneck in computing, *i.e.*, execution performance is strictly limited by fetch performance. The trend towards exploiting more ILP in execution cores works to place further demands on the rate of instruction delivery from the front-end. Without complementary increases in front-end delivery performance, more exploitation of ILP will only decrease functional unit utilization with little or no increase in overall performance.

In this thesis, we focus on improving the scalability and performance of the front-end by decoupling the branch predictor from the instruction cache. A *Fetch Target Queue* (FTQ) is inserted between the branch predictor and instruction cache, as seen in Figure I.2. The FTQ stores predicted fetch addresses from the branch predictor, later to be consumed by the instruction cache. The FTQ serves two primary functions: latency tolerance and fetch stream look-ahead.

Typically, the branch predictor and instruction cache work together in the processor pipeline. If the instruction cache misses or there are insufficient ports on the instruction cache to consume an entire branch prediction, the branch predictor must stall. The FTQ buffers fetch block predictions and can allow the branch predictor and the instruction cache to operate relatively independently and better tolerate latency in the instruction cache. Moreover, the ability of the FTQ to tolerate latency enables the design of a multilevel branch predictor. A multilevel branch predictor has a small, fast first level predictor along with multiple larger higher levels that have greater capacity and accuracy, but which take longer to access. This type of predictor can provide highly accurate predictions while keeping the structure's access time low, operating much like a multilevel cache hierarchy. This technique can be used with virtually any branch predictor, and may become even more useful as process technology sizes shrink and the access time for large structures grows [2, 60]. The FTQ allows the instruction cache to continue in the face of a first level miss in a multilevel branch predictor hierarchy – if there are sufficient fetch predictions stored in the FTQ.

The second primary function of the FTQ is to provide a look-ahead mechanism. The FTQ enables the branch predictor to run ahead of the instruction cache and provide a glimpse at the future stream of instruction fetch addresses. These addresses can then be used to guide a variety of PC-based predictors, such as instruction and data cache prefetchers, value predictors, and instruction reuse tables. In prior architectures, some of these structures are accessed after the decode stage, and may need to be quite large to provide high accuracy. This also means that the structures will likely have high latency. However, by bringing the predictor access before even the instruction cache is accessed, the processor can tolerate longer latency predictors. Alternatively, this look-ahead can enable intelligent multi-level predictors. Furthermore, the fetch address stream made available is no longer constrained by the number of ports on the instruction cache. The rate at which predictions can be made determines the rate at which fetch addresses are delivered to the FTQ, which can then be consumed by prediction mechanisms. A single ported instruction cache can only provide a cache block every cycle, but a high bandwidth branch predictor can provide several fetch block addresses each cycle. In this thesis, we investigate an instruction cache prefetching scheme that uses the FTQ to guide instruction cache prefetch into a fully-associative buffer. We investigate the use of different filtration methods to reduce the bus utilization of the prefetcher.

The remainder of the thesis is organized as follows. Chapter III looks at some of the hardware trends and pipeline dependency issues that motivate this work. Chapter IV reviews some of the relevant prior work in this area. The choice of metrics we use to evaluate the architectures in this thesis is described in Chapter V. Chapter VI describes the simulation and timing methodology and Chapter VII explores the changes made to the timing model for this thesis in more depth. In Chapter VIII we present the FTQ and the decoupled front-end it enables. The benefits and implementation of a multi-level branch predictor are described in Chapter IX. In Chapter X, we investigate the optimizations made possible by using the stream of fetch addresses stored in the FTQ. Chapter XII provides a summary and we conclude with future directions in Chapter XIII.

Chapter II

Basic Processor Pipeline

Processor pipelines vary greatly from one machine to another. The high-level pipeline we examine in this thesis is shown in Figure II.1. This pipeline has two major components, the front-end processor and the execution core.

II.A Front-End

The basic front-end design has three major components: the branch prediction hardware, the instruction cache, and the decode hardware. The front-end provides instructions to the issue buffer. The front-end of the machine tracks the current program counter (PC) of the instruction currently being fetched. This PC indexes into the branch prediction hardware and instruction cache, which collectively return a branch prediction and an instruction cache block – assuming that the branch predictor hardware can process a single branch per cycle and the instruction cache is single ported. The decode hardware takes the cache block and branch prediction and masks out instructions in the cache block that will not be executed. The instructions to be executed are placed in the issue buffer.

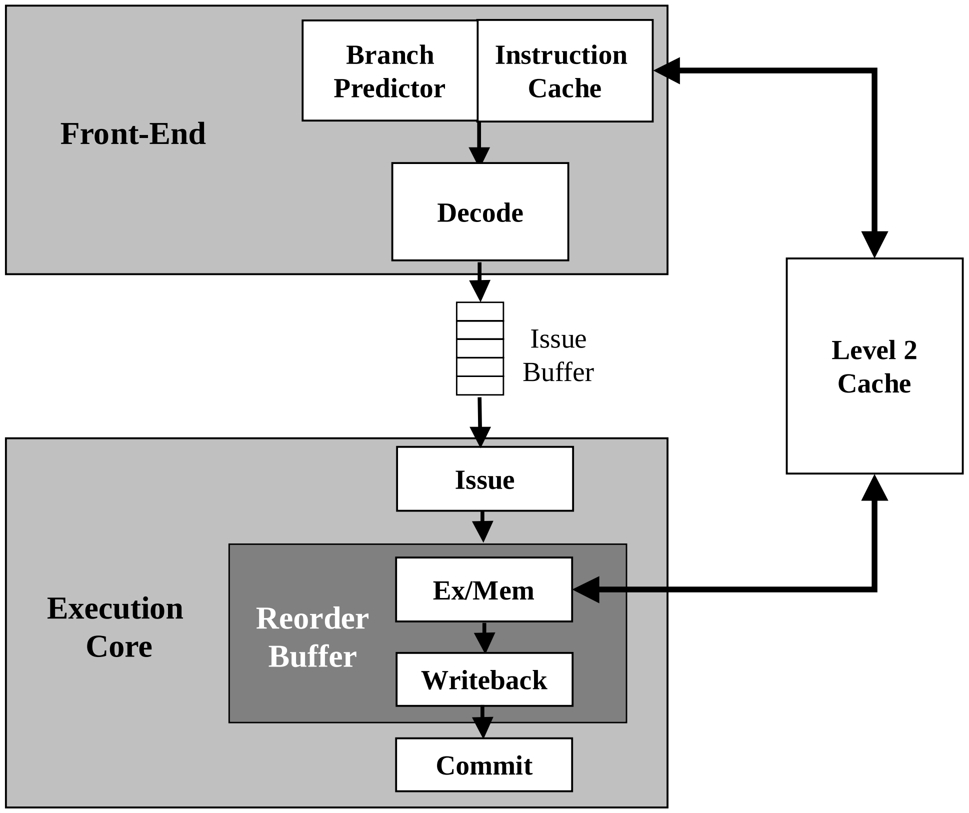


Figure II.1: Simplified Processor Pipeline
Simplified view of the front-end and execution core of the processor pipeline. Instructions are prepared by the front-end and placed in the issue buffer where they can then be consumed by the execution core. A data cache exists in the Execute/Memory (Ex/Mem) stage(s) of the pipeline and shares a second level cache with the instruction cache.

II.A.1 Branch Prediction Hardware

Branch prediction is an essential part of the processor pipeline. There are a variety of different branch types: conditional, unconditional, returns, and indirect branches. When a branch goes through the pipeline, its type is not known by the processor pipeline until the decode stage. In the case of a conditional branch, the direction (i.e. whether or not the branch is taken) is not known until the value that the branch is predicated on has been computed. Therefore, a processor without any form of branch prediction mechanism would need to stall the pipeline until a branch had been resolved in order to know the control path of the processor. This would seriously hamper the performance of the processor. Instead, if a control path is predicted, the processor can continue to insert instructions into the pipeline while the branch is being resolved. If the prediction is correct, no performance has been lost. If the predictor is incorrect, the pipeline must be flushed and restarted with the correct path. The penalty for an incorrect prediction is dependent on the depth of the pipeline, as this will impact the length of time it takes for the pipeline to refill with useful instructions. Therefore, as pipelines get deeper and deeper, the branch prediction hardware becomes even more critical to processor performance. Much research has gone into improving the accuracy of branch prediction hardware in an attempt to minimize the impact of mispredictions.

Many forms of branch prediction hardware exist. The prior work section of this thesis (Chapter IV) explores this in more detail. In the pipeline we simulate, there is a branch target predictor, a branch direction predictor, and a return address predictor. The branch direction predictor guesses whether a branch is taken or not, the branch target predictor guesses what address a taken branch goes to, and the return address predictor tracks the addresses of procedure calls so that subroutines can return to the original program control flow upon completion.

II.A.2 Instruction Cache

The PC provides a unique identifier of a given instruction for a particular program, but in order to be executed, the PC must be translated into an actual instruction. The program itself is loaded into a portion of memory, and the PC represents the address where a particular instruction is stored. The instruction cache provides a means of hiding the latency to memory for instruction addresses. The instruction cache contains a small subset of the total instruction addresses, and can dynamically swap these instruction addresses in and out of the cache to match the access pattern of the program. Instructions are moved in and out of the cache as part of a *cache block* of instructions. In our simulations, eight consecutive instruction addresses map to the same cache block.

The more instruction addresses that hit in the instruction cache, the more memory latency that is hidden. Therefore, it is critical to manage the contents of the instruction cache to maximize the hit rate of the cache (the ratio of instruction memory accesses that are actually in the cache). One part of this is determining *when* to bring in *what* blocks. Our base architecture only brings in new cache blocks *on demand*. This means that cache blocks will only be brought in from higher levels of the memory hierarchy when a miss occurs in the stage of the pipeline that accesses the instruction cache (so called demand fetches because the cache block is “fetched” from memory when the processor “demands” the block). The alternative to this is to *prefetch* cache blocks *before* they are needed (demanded). We will explore prior instruction cache prefetching strategies in Chapter IV and will introduce a novel cache prefetching scheme in Chapter X. Prefetches and demand fetches both bring new cache blocks into the instruction cache, and therefore require some means to choose which cache blocks need be removed from the cache to make room for the incoming blocks. Our base architecture makes use of an optimal least recently used (LRU) strategy [68] for cache replacement. This strategy orders cache blocks by their use, and will replace the cache block that has not been used in the longest amount of time. We examine a novel cache block replacement strategy in Chapter X.

II.B Execution Core

The execution core is responsible for executing and retiring instructions from the front-end. In our simulations, instructions are issued into a reorder buffer where they may then execute out-of-order and commit in-order. Figure II.1 shows a simplified execution core pipeline. The major structures of the execution core include the issue hardware, the reorder buffer, the functional units, the data cache (and memory management unit), and the commit hardware. This thesis focuses mainly on the structures of the front-end processor, but it is still important to clarify the operation of the structures of the execution core at a high level. A more detailed study of these issues can be found in [36].

II.B.1 Issue Hardware

If the reorder buffer is not full, the issue hardware takes instructions from the issue buffer and allocates space for them in the reorder buffer. Due to design and complexity constraints, the number of instructions that can be issued in a given cycle is limited. We examine an architecture that can issue 8 instructions in a single cycle. The Alpha 21264 issues 4 instructions in a single cycle [44].

II.B.2 Reorder Buffer

The reorder buffer allows instructions to execute out-of-order. Instructions issue in-order to the reorder buffer, but once their data dependencies have been met, they may execute whenever functional units are available. This technique allows the processor to more fully utilize functional unit resources and tolerate data dependence delays. In order to maintain correct processor state, instructions do not affect the register file or memory hierarchy – but rather write their results to the reorder buffer. When instructions issue, the issue hardware determines the input dependences of each instruction. If there is an input dependency from one instruction to another, and the instruction that is depended upon executes speculatively and writes a value to its reorder buffer entry, then the dependent instruction can also execute speculatively using that value.

II.B.3 Functional Units

The functional units of the execution core do the actual computation. There are a number of different functional unit types in our simulated processor: integer adders, load/store units, floating point adders, integer multiply/divide units, and floating point multiply/divide units. In practice, multiply and divide may use different functional units. Each functional unit has a different latency (see Chapter VI for the details on this), except for the multiply/divide unit which has two latencies (one for divide and one for multiply) and the load/store unit which depends on the data cache. Except for divide operations, all functional units are fully-pipelined, so a given functional unit can begin a new operation in each cycle.

II.B.4 Data Cache and MMU

The data cache and memory management unit (MMU) help to handle load/store instructions in the pipeline. The data cache hides the latency of data memory accesses in the same way that the instruction cache hides the latency of instruction memory accesses. It can also be limited by the number of ports, size, associativity, and block size of the structure - and it can be pipelined to be accessed over several cycles. The MMU schedules memory operations to use different levels of the memory hierarchy in the event of a memory hierarchy miss (*i.e.* on a data cache miss, the MMU can schedule the instruction to use the next level of the memory hierarchy).

II.B.5 Commit Hardware

Instructions in the reorder buffer remain in the buffer until committed. Upon commit, the instruction writes its result to the register file and its reorder buffer entry is deallocated. Because instructions execute out-of-order, it is possible for an instruction down a mispredicted branch path to finish executing before the branch misprediction is detected (for example, if the branch is dependent on a long latency instruction like a load or divide). To avoid disrupting the register file with data from an incorrect path, instructions in the reorder buffer commit *in-order*. Thus, instructions can only commit once they have completed execution and after all prior instructions in the reorder buffer have committed.

Chapter III

Motivation

The scalability of the front-end of the processor pipeline is complicated by a number of factors. These include the ability of devices to scale to future technology sizes, which are referred to as hardware concerns, and the inherent dependencies between the different stages of the front-end, which are referred to as latency concerns.

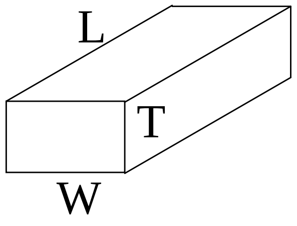
III.A Hardware Concerns

There are a number of hardware challenges that future microarchitects will need to face to design the next generation of microprocessors. One of the primary means of increasing the speed of microprocessors has been to scale the feature size of the current technology. As feature sizes continue to shrink, it has become evident that wire latencies are not scaling with transistor latencies. This trend has been termed the *interconnect scaling bottleneck* [8, 9]. Architects have been adding a variety of components to the chip, such as new speculative structures, additional pipelines, and larger caches, to improve processor IPC. But as the clock speed of the processor continues to increase, correspondingly dropping the cycle time of the processor, the addition of large memory structures to an architecture may be constrained by how fully they can be pipelined. Agarwal et al. [2] report that current processor designs will improve at best 12.5% per year over the next fourteen years due to hardware concerns. Finally, as the clock speed of the processor continues to grow, the amount of energy dissipated by the processor becomes more and more critical to the design of the microprocessor. In addition to limiting the battery life of mobile computers, the amount of energy dissipated by a chip can influence packaging and cooling of the chip as well as the layout of structures on the chip.

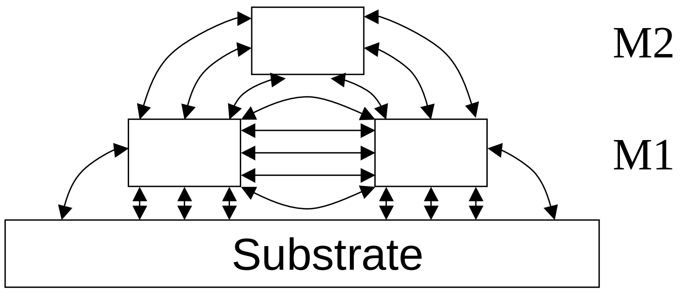
III.A.1 Wire Latency

Interconnect is expected to scale poorly due to the impact of resistative parasitics and parasitic capacitance. The resistance of wire is proportional to the cross sectional area of the wire (and therefore the width and thickness of the wire). Wire width must scale with the feature size. Therefore, the thickness of the wire may not be able to scale proportionally to the width due to the increased resistance that would result from the smaller cross sectional area. Electromigration, or ion transport, could also result if wires become too thin. Electromigration is the physical breakdown of the wire itself, and is therefore a major concern for processor reliability. Finally, thinner wires are more difficult to manufacture.

Wire capacitance has two components that compose the overall parasitic: parallel plate capacitance and fringing capacitance. Parallel plate capacitance is the capacitance between the wire and the substrate and is proportional to $\frac{W}{H}$, where W is the wire width and H is the distance between the wire and the substrate. Fringing capacitance is the capacitance between the side-walls of the wires and the substrate and is proportional to $\frac{T}{H}$, where T is the wire thickness and H is the distance between the wire and the substrate. Parasitic capacitance exists between wires and the substrate and between wires themselves. This lat-



(a)



(b)

Figure III.1: Interconnect Parasitics
(a) Physical dimensions of a wire. W is wire width, T is wire thickness or height, and L is wire length.
(b) Layout of wires with capacitative parasitics. Shown are three wires, two in metal layer 1 (M1) and one in metal layer 2 (M2). The straight lines represent parallel plate capacitance between a wire and either another wire or the substrate. The curved lines represent fringing capacitance between a wire and either another wire or the substrate.

ter component, known as coupling capacitance, also has both a parallel plate component and a fringing component, and is proportional to the distance between wires on the same layer or in different layers. As this distance shrinks, the coupling capacitance increases. As W decreases in size, the fringing capacitance component begins to dominate the overall capacitance. The following equation from [90] provides an empirical formula for capacitance:

$$C = \epsilon[(\frac{W}{H}) + 0.77 + 1.06(\frac{W}{H})^{0.25} + 1.06(\frac{T}{H})^{0.5}] \qquad \text{(III.1)}$$

The ϵ in this equation is permittivity of the insulator. Figure III.1 illustrates the different capacitative parasitics.

One result of parasitics between wires is noise. The Miller effect is one example of this. When adjacent wires both switch at the same time in different directions (i.e. one wire switches from V_{DD} to 0 and the other from 0 to V_{DD}), the voltage swing between the two wires is actually $2V_{DD}$. This can impact the delay time of the interconnect. Another example of noise is crosstalk. Crosstalk involves a static wire (not exhibiting switching behavior) that receives an involuntary voltage spike due to neighboring wires that switch in the same direction. This effect is typically addressed by adding shielding wire (GND or V_{DD}) between wires. This can also be done to reduce inter-layer crosstalk by adding a GND or V_{DD} metal plane between each layer. However, there will still be capacitative effects between the metal plane and the wires.

Global wires are more impacted by resistative effects due to their longer length, while local wires are plagued by capacitative effects, especially in dense interconnect. A more complete discussion of these parasitic effects can be found in [69].

There has been a significant amount of analytical [55] and empirical [8, 59] analyses of the interconnect scaling bottleneck in the process technology literature. Recently, these analyses have carried over into the computer architecture literature where their effects on the execution core have been examined [60].

There are three important results of the interconnect scaling bottleneck:

1. memory structures experience the full extent of this trend because they are composed of significant amounts of closely packed interconnect

2. larger memory performance scales worse than small memory performance because larger memory is composed of significantly more interconnect

3. interconnect scaling degrades as process feature size decreases due to increasing parasitic capacitance effects; if current trends continue, wire latency will no longer scale and may increase in future process generations.

To better understand why on-chip memory performance scales poorly with process feature size, we need to examine more closely the structure of on-chip memory. On-chip memory devices are composed of large two-dimensional arrays of memory cells. Connecting these memory cells to other parts of the chip is a tapestry of wire that forms two buses. The *wordline* bus runs the rows of the array, bringing signals to the cells that indicate if the cells are being accessed. The *bitline* bus runs the columns of the array, providing access to memory cell contents. To access the memory, a decoder activates a row of the memory array by asserting a single wordline, this results in the contents of every cell in the row being asserted on the bitline bus. A multiplexor at the end of the bitlines is used to select the accessed data. A more complete analysis of cache structure is presented in Chapter VII.

The latency of a memory device, to a first order, is the latency to exercise the logic in the decoder, assert the wordline wire, read the memory cell logic, assert the bitline wire, and finally exercise the logic in the bitline multiplexor to select the accessed data. As the process feature size is scaled, the latency of the transistors is scaled proportional to their size, thus the latency of the logic scales linearly with feature size reductions.

The latency of the wordlines and bitlines, on the other hand, does not scale as well due to *parasitic capacitance* effects that occur between the closely packed wires that form these buses. As the technology is scaled to smaller feature sizes, the thickness of the wires does not scale. As a result, the parasitic capacitance formed between wires remains fixed in the new process technology (assuming wire length and spacing are scaled similarly). Since wire delay is proportional to its capacitance, signal propagation delay over the scaled wire remains fixed even as its length and width are scaled. This effect is what creates the *interconnect scaling bottleneck*.

Since on-chip memory tends to be very wire congested (wordlines and bitlines are run to each memory cell), the wires in the array are narrowly spaced to minimize the size of the array. As a result, these wires are subject to significant *parasitic capacitance* effects. Agarwal et al. [2] conclude that architectures which require larger components will scale more poorly than those with smaller components. They further conclude that larger caches may need to pay substantial delay penalties.

Recently, some process technologies have begun employing copper interconnect and *low-k* dielectrics as a way to reduce the impact of poor interconnect scaling [48, 49]. These materials lower the resistance and capacitance of wires, respectively, thereby improving signal propagation performance. Copper wires are also more resistant to electromigration. However, these techniques only offer a one time reprieve for the first process generation that employs them. Poor interconnect scaling trends continue. It has been shown that splitting long wires with buffers can reduce their propagation delay [6]. However, this approach cannot be applied to the the densely packed interconnect of memory arrays without significantly increasing their area (due to the buffers required).

In order to avoid the potential hazards of the interconnect scaling bottleneck on the memory structures of the front-end, architects will need to observe the following:

1. prediction structures that have intercycle dependences (*i.e.* where the prediction from cycle i depends on cycle $i-1$ - such as branch predictors) must be kept small in order to reduce the processor cycle time

2. other predictors will need to be pipelined to keep up with decreasing cycle times (such as instruction caches) — the fundamental limitations on how far a structure may be able to be pipelined may require these structures to be kept small as well

3. to offset the loss in performance due to decreased predictor size, architects will need to make use of techniques like multilevel predictor hierarchies, helper engines [82], and aggressive prefetching

III.A.2 Clock Scaling

As processor clock speeds increase, and cycle times decrease, structures in the processor pipeline need to either decrease their access time or increase the degree to which they are pipelined. Unfortunately, in order to decrease the access time of a memory structure, the size or complexity (i.e. number of ports, associativity, etc) of the structure would also likely need to be reduced. Reducing the size or complexity of a memory structure can mean reduced performance. Although a structure can be pipelined to be accessed in multiple cycles, there may be a limit on the amount of pipelining that is possible for a given structure. Agarwal et al. [2] used the *fanout-of-four* (FO4) delay metric to study the limitations of pipelining processor structures in future technology sizes. They found that pipelining was limited by the overhead associated with latches between pipeline stages and the amount of data dependencies between instructions in different stages of the pipeline. According to their study, architects may no longer be able to both improve IPC and increase the clock speed of the processor. These goals may become antagonistic in conventional microarchitectures.

As an example of this, consider the instruction cache. Instruction cache misses stall instruction delivery until instructions are returned from the next level of the instruction memory hierarchy. One solution to this is to increase the size or associativity of the instruction cache. However, as processor cycle times continue to decrease, this option becomes less viable, especially in light of the interconnect scaling bottleneck. A larger instruction cache could be pipelined to fit a particular cycle time, but in a contemporary front-end architecture, this would also imply that the branch prediction architecture would need to be pipelined. Pipelining the branch predictor is difficult, since each prediction relies on the previous prediction, although this design has been proposed in [77]. Moreover, there may be a fundamental limitation on how far the instruction cache can be pipelined.

III.A.3 Energy Dissipation

In the pursuit of high performance processors, microarchitects have attempted numerous strategies to improve throughput or exploit ILP, some of which have also increased the complexity of the processor or hastened the frequency of the clock. Such strategies can result in greater power consumption, which in turn impacts cooling, packaging, and in the case of mobile computers, battery lifetime [69]. Gwennap [34] points out that power consumption may limit not only what can be integrated onto a chip, but also how fast the chip can be clocked. Pollack [67] observes that the power density of current microprocessors is growing at an alarming rate - at a $0.5\mu m$ feature size, the Pentium line of microprocessors has a power density around 10 W/cm^2 (the power density of a hot plate).

While structures that are closer together on chip may have less delay between them (shorter wires), this will also imply that the energy dissipated by these structures will be distributed over a smaller area – implying that more heat must be removed from a smaller area to avoid thermal failure. While standard processors make use of circulated air as a cooling medium, novel heat sink designs coupled with liquid or inert gas cooling mediums are becoming more popular in supercomputing and may be more important in standard processors as clock speeds continue to increase. [69]

In order to address these concerns, architects must consider the impact a potential processor enhancement might have on power consumption, in addition to examining ways of making current processor components more energy efficient. Brooks et al. [11] report that instruction fetch and the branch target buffer are responsible for 22.2% and 4.7% respectively of power consumed by the Intel Pentium Pro. Brooks also reports that caches comprise 16.1% of the power consumed by Alpha 21264. Montanaro et al. [25] found that the instruction cache consumes 27% of power in their StrongARM 110 processor.

III.B Latency Concerns

In addition to examining the hardware scaling implications that can impact the front-end processor, the actual structures of the front-end have fundamental interdependencies which can impact processor performance. First, we will examine the two major structures of the front-end: the instruction cache and the branch predictor. Then, the relationship between these structures will be examined.

III.B.1 Instruction Cache

Instruction cache performance is vital to the front-end of the processor. As mentioned in Chapter III.A, larger instruction caches will scale poorly compared with smaller instruction caches to future technology sizes. The number of ports on the instruction cache directly impacts the area of the cache and can impact the timing and energy dissipation of the cache, even within the same technology size [74]. The number of ports on the instruction cache directly impacts the available fetch bandwidth of the processor.

III.B.2 Branch Prediction

There are several complications that arise in the design of a branch prediction architecture. The misprediction of the address or direction of a branch forces a pipeline flush, resulting in wasted fetch bandwidth between the time the branch was mispredicted and the time the misprediction was detected. Therefore, a predictor must keep mispredictions at a minimum [57, 18].

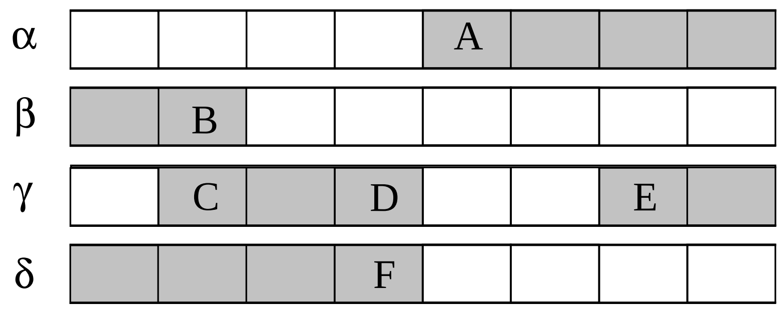
Secondly, in modern front-end designs, predicting the target of a branch requires an access to the branch predictor and branch target buffer (BTB). As a result, the rate at which these devices can be cycled times the average basic block size places an upper limit on instruction delivery rates. Techniques such as predicated execution [62] attempt to increase the size of basic blocks. Another alternative is to increase the number of branches predicted in a single cycle via traces [75], fetch blocks [70], or with multiple predictors [77].

III.B.3 Interaction Between Branch Prediction and Instruction Cache

In a conventional microprocessor architecture, the branch predictor and instruction cache are accessed in parallel – coupled together in the same stage. Both structures stall when the issue buffer between the front-end and the execution core fills. The branch predictor will stall if the instruction cache stalls.

Branch predictors produce basic or fetch blocks, which can span multiple cache blocks. Because instruction caches have a limited number of ports, and because adding ports to the instruction cache is expensive, the branch predictor may need to stall while the instruction cache consumes an entire branch prediction – possibly over several cycles. Moreover, when the instruction cache stalls in a contemporary front-end architecture, the branch predictor must also stall. Both cases result in wasted cycles where the predictor could be generating a fetch stream. Simply increasing the issue buffer size will not alleviate this, as the bottleneck in this case is the number of ports on the instruction cache. Stark et al. [84] propose a mechanism to allow the instruction cache to continue in the face of a miss, but are still limited by the number of ports on the instruction cache.

Figure III.2(a) provides a sample fetch stream that has been organized according to instruction cache blocks. In this figure, cache blocks are represented



(a)

Cycle	Branch Predicted	Cache Block Provided
1	B	α
2	STALL	β
3	D	γ
4	F	γ
5	STALL	δ

(b)

Cycle	Branch Predicted	Cache Block Provided
1	B	α
2	D	β
3	F	γ
4	H	δ

(c)

Figure III.2: Example of predictor stalls in a coupled front-end design.
(a) is a sample fetch stream of four cache blocks. α, β, γ, and δ are cache block labels. A–F are instructions in the cache blocks that bound the basic blocks of the fetch stream. Darkened instructions are part of the fetch stream (A–B, C–D, E–F are the basic blocks).
(b) shows how a coupled front-end design (with a single cache port and the capability to perform a single branch prediction per cycle) would handle the fetch stream.
(c) shows how the performance of the front-end improves when the branch predictor stalls are removed. (H is the branch in basic block G–H, the next basic block in the fetch stream – this is not shown in (a))

by an 8 entry rectangular buffer (each entry represents a single instruction). The sample stream ranges over four cache blocks that are labeled α, β, γ, and δ. Blocks α and β are sequential cache blocks, and blocks γ and δ are sequential cache blocks. The letters A-F represent different instructions in the cache blocks. Letters A, C, and E are the heads of basic blocks. Letters B, D, and F are the tails of basic blocks and are branches. In this example, all branches are taken. A–B, C–D, and E–F form three basic blocks in this sample fetch stream. Blocks A–B and E–F span 2 cache blocks (A–B spans α and β, E–F spans γ and δ).

Figure III.2(b) demonstrates how a contemporary fetch architecture would handle this fetch stream, assuming a single ported instruction cache and a branch predictor that can perform a single prediction per cycle. The first column shows the cycle starting from 1, the second column shows the branch that is predicted in the given cycle, and the third column shows the cache block provided by the instruction cache in that cycle. Using the PC of instruction A, branch B is predicted in cycle 1, and cache block α is provided. Since the basic block predicted spans two cache blocks, the branch predictor stalls in cycle 2 while the instruction cache provides block β. In cycle 3, branch D is predicted, and the instruction cache provides block γ. In cycle 4, branch F is predicted, and the instruction cache must again provide block γ. Finally, in cycle 5, the instruction cache provides block δ – but the branch predictor must stall again. Thus, five cycles are required to provide the four cache blocks from Figure III.2(a).

If the branch predictor did not have to stall, only 4 cycles would be required to provide the prediction. Figure III.2(c) illustrates this. Again, branch B is predicted in cycle 1, and the cache provides block α. But this time, the branch predictor can continue even though the instruction cache has not yet completed consuming the current prediction. In cycle 2, the instruction cache grabs the next block (β) while the branch predictor predicts branch D using the target of the first prediction (instruction C is the target and head of the next basic block). In cycle 3, branch F can be predicted and block γ can be provided. Finally, in cycle 4, block δ can be provided and the branch predictor can provide a prediction for the next basic block in the fetch stream (assuming a basic block G–H not shown in Figure III.2(a)). Here, only four cycles were required to provide the same bandwidth, and one additional prediction is queued up to be used (branch H) – potentially providing another opportunity for performance improvement.

Chapter IV

Prior Work

Much work has been put into the front-end architecture in an effort to improve the rate of instruction delivery to the execution core. Techniques to reduce the impact of I-cache misses include multi-level instruction memory hierarchies [41] and instruction prefetch [87]. Techniques to reduce the impact of branch mispredictions include hybrid [57] and indirect [18] branch predictors, and recovery miss caches to reduce misprediction latencies [10]. A number of compiler-based techniques work to improve instruction delivery performance. They include branch alignment [14], trace scheduling [28], and block-structured ISAs [35].

We will now examine some of the more relevant prior work in detail.

IV.A Future Technology Scaling

Palacharla et al. [60] examined the effects of process technology scaling on the performance of the execution core. They found that the performance of operand bypass logic and datapaths scales poorly due to the large amount of interconnect in their datapaths.

Agarwal et al. [2] also examined the effects of process technology scaling, and found that current techniques of adding more transistors to a chip and superlinear clock scaling will provide diminishing returns to processor performance. Both studies show that larger memory performance scales worse than small memory because it is composed of significantly more interconnect.

IV.B Out of Order Instruction Fetching

Stark et. al. [84] proposed an out-of-order fetch mechanism that allows instruction cache blocks to be fetched in the presence of instruction cache misses. On an instruction cache miss, the branch predictor would continue to produce one prediction per cycle, fetch the instructions, and put them into a result fetch queue out of order. The instructions are decoded in-order however.

The main difference between our approach and the one by Stark et. al. [84] is that our decoupled architecture can expose a larger address stream in front of the fetch unit enabling additional optimizations. A branch predictor can run farther ahead of the instruction cache, since it can produce fetch addresses for several cache blocks per prediction, which is more than the instruction cache can consume each cycle. These addresses can then be used for optimizations like prefetching, building a multi-level branch predictor, or designing more scalable predictors (e.g., value prediction). In their approach [84], the branch predictor is tied to the lockup-free instruction cache, and produces addresses that are directly consumed by the instruction cache. This keeps the instruction cache busy and provides potential fetch bandwidth in the presence of instruction cache misses, but does not expose the large number of fetch addresses ahead of the fetch unit as in the decoupled design.

We call the number of instruction addresses stored in the FTQ at a given time the occupancy of the FTQ. Results in Chapter VIII show that there can be a large FTQ occupancy even when the instruction window doesn't fill up and stall, since the branch predictor can predict more cache blocks than the instruction cache can consume each cycle.

IV.C Helper Engines

Smith [82] predicts a shift in processor design goals – from the goal of maximizing ILP to facilitating inter-instruction communication. Because it may be difficult to communicate across the chip in a single cycle [2], the actual communication of data between dependent instructions may become a critical bottleneck to processor performance. Smith advocates a simple core pipeline, designed with the fastest transistors possible, that is further augmented by *helper engines*. These speculative engines can work to improve processor performance further by specializing on particular aspects of the pipeline. He cites our branch prediction architecture [70] as an example of such a speculative engine. The instruction coprocessor [22] is another such engine.

IV.D Branch Prediction

Branch Target Buffers (BTB) have been proposed and evaluated to provide branch and fetch prediction for wide issue architectures. A BTB entry holds the taken target address for a branch along with other information, such as the type of the branch, conditional branch prediction information, and possibly the fall-through address of the branch.

Perleberg and Smith [64] conducted a detailed study into BTB design for single issue processors. They even looked at using a multi-level BTB design, where each level contains different amounts of prediction information. Because of the cycle time, area costs, and branch miss penalties they were considering at the time of their study, they found that the "additional complexity of the multi-level BTB is not cost effective" [64]. Technology has changed since then, and as we show in this thesis, a multi-level branch prediction design is advantageous.

Yeh and Patt proposed using a *Basic Block Target Buffer* (BBTB) [94, 95]. The BBTB is indexed by the starting address of the basic block. Each entry contains a tag, type information, the taken target address of the basic block, and the fall-through address of the basic block. If the branch ending the basic block is predicted as taken, the taken address is used for the next cycle's fetch. If the branch is predicted as not-taken, the fall-through address is used for the next cycle's fetch. If there is a BBTB miss, then the current fetch address plus a fixed offset is fetched in the next cycle. In their design, the BBTB is coupled with the instruction cache, so there is no fetch target queue. If the current fetch basic block spans several cache blocks, the BBTB will not be used and will sit idle until the current basic block has finished being fetched.

Several architectures have been examined for efficient instruction throughput including the two-block ahead predictor [77], the collapsing buffer [24], and the trace cache [75]. Seznec et. al. [77] proposed a high-bandwidth design based on two-block ahead prediction. Rather than predicting the target of a branch, they predict the target of the basic block the branch will enter, which allows the critical *next PC* computation to be pipelined. Conte et. al. [24] proposed the collapsing buffer as a mechanism to fetch two basic blocks simultaneously. The design features a multi-ported instruction cache and instruction alignment network capable of replicating and aligning instructions for the processor core. Rotenberg et. al. [75] proposed the use of a trace cache to improve instruction fetch throughput. The trace cache holds traces of possibly non-contiguous basic blocks within a single trace cache line. A start trace address plus multiple branch predictions are used to access the trace cache. If the trace cache holds the trace of instructions, all instructions are delivered aligned to the processor core in a single access. Patel et. al. [63] extended the organization of the trace cache to

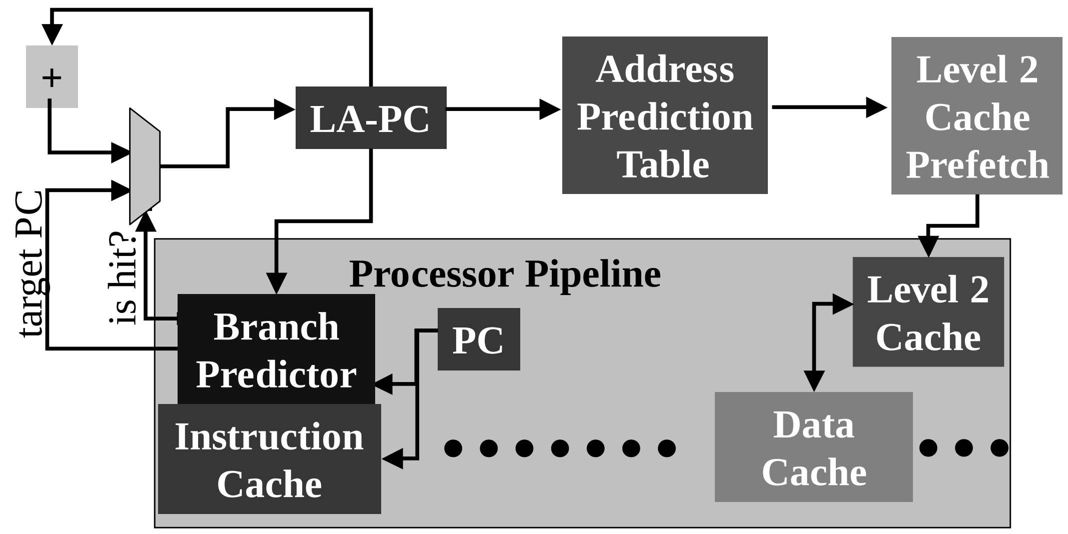


Figure IV.1: Look-Ahead PC of Chen and Baer [21]

include associativity, partial matching of trace cache lines, and path associativity.

IV.E Fetch Guided Cache Prefetch

A form of fetch guided prefetching was first proposed by Chen and Baer [21] for data prefetching. In their prefetching architecture they created a second PC called the *Look-Ahead PC*, which runs ahead of the normal instruction fetch engine. This LA-PC was guided by a branch prediction architecture, and used to index into a reference prediction table to predict data addresses in order to perform data cache prefetching. Since the LA-PC provided the address stream farther in advance of the normal fetch engine, they were able to initiate data cache prefetches farther in advance than if they had used the normal PC to do the address prediction. This allowed them to mask more of the data cache miss penalty. Chen and Baer only looked at using the LA-PC for data prefetching [21]. Figure IV.1 provides a summary of the architecture.

Chen, Lee and Mudge [19] examined applying the approach of Chen and Baer to instruction cache prefetching. They examined adding a separate branch predictor to the normal processor; so the processor would have 2 branch predictors, one to guide prefetching and one to guide the fetch engine. The separate branch predictor uses a LA-PC to try and speed ahead of the processor, producing potential fetch addresses on the predicted path. This separate branch predictor was designed to minimize any extra cost to the architecture. It only included (1) a global history register, (2) a return address stack, and (3) an adder to calculate the target address.

In their design, each cycle the cache block pointed to by the LA-PC is fetched from the instruction cache in parallel with the normal cache fetch. If it is not a miss, the cache block is decoded to find the branch instructions and the target addresses are calculated. When a branch instruction is found in the cache block it is predicted using the separate branch prediction structures, the LA-PC is updated, and the process is repeated. This whole process is supposed to speed ahead of the normal instruction fetch, but it is limited as to how far it can speed ahead because (1) the prefetch engine uses the instruction cache to find the branches to predict and to calculate their target addresses, and (2) their prefetch engine has to stop following the predicted stream whenever the LA-PC gets a cache miss. When the LA-PC gets a cache miss, their prefetcher continues prefetching sequentially after the cache line that missed. The prefetching architecture that we describe in Chapter X follows the fetch stream prefetching *past* cache blocks that miss in the cache and does not need to access the instruction cache to provide predicted branch target and prefetch addresses since the branch predictor is completely decoupled from the instruction cache fetch via the fetch target queue.

IV.F Decoupled PC-based Value Prediction

Value prediction predicts the actual data value that is to be brought in from memory, allowing instructions dependent on the load to speculatively execute with the predicted value. If the prediction is correct, this breaks true data dependencies since the value is produced without having to wait on the load instruction.

When a load is value predicted, the value is used to update the current value and status of the load's destination register. This value will then be seen and used by subsequent instructions. The load still takes its normal path of execution for a non-speculative load — it calculates its effective address, accesses the store buffer and memory. When the load's value becomes available it is checked against the predicted value of the load instruction for miss-speculation. This is called a *check-load*, since the load is used to check the predicted value.

Several architectures have been proposed for value prediction including last value prediction [51, 52], stride prediction [29, 31], context predictors [76], and hybrid approaches [89]. We have examined the use of confidence to improve value predictor accuracy [17].

Lee et al. [50] proposed a decoupled value prediction architecture that worked in conjunction with a trace cache. Value prediction occurred in the write-back stage, and predictions were stored in a Value Prediction Buffer accessed in parallel with the trace cache. This allows more flexibility in the design of the value predictor, as predictor access time is no longer on the critical path.

IV.G Low Power Cache Research

Recently, there has been a lot of research into creating power efficient caches. Kin et al. [45] looked at using a small L0 cache, called the filter cache, to capture the most frequently accessed cache blocks and to decrease energy dissipation. Bellas et al. [7] examine using compiler techniques in conjunction with the filter cache to statically determine what to put in the instruction cache and what to put in the filter cache. The compiler can then lay out frequently accessed blocks contiguously in memory to try and increase program locality and the hit rate of the filter cache. These techniques could be used in concert with the speculative fetch architecture described in Chapter X.

Bahar et al. [5] examined using a flexible small cache (buffer) in conjunction with the instruction cache. They examined using the buffer as a non-swapping victim cache to conserve power. They also look at using the buffer as auxiliary storage for the instruction cache – filtering references that were either low in confidence due to a potentially mispredicted branch or that were not determined as critical (based on the number of entries in the dispatch queue or RUU).

Su and Despain [85] suggest using block buffering, cache sub-banking, and gray code addressing to reduce cache power consumption. Cache sub-banking saves power by hashing the banks. Ko et al. [46] propose another form of sub-banking called multiple-divided modules (MDM), which divides each sub-array (module) even further. Ghose and Kamble [30] proposed using multiple block buffering, with sub-banking, to reduce the energy dissipated by caches. This is similar to a filter cache, but the multiple block buffers are probed in parallel with the cache. This technique could benefit from the approach described in Chapter X by only selectively accessing the buffer (if any) that hits, or by avoiding the buffer access entirely. All of these techniques still keep all ways of an associative set in the same sub-bank (or MDM) however, so they still consume power driving all ways of an associative set. In using a serial cache, only ways that have been verified as cache hits are driven.

Albonesi [3] examined partitioning all ways of the data component of the cache into separate sub-arrays. He examined selectively turning off different cache ways to conserve energy. Agarwal and Pudar [1] proposed the column associative cache as a way of reducing conflict misses in a direct mapped cache. The cache is split into two banks, each indexed by a different hashing function. The first bank is examined first, and on a miss, the second is probed. Calder et al. [16] followed this work with the predictive selective associative cache as a means of providing the same performance (and energy efficiency) of a direct mapped cache, but with the hit rate of a set associative cache. This approach speculates on the location of the requested data in the cache, instead of probing the tag array initially. Inoue et al. [38] also examined using an MRU algorithm to predict what way of an associative cache to access. All of these caches are speculative in nature, concentrate on data cache performance, and can result in wasted energy and missed access opportunities on cache misses and mispredictions.

Another way of reducing the energy dissipated by a cache is to make use of a serial cache. We demonstrate the ability of the serial cache to reduce energy dissipation in Chapter X. Many researchers have examined serial cache designs. In this design the tags are first checked, and afterwards the data is looked up. This has been used for L2 caches, and recently for data caches on graphics cards [54, 37]. For example, the Alpha 21264 [44] splits the tag and data component of its direct-mapped second level cache, effectively creating a serial L2 cache design.

IV.H Next Line and Set Prediction

Calder and Grunwald [15] proposed an alternative form of branch prediction called *next cache line and set prediction (NLS)* that predicts an index into the instruction cache rather than a branch target address. This predictor provides a pointer into the instruction cache, indicating the target instruction of a branch. This form of predictor is used in the Alpha 21264 [44] and provides extremely accurate predictions. Moreover, it supplies a prediction of what associative way the predicted cache block is located in. The authors refer to way prediction as set prediction in [15], but essentially the predictor determines what associative way contains the desired cache block. This way prediction, combined with an energy efficient cache design that allows a single cache way to be selectively enabled can provide significant energy savings. Calder et al. [16] demonstrate this, and we examine this further in Chapter XI. Unfortunately, the NLS predictor can only supply a single instruction cache block per prediction. This means that the NLS predictor cannot run ahead of the instruction cache and provide the same latency tolerance and look-ahead mechanism that our decoupled front-end can provide.

Wallace and Bagherzadeh [88] extended this work to provide multiple cache blocks per cycle using multiple NLS predictors. However, they do not examine providing multiple way predictions. They use the current cache block to index into two predictors which in turn provide the next two cache block predictions. There is a potential problem with their design however, as only a single group of target blocks are stored for the secondary prediction. If the first prediction is based on a conditional branch and can potentially lead to two cache blocks (fallthrough and target blocks), then only one of these cache blocks will have their prediction stored in the secondary predictor. In other words, if the first prediction is not biased towards taken or not taken, then the prediction from the secondary predictor may correspond to the wrong cache block.

Chapter V

Evaluation Strategy

Through the years, architects have made use of a variety of techniques to quantify the performance of different microprocessor designs.

V.A Prediction Rate

Initially, architects used prediction or misprediction rates to evaluate architectural performance. Various structures, such as branch predictors or caches, can be analyzed by reporting their prediction rate – the frequency with which they return a correct prediction. However, this metric fails to capture the notion of prediction importance. Some predictions are more important than others, and continuously mispredicting a branch on the critical path could have considerably more impact than mispredicting a branch off the critical path. However, both of these mispredictions would be equally weighed in a scheme that just measured prediction rates. Moreover, processor pipelines are complex and require full simulations to make intelligent conclusions about the performance of a particular architecture. For example, in an architecture that is severely fetch constrained, improvements to the data cache will likely have minimal impact on the actual processor performance. But, improvements to the branch predictor may greatly improve performance. Simply looking at the misprediction rates of these structures will not necessarily demonstrate this.

V.B IPC

Instruction Per Cycle (IPC) is a metric that is used to capture the impact of architectural decisions on the processor pipeline as a whole. Rather than simply considering the prediction rate of a given pipeline structure, IPC captures how fast instructions are committed from the pipeline, taking into account the importance of predictions on the critical path.

IPC can be determined using two different simulation techniques: trace-based simulations and execution-based simulation. A trace is a recorded series of instructions that was previously run on a real processor. This trace can then be fed into a simulator to determine how well the architecture being simulated performs. Unfortunately, because the actual program is not being executed, this type of simulation does not model the effects of going off the nonspeculative path. Branch mispredictions impact performance by stalling the processor pipeline (which can be simulated even when using a trace), but the effect of going down a mispredicted path on the various structures of the pipeline (such as the instruction cache and data cache) is not modeled. Mispredicted path instructions can cause useful entries in the instruction cache to be replaced with entries that may never be used. Conversely, mispredicted path instructions can provide useful "prefetches" to the various structures – as would be the case for a mispredicted short forward branch. In any case, these secondary effects cannot be modeled using instruction traces.

The alternative is to use an execution-based simulator. Here, the actual program is executed on top of a simulator architecture. The simulator tracks microarchitecture state for each cycle of execution – and for each stage in the processor pipeline. The state of the simulator represents the actual state of the microarchitecture being simulated. So, if a microarchitecture would mispredict a particular branch in a given program, the simulator will also mispredict that branch and will execute instructions down the mispeculated path. The effects of mispredicted paths on other speculative structures (caches and predictors) can be fully explored in this type of simulation. If a branch is never taken during the formation of a trace, that branch target address will not be known – but with an execution-based simulator, since each individual branch is executed, each branch target can be computed and potentially executed (even on a mispeculated path).

Another possible hazard is the simulation of initialization effects. It is often not practical to simulate an entire program to completion on a execution-based architectural simulator. Often, only a selected number of instructions are simulated. Care must be given to selecting both the number of instructions and the point in the program at which measurement will begin. First, enough instructions should be executed to give a reliable estimate of the performance of the program. This should include all major loops and procedures of the program. Secondly, the measurement should start after the initialization portion of the program (the part of the program where data is loaded into memory, arrays are initialized, etc) as this part of the program does not accurately reflect the majority of the execution time of the program. Skipping the initialization section of a program is known as fast forwarding. A more complete discussion of this can be found in [78].

With decreasing cycle times and increasing pipeline depth, this metric still does not capture the whole picture. A particular instruction cache might have a greater hit rate and higher IPC than another design, but it might take longer to access, and therefore could impact the cycle time of the processor. Consider a processor with a cycle time of 1.0 nanosecond that can achieve an IPC of 2.5. This processor would take 0.4 seconds to commit 1 million instructions. A processor with a cycle time of 0.6 nanoseconds that could only achieve an IPC of 1.8 would be able to commit the same number of instructions in only 0.33 seconds.

V.C IPC with Timing

The cycle time of the processor is difficult to calculate precisely without performing some form of layout. But, it can be estimated by examining the access time to the structures of the processor pipeline. This can also provide insight into the amount of pipelining necessary for the various structures of the processor pipeline. For example, the cycle time of the processor can be set to that of the branch predictor, and the instruction cache can be pipelined to fit within the cycle time. This assumes that all other structures of the architecture can be appropriately pipelined to fit this cycle time. Now, a branch prediction design that improves IPC, but increases the cycle time can be compared to a smaller branch predictor that has a lower IPC but a faster clock. However, a single unified metric would be better in making comparisons of architectures with different cycle times.

V.D BIPS

To compare architectures with different cycle times, we propose a new metric, *Billions of Instructions Per Second (BIPS).* BIPS is a combination of IPC and the cycle time of the processor in nanoseconds:

$$BIPS = \frac{IPC}{cycle\ time\ in\ nanoseconds}$$

This metric captures both performance and timing information. Architectures with different cycle times can be compared using this metric.

V.E Energy Delay Product

Many architectures today are concerned with conserving energy, whether it be for mobile computing or to simplify packaging of the processor. Energy dissipation can be measured in Joules, but can also be incorporated into the BIPS measurement. The energy delay product [32] is the product of the total time to execute a particular program on the processor and the total energy dissipated by the processor. In this thesis, we are primarily concerned with the energy dissipation of the front-end of the processor, and therefore will only consider the total energy dissipated by the front-end processor in the calculation. Therefore, the energy delay product will be calculated as follows:

$$EDP = \frac{number\ of\ instructions\ committed\ x\ total\ energy\ dissipated\ in\ Joules}{BIPS}$$

The number of instructions committed should not change when comparing two architectures. Even architectures that dynamically reduce the number of instructions executed can still count the same number of instructions committed by counting committed instructions based on the original instruction stream. If an architectural improvement results in a smaller energy delay product for a particular architecture, this means that there has either been an improvement in BIPS, a reduction in the amount of energy dissipated, or both.

V.F Conclusion

The rest of this thesis will make use of either BIPS or IPC with cycle time considerations. IPC will be used whenever comparisons are made between architectures that share a common cycle time, but the structures in the architecture will be appropriately pipelined to match the cycle time. BIPS will be used to compare architectures which may have different cycle times. To measure energy dissipation, Joules or the energy delay product will be used. With any composite metric, it is useful to look at the other metrics which compose it to better understand the result. For example, in the case of the energy delay product, it is useful to examine IPC, cycle time, and total energy dissipation – the three variable components of the energy delay product. The overall metric simply provides a notion of the overall impact of the architectural enhancement being investigated.

In this thesis, the processor cycle time will be based upon the structures of the front-end, namely the instruction cache and branch predictor. This assumes that all other structures in the pipeline can be pipelined to accommodate the given cycle time.

Chapter VI

Simulation Methodology

This Chapter examines the simulator, benchmarks, and timing model used to evaluate the various architectures in this thesis.

VI.A Architectural Simulator

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [12], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

The baseline simulation configuration models a future generation out-of-order microprocessor architecture. Parameters have been selected to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle (from up to two cache blocks) and issue up to 16 instructions per cycle. The architecture has a 128 entry re-order buffer with a 32 entry load/store buffer. In [71], we examined the performance of store sets [23] to perform dependence prediction. This technique provided nearly perfect dependence prediction for the architectures examined. Therefore, for this thesis, we assume perfect dependence prediction for all runs - a very close approximation of the performance benefit of using store sets.

There is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4 load/store units, 2 FP adders, 2 integer MULT/DIV, and 2 FP MULT/DIV. The latencies for these units are: ALU = 1 cycle, MULT = 3 cycles, Integer DIV = 12 cycles, FP Adder = 2 cycles, FP Mult 4 = cycles, and FP DIV = 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

Both instruction and data caches have block sizes of 32 bytes. The data cache is a 32K 4-way set associative cache with 2 read/write ports. The instruction caches used vary according to Chapters in this thesis and will be described within each Chapter. There is a unified second-level 1 MB 4-way set-associative cache with 64 byte blocks, and a 12 cycle cache hit latency. If there is a second-level cache miss it takes a total of 120 cycles to make the round trip access to main memory. The L2 cache has only 1 port. The L2 bus is shared between instruction cache block requests and data cache block requests. We modified SimpleScalar to accurately model L1 and L2 bus utilization. A pipelined memory/bus is modeled, in which a new request can occur every 4 cycles – so each bus can transfer 8 bytes/cycle.

There is a 32 entry 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

We used the McFarling gshare predictor [57] for conditional branch prediction. The predictor has a 2-bit meta-chooser and a 2-bit bimodal predictor, both stored in the branch predictor entry with their corresponding branch. In

Table VI.1: Program statistics for the baseline architecture.

Program	Input	fast fwd (M)	% br exe	% il1 miss	% dl1 miss
compress	ref	0	19.3	0.0	2.6
crafty	ref	2000	11.3	10.6	0.6
deltablue	ref	0	8.2	0.4	16.4
eon	rushmeier	2000	10.1	8.0	0.1
gcc	1cp-decl	400	17.4	7.1	1.9
go	5stone21	1000	13.4	2.7	1.0
groff	someman	0	17.3	5.3	0.5
ijpeg	specmun	2000	4.7	0.0	0.8
li	ref	2000	18.0	0.0	1.4
m88ksim	ref	1000	19.7	1.7	0.1
perl	scrabbl	2000	17.1	3.3	0.4
vortex	ref	1000	14.7	12.7	1.0

The first column gives the benchmark name, the second column provides the data set used in the simulation runs, and the third column shows the number of instructions fast forwarded before simulation (measured in millions and taken from [78]). The final columns provide results for an architecture with a 16K 2-way set associative instruction cache and a 32K 4-way set associative data cache: the percent of committed branches and the percent instruction cache and data cache miss rates.

addition, a tagless gshare predictor is also available, accessed in parallel with the branch predictor. The meta-chooser is incremented/decremented if the bimodal/gshare predictors are correct. The most significant bit of the meta-chooser selects between the bimodal and gshare predictions.

VI.B Benchmarks

We examined a number of SPEC95 and SPEC2000 C benchmarks, as well as two additional benchmarks. `Groff` is a text formatting program and `deltablue` is a constraint solving system. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table VI.1 shows the data set used in gathering results for each program, the number of instructions executed (fast forwarded) before actual simulation (in millions), the percent of committed branches in each program, and the instruction and data cache miss rates. In this case, a 16K 2-way set associative instruction cache is assumed. Each program was simulated for up to 200 million instructions.

For each Chapter, different benchmarks may be used in the evaluation of different processor architectures. For example, results for `ijpeg` will not be shown in Chapter X since `ijpeg` has very few instruction cache misses. The results for this benchmark show no change from a base architecture to any architecture with a form of instruction cache prefetch.

VI.C Timing Model

In order to get a complete picture of the relative performance of various architectural designs, it is useful to investigate the cycle time of the designs. IPC results obtained through SimpleScalar can be combined with timing analysis to provide results in *Billion Instructions Per Second (BIPS)*(as shown in Chapter V).

The timing data we need to generate results in IPS is gathered using the *CACTI* cache compiler version 2.0 [74]. *CACTI* contains a detailed model of the wire and transistor structure of on-chip memories. *CACTI* uses data from $0.80\mu m$ process technology and can then scale timing data by a constant factor to generate timings for other process technology sizes. This thesis reports timings for the $0.10\mu m$ process technology size. However, this scaling assumes ideal interconnect scaling, unlike the model used in [70]. This provides a lower bound on the performance improvement that the decoupled front-end and associated optimizations might make possible. Chapter VII details the modifications made to the original CACTI model [92] to create version 2.0.

Table VI.2 contains the timing parameters for the multilevel branch

Table VI.2: Timing data from *CACTI* version 2.0

Number of Predictor Entries	Access Time (ns)
64	0.58
128	0.59
256	0.62
512	0.65
1024	0.70
2048	0.81
4096	0.91
8192	1.06

(a)

Instruction Cache Size	Associativity	Access Time (ns)
16K	2	0.67
16K	4	0.68
32K	2	0.76
32K	4	0.77

(b)

Table (a) shows timing data for various first level FTB configurations. For each FTB specification (shown as the number of entries in a 4-way associative FTB), the cycle time in nanoseconds computed with *CACTI* is shown. Table (b) shows the cycle times for a variety of cache configurations. The first column specifies the size of the cache. The second column specifies the associativity.

predictor and cache configurations examined for the $0.10\mu m$ technology size. Table VI.2(a) lists the branch predictor sizes (in number of entries) and the *CACTI* timing data in nanoseconds for each configuration, showing the access time in nanoseconds. All branch predictor organizations are 4-way set-associative. Table VI.2(b) lists the timing data for the instruction and data cache configurations examined, showing the access time in nanoseconds.

Chapter VII

CACTI

The original CACTI [92] model calculates access and cycle times of hardware caches. It uses an analytical model to estimate delay down both tag and data paths to determine the best configuration for a given cache size, block size, and associativity (at $0.80\mu m$ technology size). Figure VII.1 demonstrates the architecture of the cache in the analytical model. In addition to providing timing data for each portion of the data and tag paths, CACTI also returns the number of data and tag arrays (in terms of the number of wordline and bitline divisions), and the number of sets mapped to a single wordline, for both tag and data arrays. The original CACTI model does not model cache area, but does estimate wire resistance and capacitance based on cache configuration.

Figure VII.2 illustrates the use of the Nspd, Ndwl, and Ndbl parameters from CACTI. These parameters are optimally computed for the particular cache configuration specified by the user. These parameters are for the data array, and the corresponding `Nspd`, `Ndwl`, and `Ndbl` parameters are for the tag array.

CACTI 2.0 introduces several modifications to the CACTI model. First, the transistor widths used in the original CACTI model are tuned to improve the access time and scalability of the model. Next, the potential bottleneck of the tag path is addressed through a number of techniques. Several new features are in-

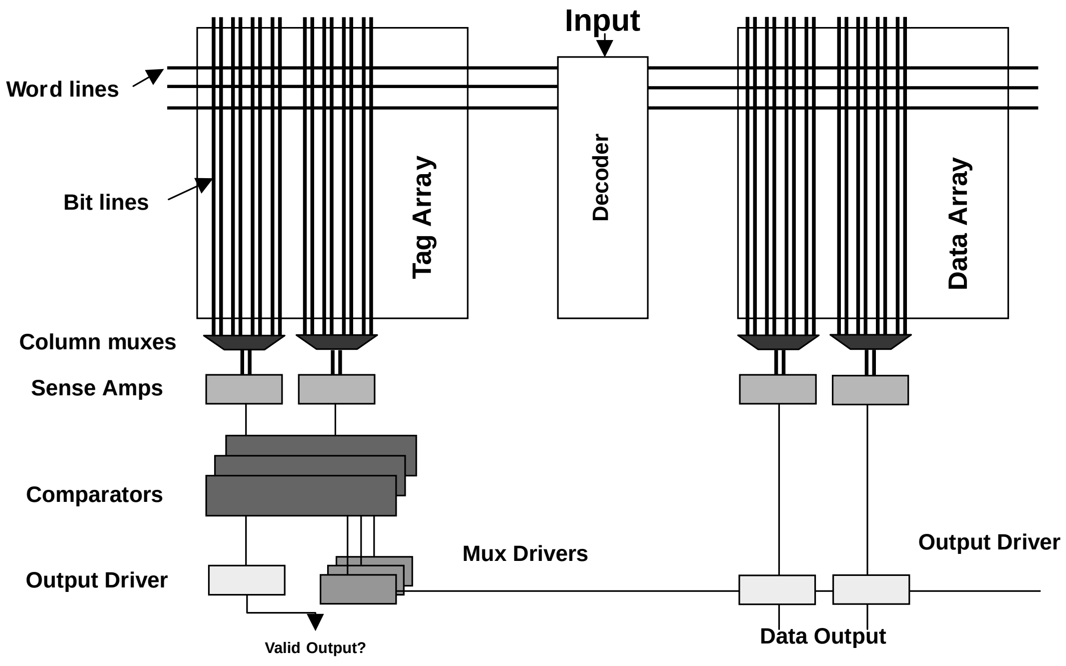


Figure VII.1: Cache model used in CACTI [92].

troduced into the CACTI model: fully associative cache modeling, multiple cache port modeling, and cache power modeling. The timing optimization techniques, fully associative modeling, and multiple cache port modeling are described in Chapter VII.A. The power model is described in section VII.B. The CACTI 2.0 model is verified with hspice. For details of this verification, and for sample input and output, refer to [74]. Finally, we detail the modifications made to the CACTI 2.0 model for this thesis in VII.D.

VII.A New Timing Features

A number of enhancements were made to the CACTI timing model. The access times for set associative caches were optimized by scaling transistor widths and improving the performance of the tag comparison hardware. In addition, support was added for fully associative caches and for caches with multiple access

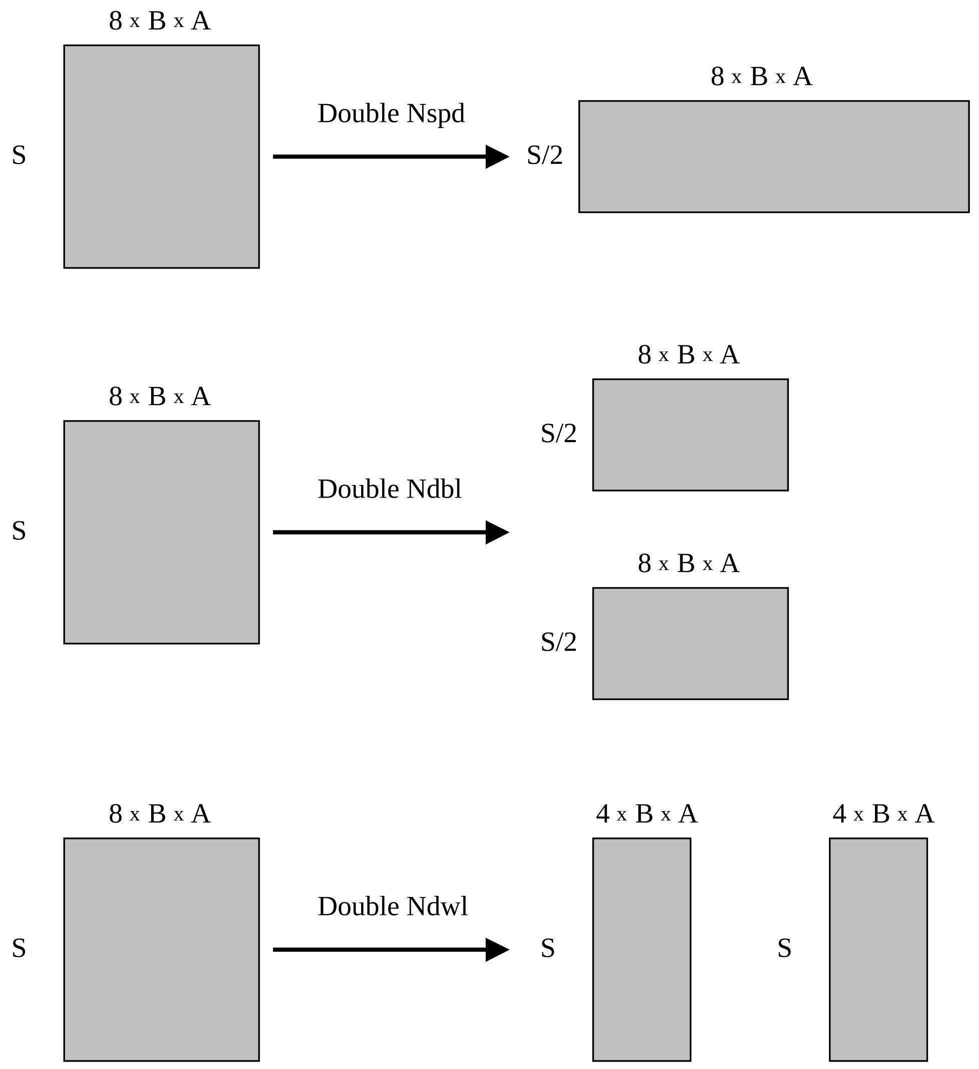


Figure VII.2: Cache division terminology used in CACTI [92].
In this Figure, A is the cache associativity, B is the cache block size, and S is the number of cache sets. The grey box is a data array of a particular cache with the given dimensions in terms of B, A, and S. The left side of the Figure shows the original cache configuration, and the right side of the Figure shows the result of doubling a particular parameter. The first part of the Figure demonstrates the `Nspd` parameter as it extends the width of the given data array by two and correspondingly reduces the length. The second part of the Figure demonstrates the `Ndbl` parameter as it splits the bitlines of the given data array and creates two data arrays half the length of the original. The third part of the Figure demonstrates the `Ndwl` parameter as it splits the wordlines of the given data array and creates two data arrays half the width of the original.

ports. Finally, the handling of process technology sizes and cache cycle time generation was changed.

VII.A.1 Transistor Tuning

Throughout the extension of the CACTI model, it was necessary to scale the width of some transistors on the critical path. Care was taken to avoid making these widths too large and wasting chip area or increasing capacitance. For the most part, the changes to transistor widths were on the tag path, especially in the multiplexor drivers. Avant! AvanWaves (version 1999.2) was used along with a spice model of the cache to determine which sections of the circuit required transistor tuning. By plotting the rise and fall of the transistor voltages, it was possible to determine potential bottlenecks in the circuit. Slowly sloping waveforms indicated a delay which might be alleviated. Changes were made to both spice and CACTI models to determine the overall effect.

VII.A.2 Improving the Tag Path

In set associative caches, the cache tags need to be checked to determine which set of output drivers to select. According to the cache model in Figure VII.1, it can be seen that the access time of the cache is equal to

$$Max(delay_{datapath}, delay_{tagpath}) + delay_{outputdriver}$$

where the data path delay does not include the output driver. In many instances, the tag path takes longer than the data path. For example, in the original cacti model, the tag path of a 16K 2-way associative cache takes 7.8 ns, while the data path takes 4.9 ns (both excluding the data output driver).

We explored three different techniques to lessen the delay of the tag path. First, we provided the option to move the output drivers on the data path

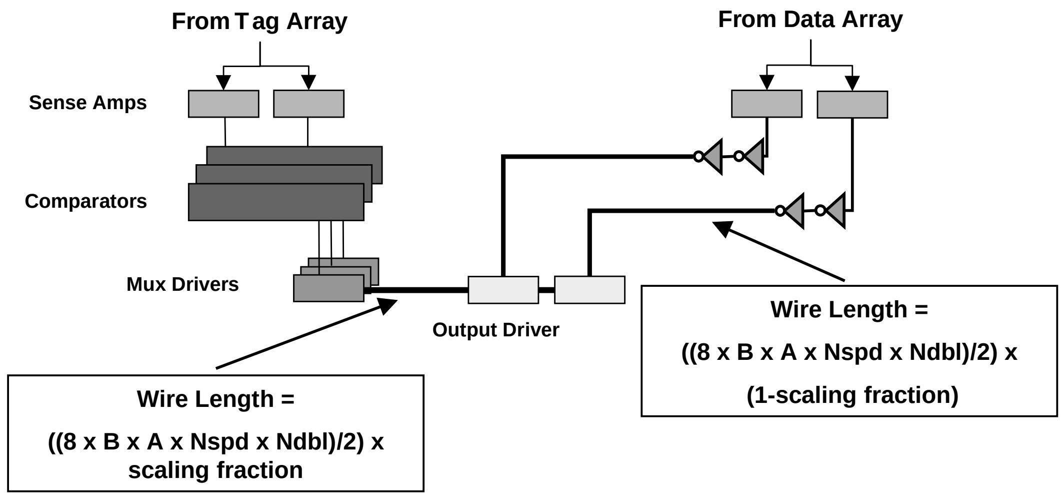


Figure VII.3: Balancing the tag and data paths.
CACTI will determine the scaling factor for the wire length between the mux drivers and output drivers that will result in the lowest overall cache delay.

closer to the multiplexor drivers on the tag path. This decreases the delay of the tag path by reducing the load on the multiplexor drivers, but does increase the delay on the data path. This technique effectively attempts to balance the data and tag paths. Second, we looked at splitting the comparator on the tag path into two structures to reduce its latency. Third, we increased the amount of column multiplexing done on the data path prior to the sense amp stage, while decreasing the amount of multiplexing done on the data path after the compare stage. This has the added benefit of reducing the number of sense amps needed on the data path.

Balancing the Tag and Data Path

The output drivers on the data path were moved closer to the multiplexor drivers on the tag path, trading cache area, power, and data path delay, for decreased tag path delay. In order to drive the increased distance to the output drivers, two inverters were inserted on each sense amp output on the data path.

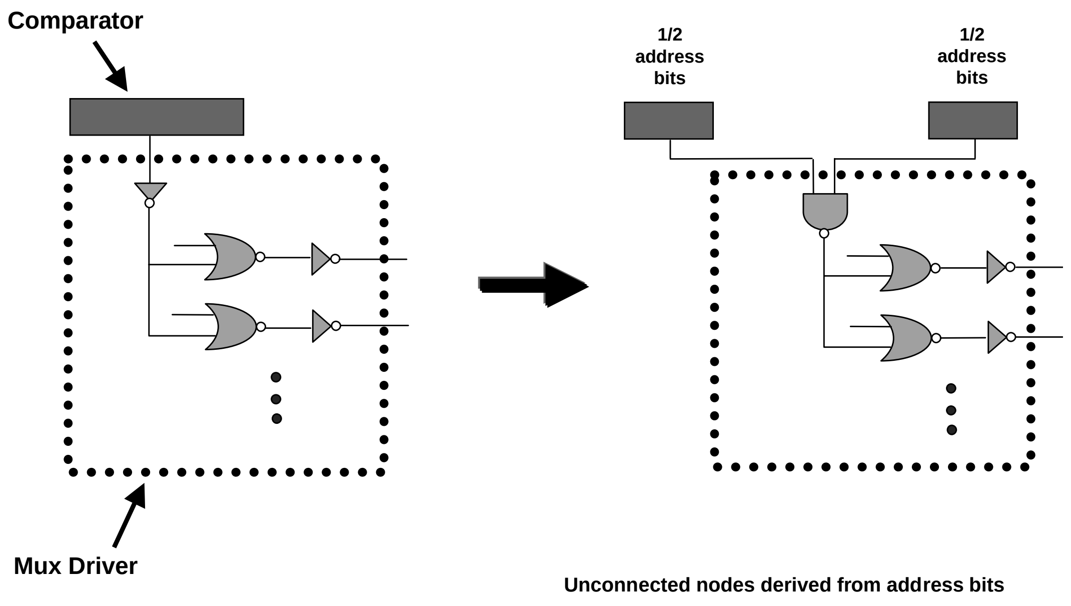


Figure VII.4: Illustration of split comparator.
Each split comparator handles half of the address bits - each performing half of the comparisons of the original single comparator. A NAND gate replaces the inverter in the mux driver and is used to join the signals from the two halves.

This can be seen in Figure VII.3. The CACTI model attempts a range of values for the scaling factor seen in the figure, and will choose the relative position of the output drivers that results in the smallest overall delay. If the benefit of this optimization does not outweigh the cost, the tag path will be left as before, without the additional inverters.

Split Comparator

The second technique involves splitting the comparator on the tag path into two smaller comparators. Each comparator handles one half of the address bits to be compared. This reduces the capacitative load on the comparison line. The two comparators can then be recombined using a NAND gate in the subsequent multiplexor driver stage. The NAND gate will replace the existing inverter used to drive the multiplexor driver. This can be seen in Figure VII.4. The comparator is only split once, as merging more than two signals would likely prove more costly than the savings obtained by further reducing the capacitative load on the comparison line.

Multiplexing Shift

This final change again involves shifting more of the delay from the tag side to the data side. Originally, the multiplexors following the compare stage on the tag path would select from both the different associative entries in the data array and the possible output bits in a single cache line. For example, in a 2-way associative cache with 32 byte lines and 64 output bits, there would need to be 8-way multiplexing at this stage. The multiplexors would have the choice of 512 bits to potentially drive, but only 64 bits are actually output. The 512 bits come from two 32 byte lines, as the cache is 2-way set-associative and two cache lines would share a common wordline. The output bit selection does not depend on the tag path, and therefore can be handled by the bitline column multiplexors that lead to the sense amps (as shown in Figure VII.1). Since the column multiplexors are already responsible for converging bitlines from various subarrays, we limit the degree of multiplexing to 16 (i.e. 16 bitlines to a single sense amp) for the column multiplexor. We introduce this limitation to avoid allowing too many bitlines to share a single sense amp.

VII.A.3 Fully Associative Cache

The new version of CACTI includes support for fully associative caches. In the fully associative cache model, the customary tag path is replaced by a fully associative cache decoder. Rather than tracking a separate tag and data path, the fully associative cache has a single path. The decoder will drive the wordlines of the data array as in the original cache model, but there will only be a single cache

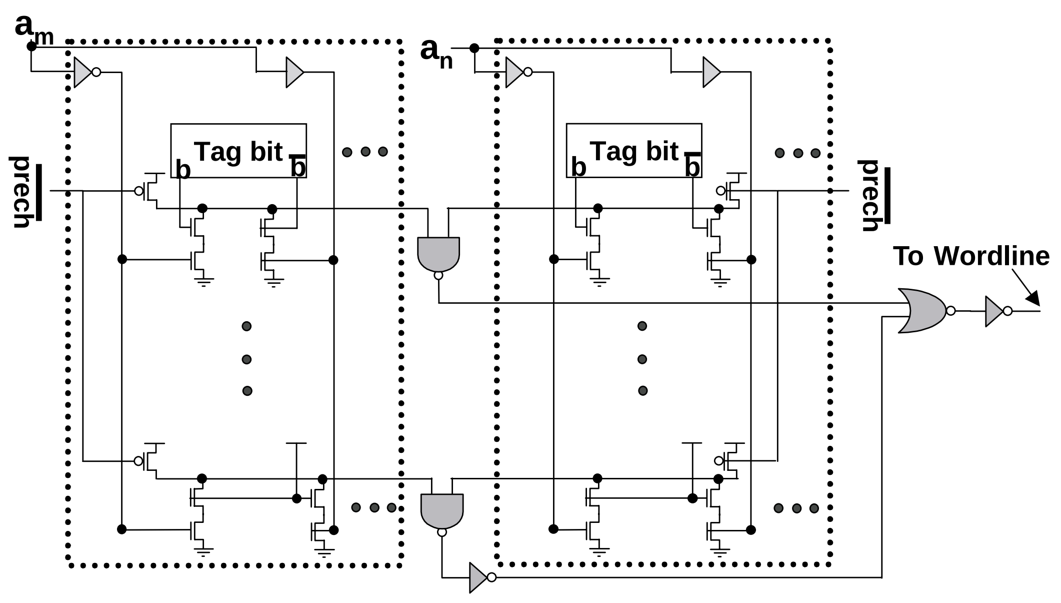


Figure VII.5: Fully associative cache model.
Each dotted square represents a portion of one tag cell. Each tag cell handles half of the address comparisons for a particular tag entry.

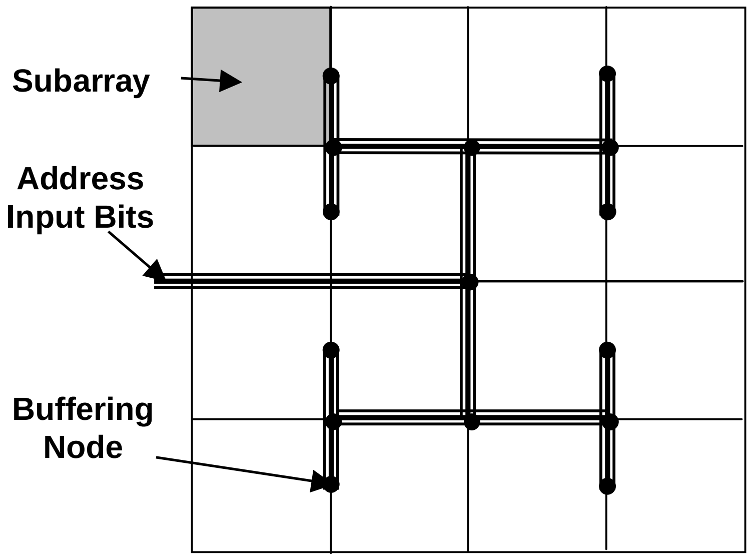


Figure VII.6: Layout of a fully associative cache with 16 subarrays.
The address bits are brought in using an h-tree structure.

entry associated with each wordline. In the decoder, all tag entries are checked for a match. Should a match exist, a single data array wordline will be enabled. Once the data array wordline is enabled, the data path of the fully associative cache proceeds in the same manner as the direct mapped cache. All decode and selection occur prior to the data array access (i.e. there is no multiplexor as in the set associative model). Moreover, as each wordline is associated with a single cache entry, there is no need to try different values of `Nspd`, `Ntspd`, `Ndwl`, or `Ntwl`. `Ndbl` may be varied, as dividing the bit lines will help reduce the delay associated with searching the entire tag array. Since selection occurs before the wordline is even driven, there are less bitlines brought low in a fully associative design - which helps in reducing the power consumed by the cache. However, since the tag comparison cannot proceed in parallel with the data array access, the delay of a fully associative cache is typically larger.

The first stage of the fully associative cache involves checking each bit of the probe address with each corresponding bit of the cache addresses. The probe address is delayed by using a number of inverters. This simulates the probe address drivers and timing chain of the cache. The tag comparison stage of the fully associative cache is split in a manner similar to the comparator in the set associative cache – each comparator only looks at half of the address bits. The two comparator halves are then combined via a NAND gate into a single signal. This can be seen in Figure VII.5. Tag bits a_m and a_n belong to the probe address, and are compared, along with their inverses, to the tag bits of the cache. Each half of the comparator stage has e lines, where e is the number of entries (and the number of wordlines) in the cache subarray. Each of these lines has $x/2$ comparator pairs, where x is the number of address bits in a tag. Every address line in the comparator is initially precharged high, and if any bits in the probe address do not match the line address, the line is brought low. On a cache access, at most one line will remain high.

To maintain correct timing of the cache, a dummy line (shown at the bottom of Figure VII.5) is used with each subarray. The dummy line has the same comparators as a regular line, but one of the comparators is fixed to bring the dummy line low when the probe address bits arrive at the comparator. Only a single comparator is fixed to pull down the address line to model the maximum delay of the address line discharge. The dummy line then passes through an inverter and is used to enable the selection of a wordline using NOR gates. Each real address line is fed into its own NOR gate which controls access to the wordline driver that corresponds to the address line. Each address line is NORed together with the dummy gate to determine when to drive the wordline. This prevents wordlines from being driven before all probe address bits have arrived. Once a wordline is driven, the data path will behave exactly as a direct mapped cache.

To model the extra space required by the tag comparison stage, the tag cell height is doubled. Tag cells for the fully associative cache are $8\mu m$ by $32\mu m$, while data cells are $8\mu m$ by $16\mu m$.

Because each address bit must travel to a comparator on every address line in the fully associative cache decoder (since we must compare the address to every tag in the cache), a tiled layout approach must be used to reduce the length of the incoming address bit wires. Figure VII.6 shows our strategy. In this figure, we tile 16 cache subarrays and route wire using an h-tree strategy. The address lines are shown meeting at several black nodes - each node represents a buffering mechanism. The tile shown in grey is a single cache subarray. Using this approach, the worst case distance to reach a subarray is reduced from n to $log(n)$.

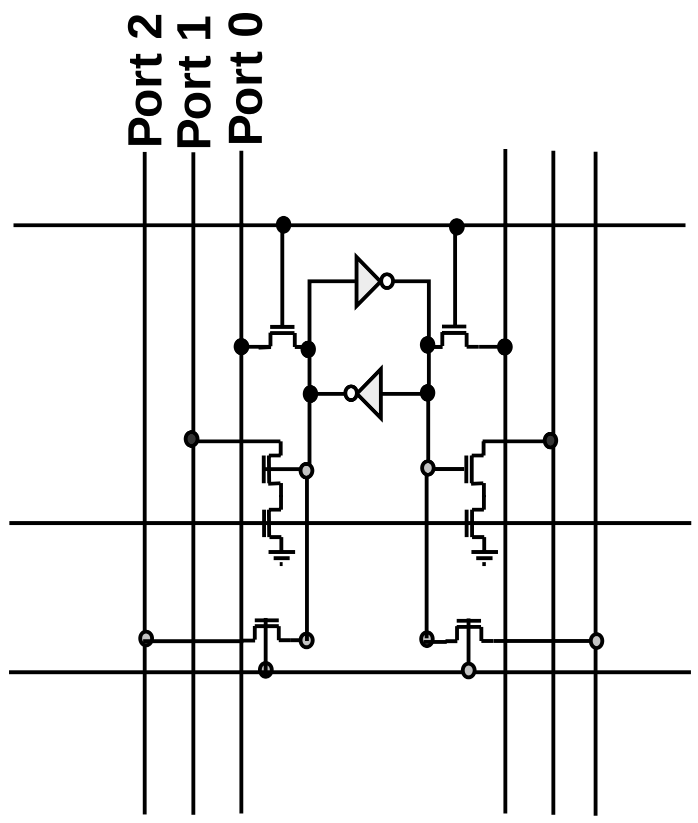


Figure VII.7: Multiple port example around a single SRAM cell. Port 0 is a read/write port. Port 1 is a read port. Port 2 is a write port. Each additional port impacts the cell size and wire lengths.

VII.A.4 Multiple Cache Ports

CACTI previously assumed a single read/write port on the cache model. We have expanded this model to allow the user to specify how many read/write ports (maximum of 2), read-only ports, and write-only ports to model on the cache. The extra ports are modeled as an increase to cell size, along with extra wordline and bitline lengths. All auxiliary structures (i.e. comparators, multiplexors) are assumed to be duplicated but are not included in the timing calculation. The auxiliary structures are included in the power model discussed in Section VII.B.

Figure VII.7 demonstrates a three port configuration on a single RAM cell. It consists of a read/write port (Port 0), a read port (Port 1) and a write port (Port 2). If the design were single-ended, it would not require both bit lines to be added for each port, however we do not model this.

The impact of extra ports on cell size is as follows:

For each extra read port:

- increase cell size by (2 × wire pitch) in the y direction (affects bitline metal)
- increase cell size by (wire pitch) in the x direction (affects wordline metal)

For each extra read/write or write port:

- increase by (2 × wire pitch) in both the x and y directions

For example, consider the following parameters:

$$Cmetal = 275fF$$
$$Rmetal = 48mO$$
$$wirepitch = 4\mu m$$

A cache with a single read/write port would have the following characteristics:

$$Cellsize = 8 \times 16$$

$$C_{bitmetal} = 4.4pF$$

A cache with two read/write ports would have the following characteristics:

$$Cellsize = 16 \times 24$$

$$C_{bitmetal} = 6.6pF$$

VII.A.5 Process Technology Scaling

As a minor tweak, the technology scaling factor already present in CACTI was made into a required command parameter. The user specifies the feature size (in microns) of the technology that is to be modeled, and the CACTI model scales measurements made at the $0.80\mu m$ size down (or up) to the desired technology size. This scaling factor affects both timing and power measurements.

VII.A.6 Cache Pipelining

In recent years, some microprocessors (e.g., the DEC 21264 [44]) have used caches that are effectively pipelined on half cycle boundaries. This can provide most of the functionality of true dual porting without the access time, power, and area penalty incurred by true dual porting. Because the cache arrays still take a full cycle to access, there are effectively two cache accesses in the cache arrays at any given time. No intermediate latches are placed in the cache, rather the cache is pipelined using an approach similar to wave pipelining [33]. Wave pipelining uses circuits that have similar minimum and maximum delays independent of input values to keep waves of logic values separate and distinct while traveling through the circuitry. Caches are particularly well suited to wave pipelining, since their access time is largely independent of the cache line being accessed.

Wave pipelining is only possible if one logic stage does not account for approximately 33% or more of the delay through the whole circuit. In the current timing model we compute the ratio of the maximum stage delay time to the total access time. If this is less than 0.333X, we assume the cache can be wave pipelined by a factor of two. This makes the cycle time equal to half the cache access time. If this is not possible, we report the minimum cycle time possible based on the maximum stage delay time.

VII.B Power Modeling

To more accurately assess the tradeoffs inherent in cache design, we extended the cache model in CACTI to model power consumption.

VII.B.1 Power Estimation

According to [69] and [90], energy consumption can be modeled as

$$E_{DD} = C_L \times V_{DD}^2 \times P_{0\rightarrow 1}$$

where C_L is the physical capacitance of a device and $P_{0\rightarrow 1}$ is the probability that the device will consume energy. We fully account for power dissipation when a capacitor is charged, and ignore discharge events. The energy value obtained from this formula can then be combined with the cycle frequency to provide the dynamic power consumption. For example, a device that consumes 3nJ of power and is clocked at 500MHz will consume 1.5W of power. Our goal with CACTI is to provide the energy consumption in nanoJoules, which can then be used to find the dynamic power consumption, depending on the frequency of the cache.

VII.B.2 Automatic Supply Voltage Scaling

CACTI requires the user to specify a technology size as a parameter to the model. Aside from being used to scale the access time reported by CACTI, this parameter scales the capacitances and the value of V_{DD} used by the power model. The value of V_{DD} is scaled by

$$V_{DD} = \frac{4.5V}{(\frac{0.8}{TECH})^{.67}}$$

Where $TECH$ is the feature size of the technology. This means that voltage will scale at a slower rate than capacitance and therefore than access time. The voltage level to which the bitlines are charged is calculated as a fraction of the scaled value of V_{DD}. We allow a maximum V_{DD} of 5V and a minimum value of 1V.

VII.B.3 Power Model

Since the CACTI model tracks the physical capacitance of each stage of the cache model, we use the energy consumption equation from Chapter VII.B.1 to calculate the power consumed at each stage. Additionally, we need to factor in the switching activity and the number of such devices in the cache (as CACTI models the activity down one particular path in the cache).

As an example, consider the power consumption modeled for the decoder on the data path of a set associative cache. CACTI models the decoder as being composed of three stages: the inverter that drives the probe address bit, the NAND gate that generates the 1-of-8 code, and the NOR gate that combines the 1-of-8 codes and drives the wordline driver (Figure 9 in [92]).

For the first stage, there are $log_2(\frac{C}{B \times A \times Ndbl \times Nspd})$ address bits, and we can estimate that a quarter of these will require the inverter to undergo a $0 \rightarrow 1$ transition (i.e. half of the address bits will be 0's and half of these were 1's before). However, we need both the true and complement forms of the address bits. So the energy consumption of the first stage can be represented as

$$E_{DD_1} = C_{stage1} \times V_{DD}^2 \times 0.25 \times log_2(\frac{C}{B \times A \times Ndbl \times Nspd}) \times 2$$

The next stage is composed of $\lceil \frac{1}{3} log_2(\frac{C}{B \times A \times Ndbl \times Nspd}) \times 2 \rceil$ blocks in each subarray. Each N_{3to8} block is composed of 8 NAND gates. We can estimate that half of these will undergo energy consuming switching. Since there are $Ndbl \times Ndwl$ decoders, the energy consumption is

$$E_{DD_2} = C_{stage2} \times V_{DD}^2 \times Ndbl \times Ndwl \times 4 \times \lceil \frac{1}{3} log_2(\frac{C}{B \times A \times Ndbl \times Nspd}) \rceil$$

Finally, the last stage is composed of the NOR gate that will drive a single wordline. Only one of the NOR gates in the decoder will be selected, which implies

$$E_{DD_3} = C_{stage3} \times V_{DD}^2$$

VII.B.4 Integration of Timing and Power Models

In the original version of CACTI, a cache configuration was chosen that optimized the access time of the cache. To optimize both power consumption and access time, we first generate the maximum values for each measurement over all configurations of a particular set of input parameters. Then, we iterate through the different configurations again, optimizing the following relationship:

$$\frac{access\_time}{maximum\_access\_time} + \frac{power\_consumption}{2 \times maximum\_power\_consumption}$$

We chose to divide the power ratio by a factor of two to emphasize optimization of the access time. This of course could be removed to optimize evenly across both measurements.

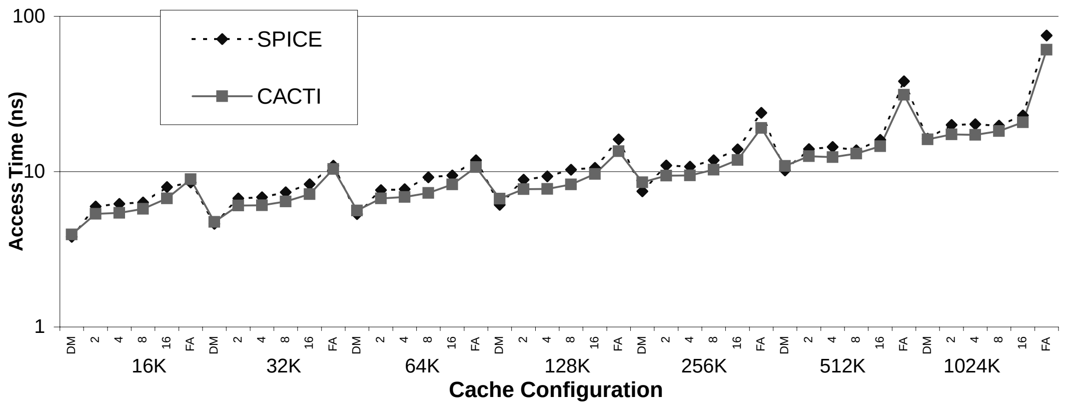


Figure VII.8: SPICE vs. CACTI – Access Time
Comparison of access times for a variety of cache configurations in both CACTI and SPICE. The y-axis is on a logarithmic scale, and shows the cache access time in nanoseconds. The x-axis ranges over a variety of cache configurations - all with 32 byte block sizes. Caches sizes of 16K, 32K, 64K, 128K, 256K, 512K, and 1024K are all shown, each broken down into six associativity configurations (direct mapped, 2-way, 4-way, 8-way, 16-way, and fully associative). So the first six points on the graph represent the six different 16K caches, then the next set of six corresponds to the 32K caches, and so forth.

VII.B.5 Prior Work

Kamble and Ghose proposed analytical models to estimate energy dissipation in [43]. They looked at using a simulation tool called CAPE which allowed them to track the transitions encountered in different components of the cache. They also investigated a number of architectural power reduction techniques.

VII.C Sample Results

First, we show a comparison of the results obtained with the new CACTI model to results obtained using SPICE. Then, we present CACTI results for a number of cache configurations and port configurations. The results presented in this section are for the $0.80\mu m$ technology size and are for caches with 32 byte block sizes, 32 bit addresses, and 64 bit outputs. Unless otherwise specified, the

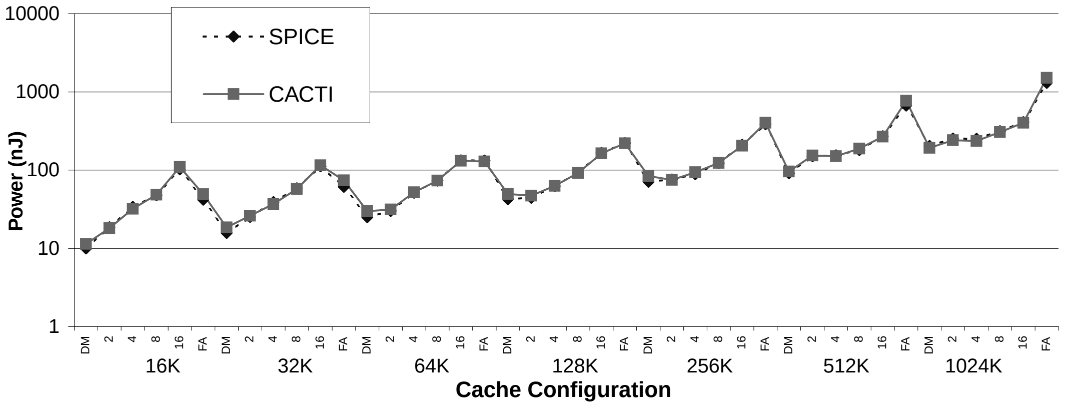


Figure VII.9: SPICE vs. CACTI – energy consumption
Comparison of energy consumption for a variety of cache configurations in both CACTI and SPICE. The y-axis is on a logarithmic scale, and shows the cache energy consumption in nanoJoules. The x-axis ranges over a variety of cache configurations - all with 32 byte block sizes. Caches sizes of 16K, 32K, 64K, 128K, 256K, 512K, and 1024K are all shown, each broken down into six associativity configurations (direct mapped, 2-way, 4-way, 8-way, 16-way, and fully associative).

caches in this section have a single read/write port.

VII.C.1 SPICE Verification

As a sanity check, we compare results obtained in CACTI to a cache model implemented in SPICE. Figure VII.8 compares cache access times in nanoseconds on a logarithmic scale across a variety of cache configurations for both models. Figure VII.9 compares cache energy consumption in nanoJoules on a logarithmic scale across the same cache configurations. The models show good correlation, even at larger cache sizes and associativities. These figures show how access time and energy consumption grow as cache size increases. The direct mapped cache configurations generally consume the least amount of energy and can be accessed in the fastest time for each different cache size. By comparison, the fully associative cache takes the longest time to access, but for small cache sizes has lower energy consumption than the 16-way set associative case.

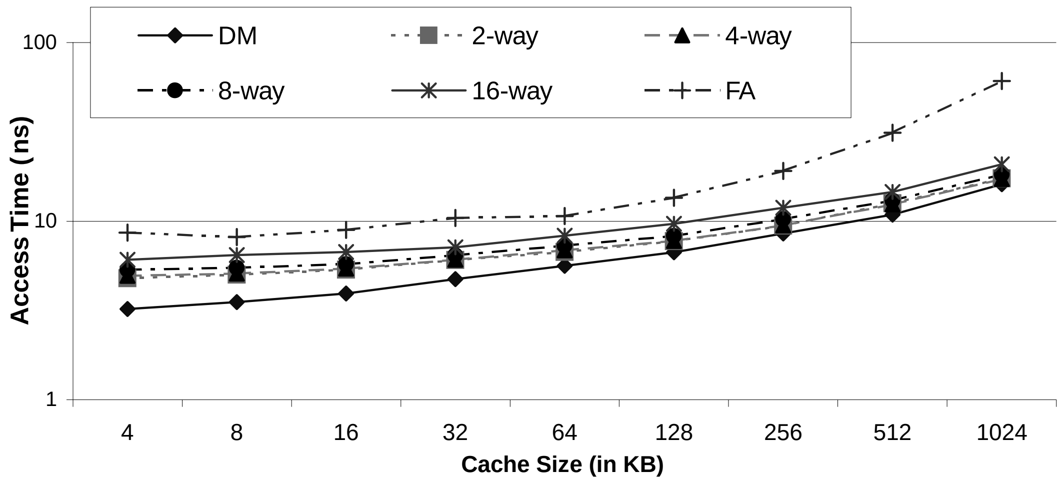


Figure VII.10: Access times for a variety of cache configurations. The y-axis shows the access time in nanoseconds on a logarithmic scale. The x-axis shows a range of cache sizes in KB. Six lines are plotted, each representing a different kind of associativity.

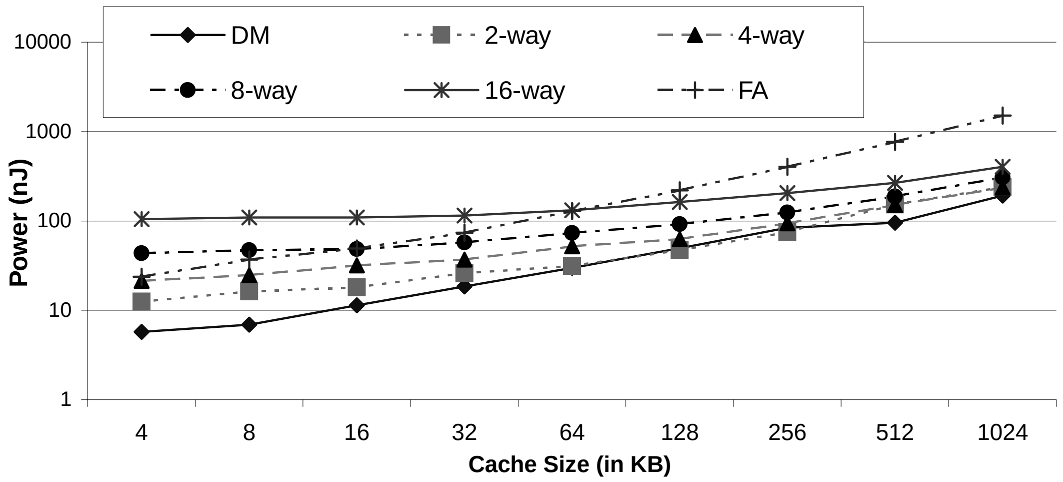


Figure VII.11: Energy consumption for a variety of cache configurations. The y-axis shows the energy consumed in nanoJoules on a logarithmic scale. The x-axis shows a range of cache sizes in KB. Six lines are plotted, each representing a different kind of associativity.

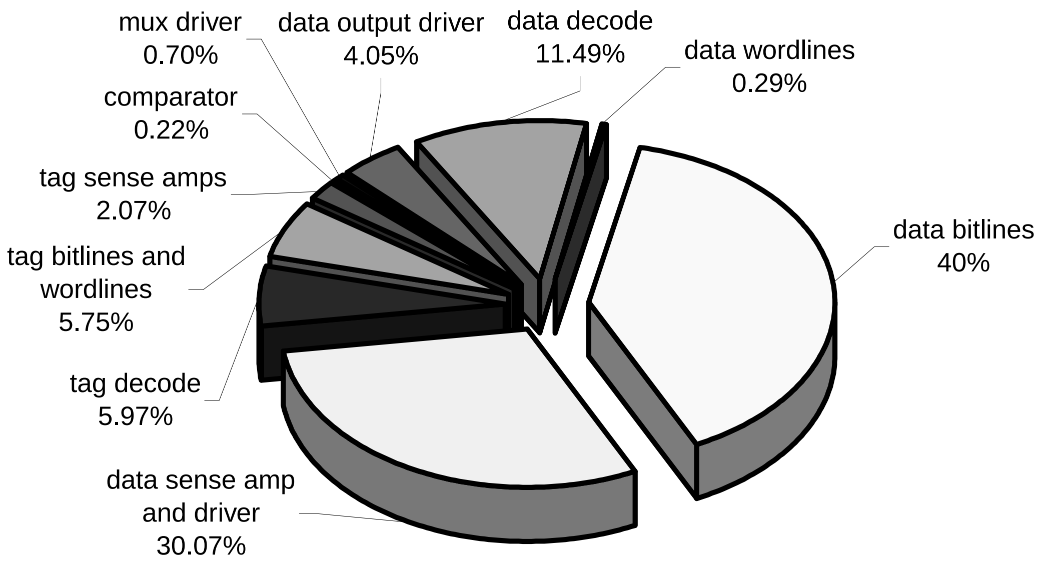


Figure VII.12: Breakdown of energy consumption for a 64K 2-way associative cache.
The data bitlines and sense amps are responsible for 40% and 30% of the energy consumption of the cache, respectively.

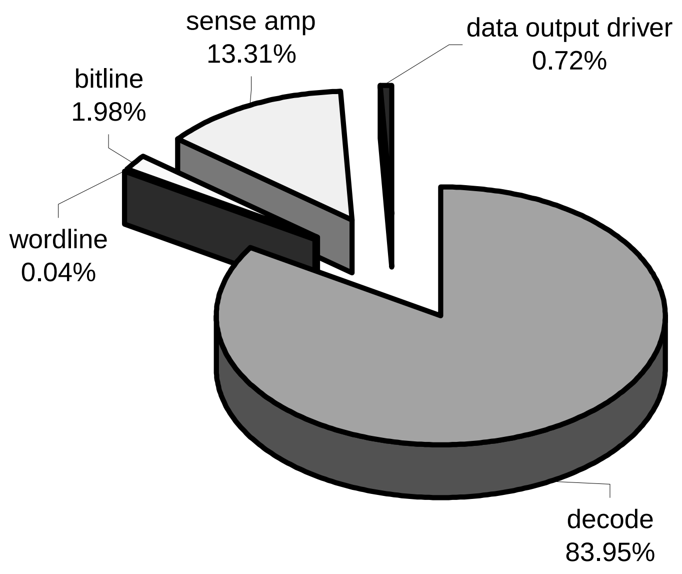


Figure VII.13: Breakdown of energy consumption for a 64K fully associative cache.
The decode portion of the cache (including tag comparisons) is responsible for 84% of the total cache energy consumption.

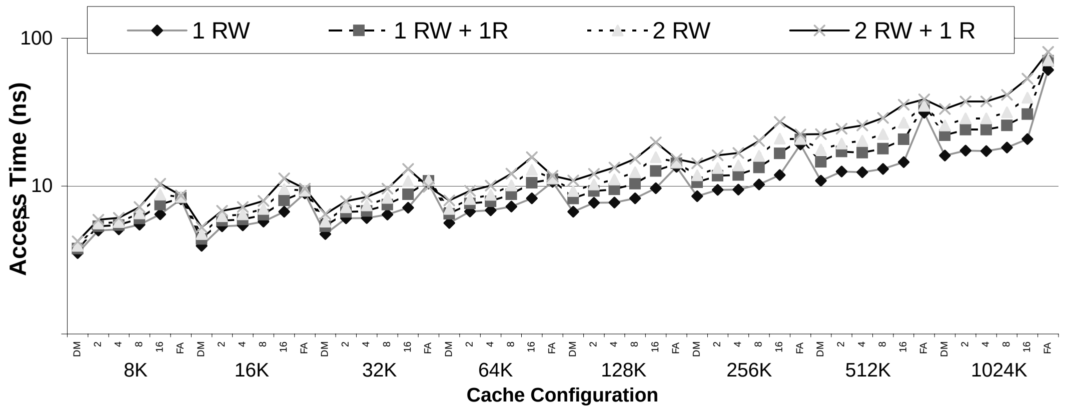


Figure VII.14: Comparison of access times for a variety of cache configurations. The y-axis is on a logarithmic scale, and shows the cache access time in nanoseconds. The x-axis ranges over a variety of cache configurations. Caches sizes of 16K, 32K, 64K, 128K, 256K, 512K, and 1024K are all shown, each broken down into six associativity configurations (direct mapped, 2-way, 4-way, 8-way, 16-way, and fully associative). Four lines are plotted: a single ported cache, a single ported cache with an extra read port, a dual ported cache, and a dual ported cache with an extra read port.

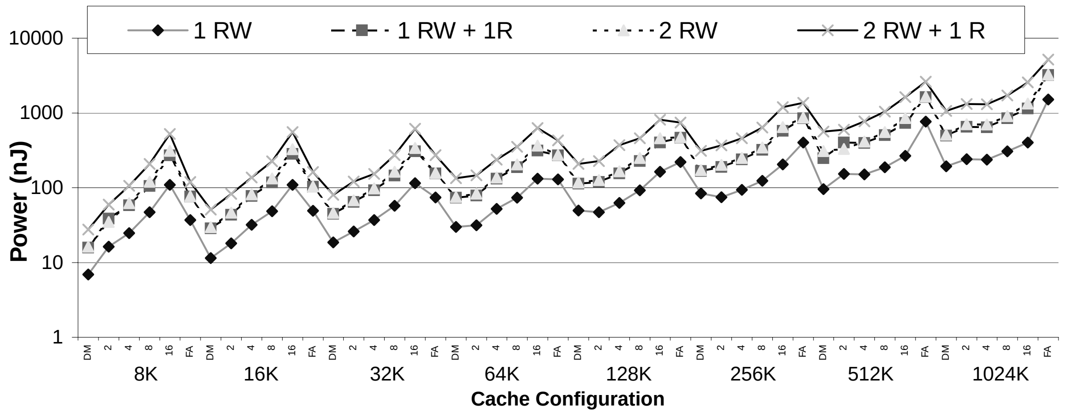


Figure VII.15: Comparison of energy consumption for a variety of cache configurations.
The y-axis is on a logarithmic scale, and shows the cache power consumption in nanoJoules. The x-axis ranges over a variety of cache configurations. Caches sizes of 16K, 32K, 64K, 128K, 256K, 512K, and 1024K are all shown, each broken down into six associativity configurations (direct mapped, 2-way, 4-way, 8-way, 16-way, and fully associative). Four lines are plotted: a single ported cache, a single ported cache with an extra read port, a dual ported cache, and a dual ported cache with an extra read port.

VII.C.2 Timing Results

Figure VII.10 shows access times for the cache configurations we selected. It is interesting to note the similarity in access times between the 2-way and the 4-way associative caches. Increasing the associativity of the cache does increase the number of sense amps that must be used and increases the number of bitlines connected to a single wordline, but it also reduces the number of rows in the decoder. For many cases, the 2-way set associative case proved to perform better with a higher value of Ndbl or Nspd, which effectively reduces the number of rows in the decoder in much the same way as increasing the associativity. However, these also carry the same detrimental effects as increasing the associativity. Unfortunately though, increasing the number of subarrays has an additional consequence - increasing the degree of multiplexing at the bitline column multiplexors. Since we limit the degree of multiplexing that can occur at these multiplexors, the more subarrays there are, the less the column multiplexors can filter output bits from the cache line (Section VII.A.2). This can have a detrimental effect on the performance of the tag path, as it will increase the delay of the comparator and output multiplexors.

This is seen in the 512K cache. Here, the 4-way case performs as well as the 2-way case (12.4ns compared to 12.6ns), despite the increase in associativity. Both have 8 bitline divisions (Ndbl), but the 2-way case has an Nspd of 2 (effectively mapping two sets to a single wordline). This means that they have the same number of rows in their decoders, and effectively the same amount of decoder and wordline delay. However, since we limit the degree of column multiplexing to 16, the 2-way case is unable to filter out the same number of output bits as the 4-way case. But, since the 4-way case has a higher associativity, both cases end up with the same number of sense amplifiers and output drivers. So the data path delay for both is identical. The tag path differs slightly though.

The 2-way case has better decoder and tag array performance, at the cost of comparator performance (due to the increase in Ntspd). Additionally, the extra output bit multiplexing that it must perform increases the delay of the output multiplexors. This causes the 2-way case to have a slightly higher tag path delay than the 4-way case (but only by about 0.2 ns).

VII.C.3 Power Results

Figure VII.11 shows power consumption for the cache configurations we selected. A majority of power dissipated by the set associative configurations is in the bitlines and sense amplifiers. Therefore, as the number of sense amps required grows (in response to the number of subarrays, the associativity, the value of Nspd, etc) the power consumption also grows. However, for the fully associative configuration, most of the power is consumed in the decode stage (where the tag check is performed). However, the fully associative case does not require a significant number of bitlines to discharge - only enough for a single cache line. This is due to the fact that each wordline only maps to a single cache entry. Therefore, at smaller cache sizes, the fully associative cache uses less power than a highly associative cache (like the 16-way case) as it has substantially less bitline activity. Moreover, a highly set associative will also require more sense amps than the fully associative case. At larger cache sizes, the delay of the fully associative decoder grows considerably and it consumes the most power of any cache associativity configuration we investigated.

Figure VII.12 shows a breakdown of power consumption for a 64K 2-way associative cache. The data path is the predominant power consumer in this case - 70% is consumed by the data bitlines (40%) and sense amps (30%). There are 128 sense amps in this cache configuration, and 512 pairs of bitlines. There are only 128 rows in the 4 data decoders, and as can be seen in the figure, the data decoder is only responsible for 11% of the total power consumption.

Figure VII.13 contrasts this, as 84% of the total power consumption is found in the decoder. In a fully associative cache, the decoder also performs the tag check and has a wordline for every entry in the cache. There are 2048 rows in this case - 64 in 32 decoders. Each of these rows has a address comparison line that must be precharged after every unsuccessful tag check. In this model, there are 256 sense amps - but these are still only responsible for 13% of the total power consumption, despite the fact that they are a significant source of power consumption (consuming around twice as much power as the sense amps in the 2-way set associative case, as would be expected). However, there are only 32 pairs of bitlines in this case, and they are only responsible for 2% of the total power consumed.

VII.C.4 Multiported Results

Figure VII.14 shows the access times for four different port configurations. Figure VII.15 shows the power consumption for these configurations. We examined a variety of cache configurations using a single ported cache, a single ported cache with an extra read port, a dual ported cache, and a dual ported cache with an extra read port. As can be seen, additional ports lengthen cache access times, especially for large sized caches. An extra read only port provides a slightly smaller increase in access time over an extra read/write port. Moreover, adding extra ports has a tremendous impact on cache power consumption. Because the extra port effectively implies replicating most cache structures and because of the extra physical area involved, the additional power required by a second port is often greater than the total power of a single ported version of the cache.

VII.D Modifications for this Thesis

First, we modified CACTI to handle more than just the time for a successful cache access. We modeled cache misses, cache probes, and cache writes. In addition, we added the ability to add extra ports just to the tag array of the cache.

In this thesis it was necessary to expand the CACTI 2.0 model further to handle other cache-like structures of the front-end. Gshare predictors, BTB-like structures, and queues were all modeled using this tool to estimate timing and energy data. For 2-bit predictors like the Gshare, we assumed that a number of 2-bit entries would share a common wordline, rather than having an extremely thin and extremely tall structure on chip. Then we ignored the tag path in CACTI to obtain timing and energy data. For the BTB-like structures, minor changes were necessary – just a change in the number of bits that the cache-like structure would output. We handled queues as direct mapped caches without tag arrays.

Additionally, we modified CACTI 2.0 to handle different types of instruction cache configurations. As will be shown in Chapter X, we looked at using a pseudo-associative cache and a multi-component serial cache in place of the original set-associative instruction cache. These modifications will be explored more in Chapter X.

Chapter VIII

Decoupled Front-End Architecture

As discussed in Chapter III, the branch predictor and instruction cache in a contemporary processor architecture are coupled together – if the instruction cache stalls, the branch predictor must also stall. This is shown in Figure VIII.1. In this Chapter, we improve the scalability and performance of the front-end by decoupling the branch predictor from the instruction cache. A *Fetch Target Queue* (FTQ) is inserted between the branch predictor and instruction cache, as seen in Figure VIII.2. The FTQ stores predicted fetch addresses from the branch predictor, later to be consumed by the instruction cache. The FTQ serves two primary functions: latency tolerance and fetch stream look-ahead.

VIII.A Fetch Target Queue

To provide a decoupled front-end, a *Fetch Target Queue* (FTQ) [73] is used to bridge the gap between the branch predictor and the instruction cache. The FTQ stores predictions from the branch prediction architecture until these predictions are consumed by the instruction cache or are squashed. On a branch

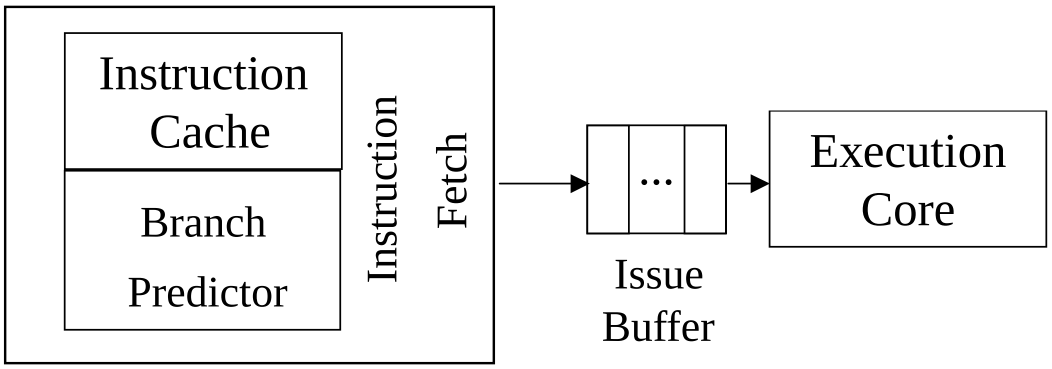


Figure VIII.1: Contemporary high level processor design.
The instruction fetch unit prepares and decodes instructions and supplies them to the issue buffer. The execution core consumes instructions from the issue buffer and then orchestrates their execution and retirement. The instruction fetch unit is a fundamental bottleneck in the pipeline: the execution core can only execute instructions as fast as the instruction fetch unit can prepare them. The instruction fetch unit in this architecture contains a coupled branch predictor and instruction cache.

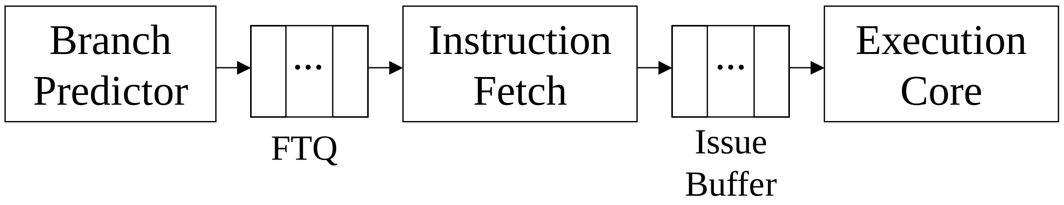


Figure VIII.2: The decoupled front-end high level design.
The fetch target queue (FTQ) buffers fetch addresses produced by the branch predictor. They are queued in the FTQ until they are consumed by the instruction fetch unit, which in turn produces instructions as in an ordinary pipeline. The FTQ allows the branch predictor to continue predicting in the face of an instruction cache miss. It also provides the opportunity for a number of optimizations, including multi-level branch predictor designs and fetch directed cache prefetching.

misprediction, the FTQ is flushed.

The addition of the FTQ does not necessarily imply an extra pipeline stage however. The instruction cache can recover from a branch misprediction in the same way that a normal decoupled cache would recover from a misprediction – it would provide the block that contains the redirected fetch address. This implies an extra degree of complexity to orchestrate such a recovery. The alternative is to simply extend the pipeline by an additional cycle, and to maintain a strict producer/consumer relationship between the branch prediction architecture and instruction cache. In this thesis we assume that the instruction cache and branch predictor can both be fed the corrected fetch address on a misprediction – thereby eliminating the additional branch misprediction penalty. Figure VIII.3 illustrates this further.

In the rest of this Chapter we examine some of the issues related to the operation of the FTQ, including the occupancy of the FTQ (number of fetch addresses stored in the FTQ) and the speculative recovery mechanisms that we use in coordination with the FTQ.

VIII.A.1 FTQ Occupancy

The occupancy of the FTQ significantly contributes to the amount of benefit obtained from the decoupled front-end architecture. We define the occupancy of the FTQ to be the total number of branch predictions contained in the FTQ. If there are a large number of cache blocks represented by the predictions in the FTQ, the instruction cache will have plenty of predictions to consume, and we will have more flexibility in the branch predictor implementation (whether it be a multi-level design or a multi-cycle access design). Moreover, the higher the FTQ occupancy, the further ahead cache prefetching mechanisms and PC-based predictors can look into the future fetch stream of the processor.

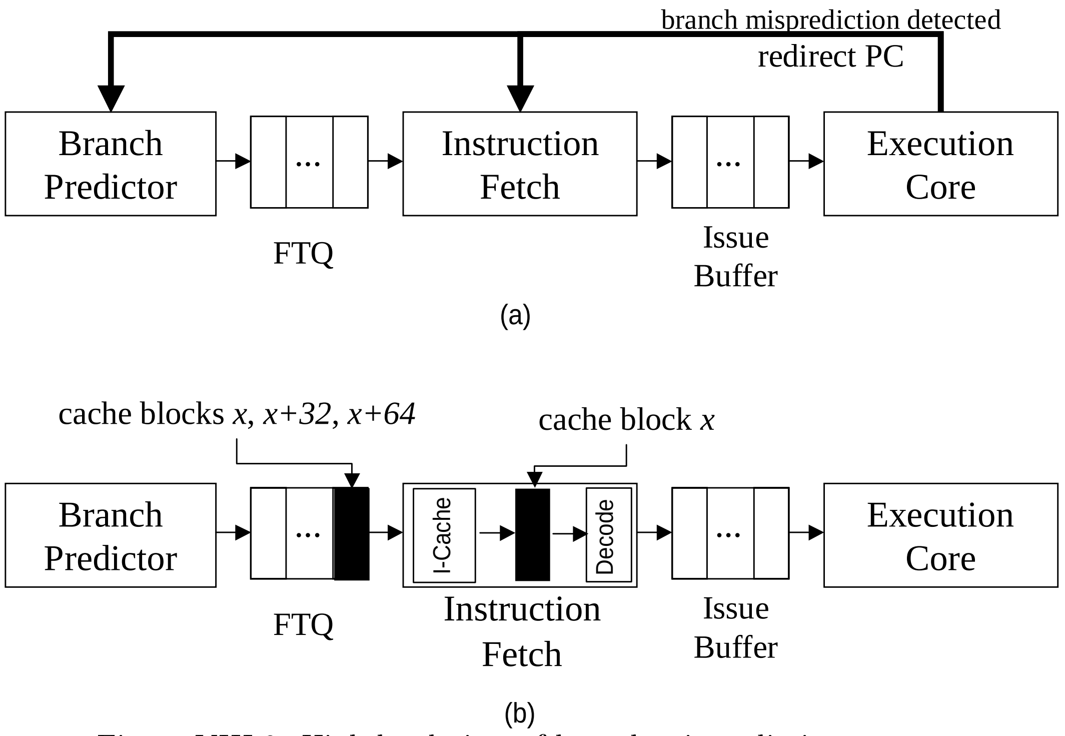


Figure VIII.3: High level view of branch misprediction recovery
In this Figure, we show how the decoupled front-end is restarted on a branch misprediction. The upper pipeline (a) shows the execution core detecting a branch misprediction and sending the recovery PC to the branch prediction architecture and to the instruction cache (in the instruction fetch unit). Then, as shown in the lower pipeline (b), both the branch prediction architecture and instruction fetch unit can be restarted. The branch predictor provides a fetch block spanning three cache blocks (in this example). The instruction cache provides the first of these cache blocks (the block which contains the redirect PC) to the decode hardware. In the next cycle, the decode hardware can make use of the branch prediction to potentially mask out bits in the cache block provided the cycle before (cache block x). The instruction cache can grab the next prediction (cache block x+*32*) and provide that block to the decode hardware. The instruction cache basically ignores the first prediction of the branch prediction architecture.

High levels of occupancy can be obtained through the prediction of large fetch blocks, by predicting multiple branches in a single cycle, during instruction cache misses, and as a result of a full instruction window. Larger fetch blocks mean that each prediction will carry more instructions, fundamentally increasing FTQ occupancy. Instruction cache misses delay consumption of FTQ entries, but the decoupled branch predictor will continue to produce predictions and fill the FTQ. An instruction window that is full due to data dependencies or even limited resources can slow the instruction fetch unit as well, thereby allowing the FTQ to fill. While these two latter situations are by no means desirable, an architecture can still take advantage of these to provide more FTQ occupancy.

In our study, we look at an FTQ that stores *fetch blocks*. A fetch block is a sequence of instructions starting at a branch target, and ending with a branch that has been taken in the past. Branches which are strongly biased not-taken may be embedded within fetch blocks. This type of block prediction was also examined by Michaud et. al. [58]. An FTQ could conceivably hold any form of branch prediction – though it is most useful when looking at larger prediction blocks, such as fetch blocks, as these will increase the occupancy of the FTQ. Chapter IX will examine the difference between fetch blocks and basic blocks more closely when we compare our branch prediction architecture to prior work.

VIII.A.2 Speculative Recovery Structures

In this section we explore two different recovery mechanisms that are needed to maintain predictor accuracy in our decoupled front-end architecture. These structures are illustrated in Figure VIII.4.

For good predictor performance, especially for machines with deep speculation and large instruction windows, it is beneficial to recover branch history when the processor detects a mispredicted branch. This is even more important

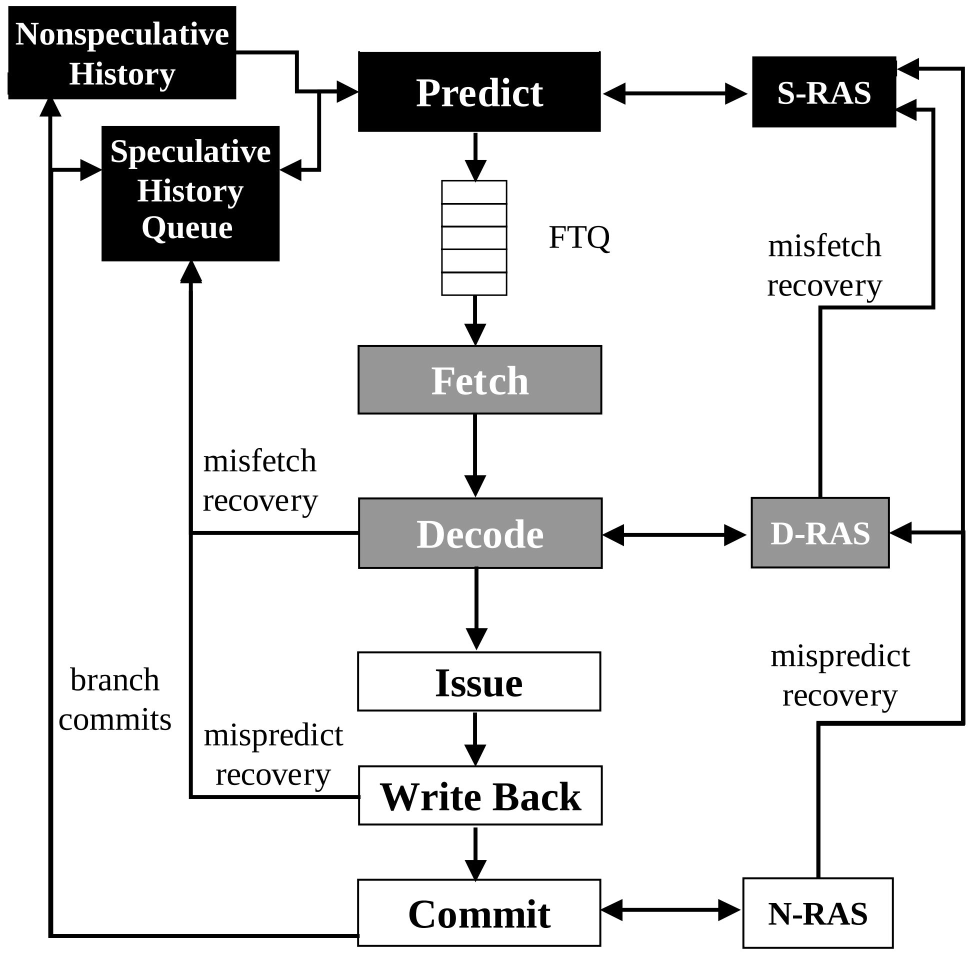


Figure VIII.4: Speculative support structures.
This figure presents a simplified processor pipeline along the center (including the FTQ). Black, grey, and white boxes represent stages in the branch prediction, instruction fetch, and execution core portions of the pipeline. The nonspeculative and speculative history structures are shown along the left side of the pipeline. Speculative history is stored in the speculative history queue (SHQ) and nonspeculative history is stored in the various predictors of the branch prediction architecture. The SHQ is used to recover the nonspeculative history in the event of a branch misfetch or misprediction. Both speculative and nonspeculative history are probed in parallel by the branch prediction stage of the pipeline. Along the right side of the pipeline, three different return address stacks are shown: the S-RAS, D-RAS, and N-RAS. The S-RAS is updated during the branch prediction stage, the D-RAS is updated during the decode stage, and the N-RAS is updated during the commit stage.

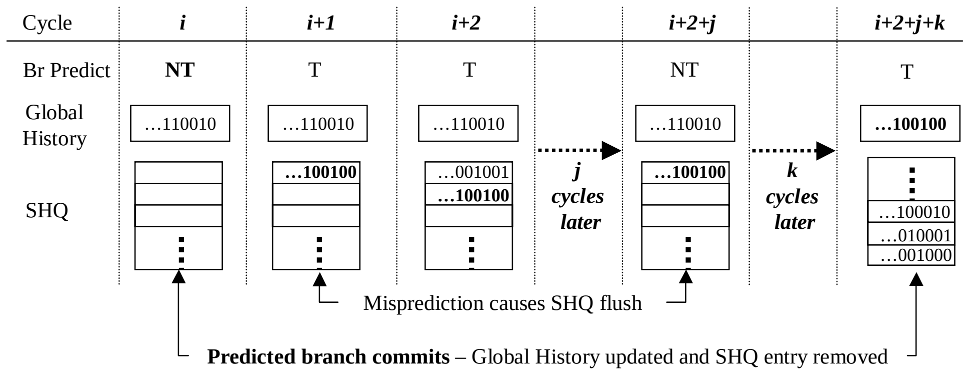


Figure VIII.5: Speculative History Queue (SHQ) example

This example illustrates the use of the SHQ for global history branch predictors. This figure shows the state of the non-speculative global history register (GHR) and the SHQ over several cycles of program execution. Cycles are delineated by vertical dotted lines and are labeled at the top of the figure. The current branch prediction is noted next, followed by the current state of the non-speculative global history register, followed by the current state of the SHQ. In this example, the SHQ contains the speculative state of the global history register. The trace begins at cycle *i*, where the SHQ is currently empty. Here, the current branch prediction is not taken (NT) and the global history register contains the current branch history up until the current prediction. In cycle *i+1*, the next prediction is taken (T) and the global history has remained the same as no new branches have committed. However, a new history entry has been added to the SHQ to reflect the speculatively modified global history (from the prediction in cycle *i*). Similarly, the SHQ in cycle *i+2* contains a new entry that reflects the addition of branch prediction information from cycle *i+1*. In this example, the branch at cycle *i* is correctly predicted and the branch at cycle *i+1* is mispredicted. Therefore, at cycle *i+2+j*, all SHQ entries corresponding to predictions made after cycle *i* are flushed due to the misprediction at cycle *i+1*. As the mispredicted branch had never committed, the global history counter itself does not need recovery. Finally, in cycle *i+2+j+k*, once the correctly prediction branch in cycle *i* has committed, the global history register is updated with the information stored in the SHQ, and the entry corresponding to that branch is removed from the tail of the SHQ.

in our scalable front-end architecture design, because the branch predictor can get many predictions ahead of the instruction fetch unit. To facilitate the recovery of branch history, we make use of a small *Speculative History Queue* (SHQ) [70] to hold the speculative history of branches. When branches are predicted, their updated global or local history is inserted into the SHQ. When predictions are made, the SHQ is searched in parallel with the branch predictor — if a newer history is detected in the SHQ, it takes precedence over the current global history or the local history in the branch predictor. When the branch at the end of a fetch block retires, its branch history is written into the global history register or the branch predictor for local history and its corresponding SHQ entry is removed. When a misprediction is detected, the entry in the SHQ at which the mispredicted branch occurred and all entries allocated after the misprediction are released. The SHQ is kept small (32 entry) to keep it off the critical path of the branch predictor. If the SHQ fills up, then the global history register and/or local history registers in the branch predictor are speculatively updated with the oldest entry in the SHQ. This allows new SHQ entries to be inserted, at the price of potentially updating the history incorrectly. Skadron *et al.*, independently developed a similar approach for recovering branch history, and they provide detailed analysis of their design in [80]. Figure VIII.5 gives an example of the operation of the SHQ for global history.

Since the branch predictor can make predictions far beyond the current PC, it can pollute the return address stack beyond the means of normal recovery techniques if it predicts multiple calls and returns. It is necessary to use sophisticated recovery mechanisms to return the stack to the correct state. Ordinarily, the architecture keeps track of the top of the return address stack and restores this to recover from a misprediction [79]. But our predictor may encounter several returns or calls down a mispeculated path that will affect more than just the top of stack. We use three return address stacks to solve this problem. One is speculative (S-RAS) and is updated by the branch predictor during prediction. The next (D-RAS) is also speculative and is updated in the decode stage. The third is non-speculative (N-RAS) and is updated during the commit stage. The S-RAS can potentially be corrupted by branch misfetches and branch mispredictions. The D-RAS can potentially be corrupted by branch mispredictions - as misfetches will be detected in the decode stage. The N-RAS will not be corrupted by control hazards, as it is updated in commit. When a misfetch is detected, the S-RAS will likely be polluted and can be recovered from the D-RAS. When a misprediction is detected, the S-RAS and D-RAS will likely be polluted and can be recovered from the N-RAS. After either situation, prediction can restart as normal, using the S-RAS. This is illustrated in the simplified pipeline in Figure VIII.4. Chapter IX demonstrates the prediction accuracy of our decoupled branch predictor with the help of these recovery structures.

VIII.B Results

In this section we explore the benefit of adding the FTQ to the processor pipeline. Here we will only consider the benefit of the FTQ itself, and the Chapters following this will illustrate the potential for further optimizations such as instruction cache prefetching and multilevel branch predictors.

The base architecture for this study is as described in Chapter VI. For this Chapter, a 16K 2-way set associative instruction cache with a single read/write port is used. For this section a 4096 entry single level fetch target buffer is used for the branch prediction architecture. The fetch target buffer will be described in detail in Chapter IX. This predictor is large enough to provide accurate fetch block prediction.

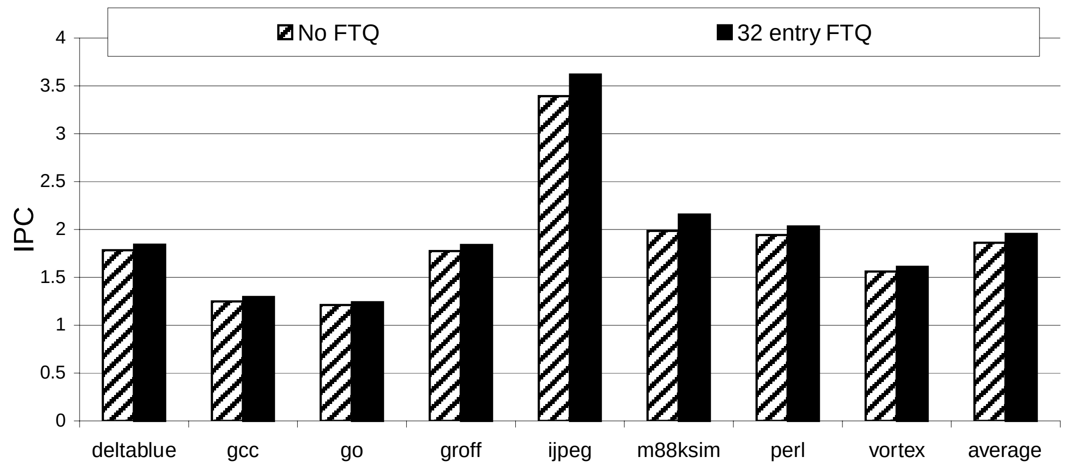


Figure VIII.6: FTQ performance comparison.
This figure presents a comparison between an architecture with a traditional coupled front-end and one that has been decoupled by the FTQ. For this comparison, we make use of a 32 entry FTQ. The y-axis shows instructions per cycle (IPC) and the x-axis shows a selection of 7 benchmarks. A large, single level branch prediction architecture is used here.

VIII.B.1 Performance

Figure VIII.6 shows the benefit in IPC obtainable from just adding an FTQ to the processor pipeline. There are a number of situations where extra queued branch predictions can provide performance benefits. One such situation was shown in Figure III.2 (Chapter III). Programs with frequent branches experience the most benefit from this technique (such as `m88ksim` which has an 8% improvement from the addition of the FTQ). On average, we see a 5% improvement in performance, just through the addition of the FTQ.

VIII.B.2 FTQ Occupancy

The greater the occupancy of the FTQ, the greater the potential benefit that can be provided by the FTQ using a variety of optimizations. Figure VIII.7 shows a breakdown of the occupancy of the FTQ for a large single level branch predictor. This figure demonstrates the percent of cycles that the FTQ held a

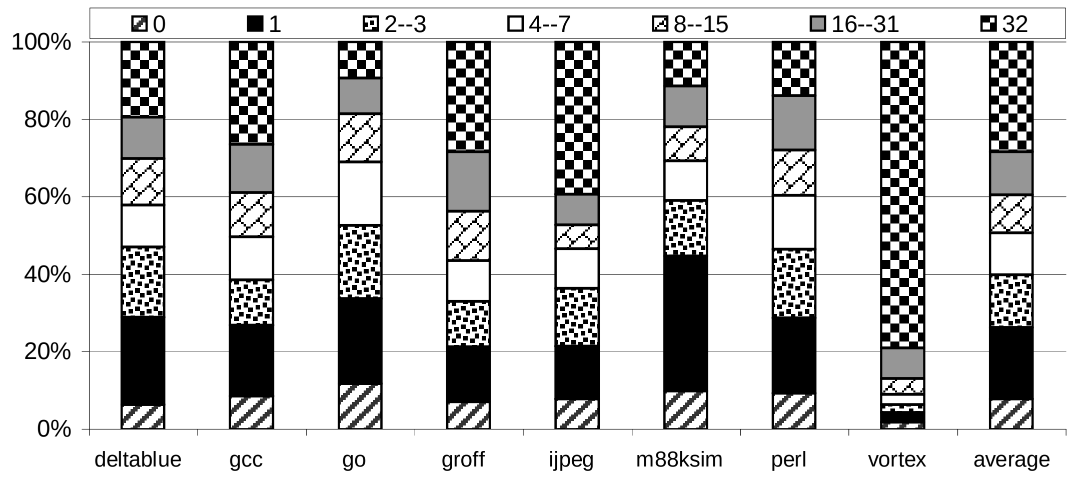
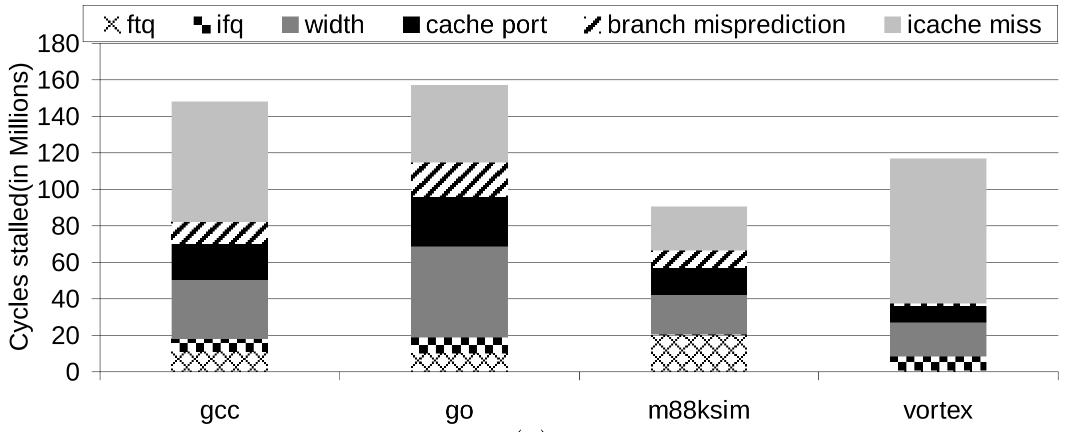


Figure VIII.7: FTQ occupancy histogram.
This graph shows the percent of cycles that the 32 entry FTQ used in this study had a given number of predictions stored within it after instruction fetch. The disjoint categories of FTQ occupancy are shown in the legend at the top of the graph. For example, the white bar component represents the percent of time that the FTQ was empty. The next component represents the percent of time that the FTQ contained exactly one prediction. The next component represents the percent of time that the FTQ contained exactly two or three predictions. The predictor used in this Figure is a large, single level branch prediction architecture (an 8192-entry FTB). In Chapter IX we will examine the impact of the branch prediction architecture on the occupancy of the FTQ.

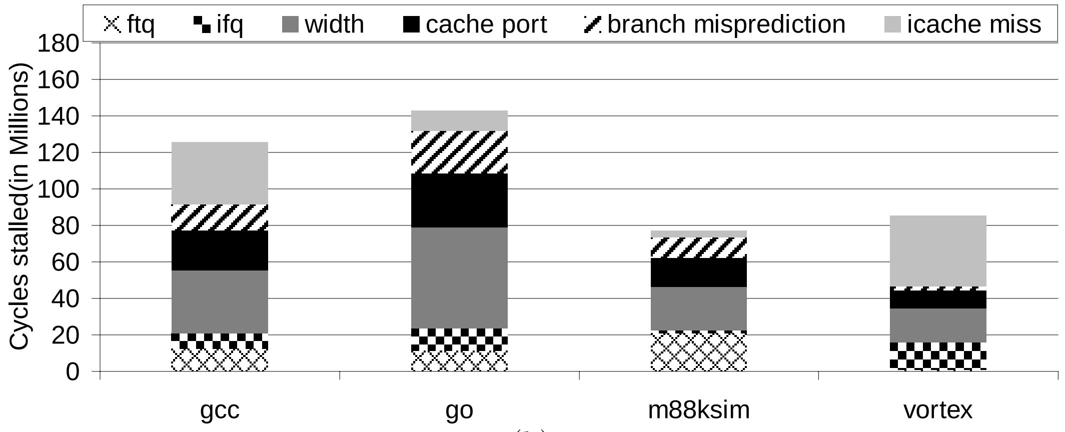
given number of entries with predictions. For example, the FTQ only buffered a single prediction 20% of the time on average.

VIII.B.3 Sources of Occupancy

Figure VIII.8(a) provides a look at the breakdown of stalls for the instruction fetch unit. This can provide insight into some of the sources of FTQ occupancy. The data shown in the Figure is collected by keeping a count of what eventually stops the instruction fetch from producing instructions each cycle. For example, the black component of the bar represents the number of times the instruction fetch was unable to continue due to insufficient ports on the instruction cache and the light grey component at the top of the bar represents the amount of time the instruction cache was forced to stall due to an instruction cache miss. Even if there are no instruction cache stalls, no port stalls, plenty of free IFQ entries, plenty of bandwidth from the branch predictor, and no branch mispredictions, there would still be a physical limit on how many instructions could be fetched in a given cycle. So collectively, the total number of cycles stalled equals the total number of cycles that the benchmark took to execute. This is directly related to the IPC of the particular run. As can be seen, a full instruction fetch queue is a relatively rare occurrence and is therefore not a critical source of FTQ occupancy, for the benchmarks and architecture we examine. Instruction fetch stalls vary according to each benchmark. Benchmarks with poor branch prediction, like `go`, experience a significant amount of stalls from branch mispredictions. Benchmarks with frequent branches, like `m88ksim`, experience a significant amount of stalls from the FTQ – but almost none from a full instruction fetch queue. While all benchmarks experience significant stalls from instruction cache misses, `vortex` experiences a majority of its stalls from instruction cache misses (and therefore can benefit enormously from accurate prefetching).

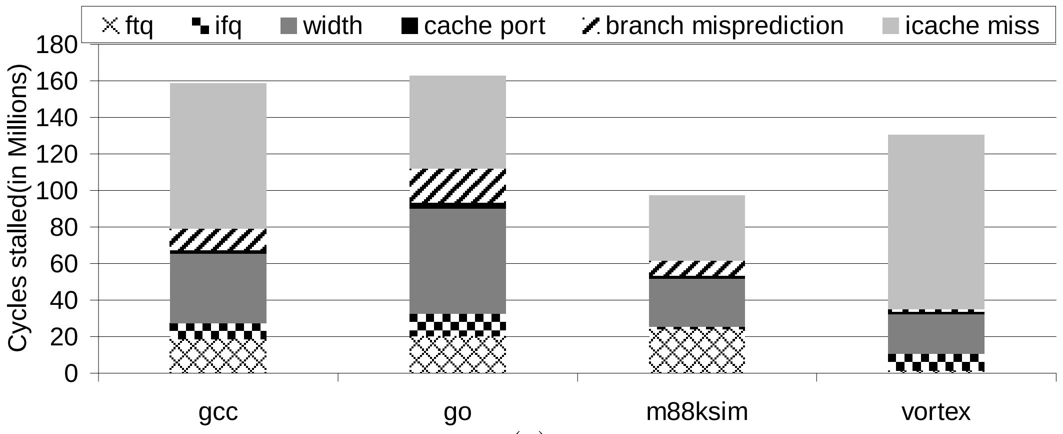


(a)

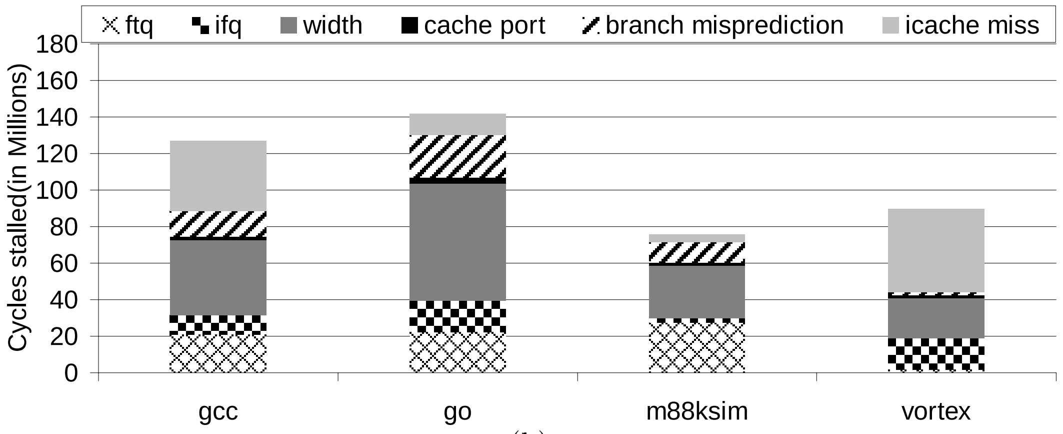


(b)

Figure VIII.8: Sources of FTQ occupancy (single ported caches).
These graphs show the total number of execution cycles for a given program, broken down by the stalls in the instruction fetch unit. The instruction fetch unit consumes predictions from the FTQ and deposits instructions in the instruction fetch queue. Starting from the bottom of each bar, the categories include: stalls due to an empty ftq (*i.e.* insufficient branch predictor bandwidth), stalls due to a full instruction fetch queue (*i.e.* a slow execution core), stalls due to insufficient fetch width (physical limitation on how many instructions can be fetched in a given cycle), stalls due to insufficient ports on the instruction cache, stalls due to branch predictor mispredictions (this is actually wasted bandwidth as the processor pipeline drains), and stalls due to instruction cache misses. The y-axis gives the cycles stalled in millions. The x-axis lists the benchmarks evaluated. (a) provides data for a 16K 2-way set associative cache. (b) provides data for a 32K 4-way set associative cache. Both (a) and (b) use a single ported instruction cache.



(a)



(b)

Figure VIII.9: Sources of FTQ occupancy (dual ported caches).
These graphs show the total number of execution cycles for a given program, broken down by the stalls in the instruction fetch unit. This is identical to Figure VIII.8, except that we use a dual ported instruction cache. (a) provides data for a 16K 2-way set associative cache. (b) provides data for a 32K 4-way set associative cache.

Branch mispredictions and frequent unbiased or taken branches have a negative impact on FTQ occupancy. Limited fetch width, instruction cache mispredictions, limited instruction cache ports, and full reorder buffers (or instruction fetch queues) have a positive impact on FTQ occupancy. These latter stalls comprise a significant amount of overall instruction fetch stalls and allow high levels of FTQ occupancy.

An improvement in one or more of the potential stall areas will not necessarily reduce the cycles stalled by the processor. If instruction cache misses are reduced for example, there may still be an insufficient number of cache ports or insufficient fetch width. Figure VIII.8(b) demonstrates the effect of using a larger instruction cache on the processor stalls. Here, a 32K 4-way set associative cache is used. The benchmark `gcc` experiences a 53% decrease in instruction cache miss stalls, but only a 19% decrease in the total cycles needed to execute the same data set since other factors caused the instruction fetch unit to stall. As would be expected, `m88ksim` experiences the smallest amount of improvement from a larger instruction cache since it had relatively few instruction cache misses to begin with.

Figures VIII.9(a) and (b) present the effect of adding an additional read/write port to the instruction cache for a 16KB 2-way set-associative cache (a) and a 32KB 4-way set-associative cache (b). In (a), the stalls for each benchmark actually *increase* with the addition of the extra port. The stalls due to insufficient cache ports almost completely disappear, but there are significant increases in stalls due to instruction cache misses, insufficient FTQ occupancy, and insufficient IFQ space. The latter two are due to the fact that the increased bandwidth in the instruction fetch unit are not matched by increased bandwidth from both the branch prediction architecture and the execution core. However, the extra instruction cache miss stalls are due to the increased fetch unit bandwidth.

On a branch misprediction, twice as many instruction blocks from incorrect paths can now potentially pollute the instruction cache. The negative impact of cache pollution in this case outweighs the benefit of the greater instruction fetch bandwidth, especially without a corresponding increase in branch prediction bandwidth and execution bandwidth. Figure VIII.9(b) shows the impact of adding an extra port to a larger cache. Here, `go` and `m88ksim` actually experience a very slight reduction in overall stalls. Even though these benchmarks have relatively poor branch prediction behavior, they have very few instruction cache miss stalls and therefore are not as affected by cache pollution with the addition of an extra read/write port. However, they are both heavily impacted by stalls due to insufficient FTQ bandwidth, which does prevent a more significant decrease in overall stall cycles. The benchmarks `vortex` and `gcc` are still impacted by pollution effects when adding an extra port. In the case of these benchmarks, it might make more sense to spend the extra area that would be required to dual port the cache on increasing the cache size.

VIII.B.4 FTQ Size

Another factor that must be considered is the size of the FTQ. A two entry FTQ is sufficient to provide the benefit shown in Figure VIII.6, but as will be shown in Chapter X, more entries enable the branch predictor to run further ahead of the current PC, and allow an instruction prefetching scheme to provide considerable benefit.

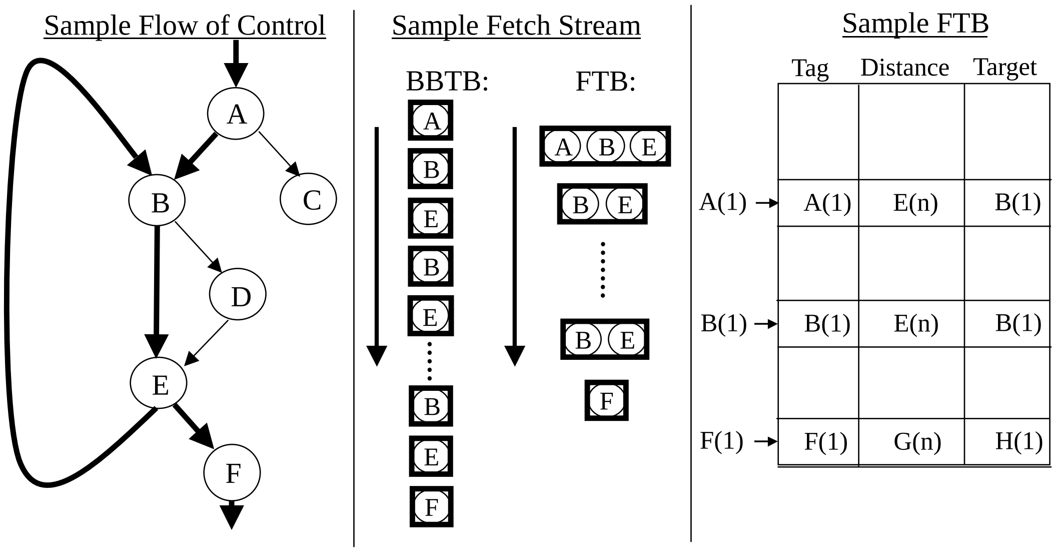
Chapter IX

Branch Predictor Optimizations

The use of a decoupled design provides us with some flexibility in branch predictor design. Because the FTQ buffers predictions made by the branch predictor, it is possible to hide the latency of a multi-level branch predictor. Any branch predictor could take advantage of the multi-level approach, especially since future process technologies may favor smaller, faster structures [2, 60]. To fully take advantage of the FTQ, we want a branch predictor that is also capable of maintaining a high degree of FTQ occupancy. This can be achieved either through large predictions (as in a trace cache) or through making multiple predictions per cycle (as in a 2-block ahead predictor). We choose to investigate the former, using a multi-level branch predictor hierarchy called the Fetch Target Buffer (FTB) [70].

IX.A Fetch Target Buffer

Figure IX.2 shows the FTB design, which is an extension of the BBTB design by Yeh and Patt [94, 95] with three main changes to their design. The first change is that we store fetch blocks rather than basic blocks. As mentioned in Chapter VIII, fetch blocks may encapsulate one or more strongly biased not-taken



	Tag	Distance	Target
A(1) →	A(1)	E(n)	B(1)
B(1) →	B(1)	E(n)	B(1)
F(1) →	F(1)	G(n)	H(1)

Figure IX.1: The difference between the BBTB and FTB.
On the left we see a sample flow of control through a program. The circles represent basic blocks and the lines are control transitions between blocks. Blocks C and D are the taken paths of the branches at the end of blocks A and B respectively. The darkened lines show the path that is followed – basic blocks C and D are never executed. The center diagram shows the corresponding fetch streams from the BBTB and FTB predictors. The BBTB predicts a single basic block at a time. The FTB predicts three different *fetch blocks*: one starting with basic block A and ending with basic block E. Another starts with basic block B and ends with basic block E. The last is solely composed of fetch block F. The rightmost diagram shows the contents of a simplified FTB. Shown are entries for the tag, the target, and the fetch distance. Here, the basic blocks are annotated with either a (1) or an (n). The (1) corresponds to the first instruction of the basic block and the (n) corresponds to the last instruction of a basic block. For example, the FTB entry indexed by B(1) is the only entry predicted and used while executing the entire loop, which represents BE. The entry is tagged by the address of the first instruction in basic block B. The fetch distance extends to the final instruction in basic block E, where the branch would be located. The target stores the address of the first instruction in basic block B.

branches, and so may contain multiple basic blocks. Therefore, an FTB entry is a form of a sequential trace, storing trace predictions with embedded fall-through branches. Figure IX.1 further illustrates the difference between fetch blocks and basic blocks. The first column of the Figure shows a sample flow of control through a simple program fragment. The circles A–F represent basic blocks in the program. The fragment begins at block A and exits at block F. Blocks A, B, and E end in conditional branches. The branches at the end of blocks A and B have never been taken over the execution of the program so far, and blocks C and D have never been executed. The third column of the Figure gives the state of an FTB for the program fragment. Here, there are three main fetch blocks: the fetch block containing basic blocks A, B, and E; the fetch block containing basic blocks B and E; and the fetch block containing basic blocks F, G, and H. (G and H are not shown and follow basic block F). The second column in the Figure provides sample fetch streams for both a BBTB and an FTB. The FTB is able to predict blocks B and E together as a single fetch block prediction. The BBTB must make two predictions to provide the same bandwidth.

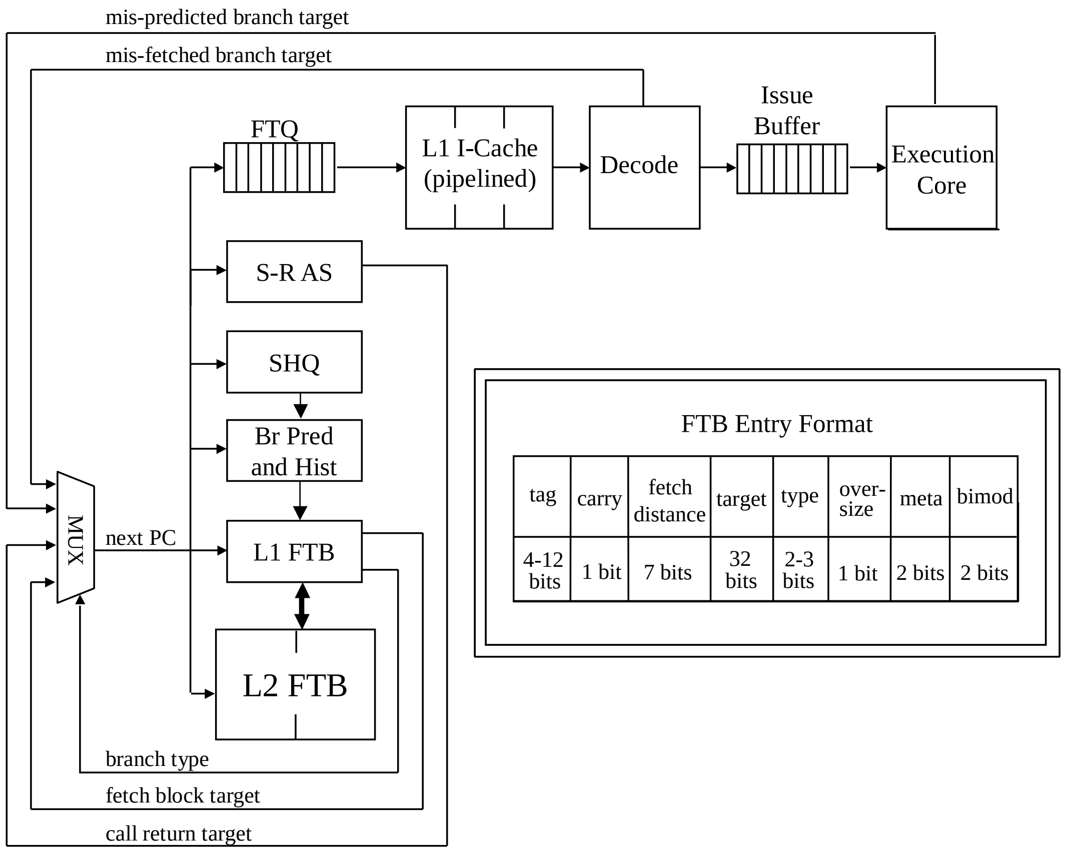
The second change is that we do not store fetch blocks in our fetch target buffer that are fall-through fetch blocks, whereas the BBTB design stores an entry for *all* basic blocks, wasting some BBTB entries. When the FTB misses, a default fetch block size is used to generate the next target fetch address.

The third change we made to the BBTB design is that we do not store the full fall-through address in our FTB. The fall-through address is instead calculated using the *fetch distance* field in the FTB and the carry bit. The fetch distance really points to the instruction after the potential branch ending the FTB entry (fetch block). We store only the pre-computed lower bits of the fall-through address in the fetch distance along with a carry bit used to calculate the rest of the fall-through address in parallel with the FTB lookup [13]. This helps reduce the amount of storage for each FTB entry, since the typical distance between the current fetch address and the fetch block's fall-through address is not large.

In addition to these changes, we explore a multilevel branch prediction hierarchy. As discussed in Chapter III, as cycle times continue to shrink, smaller speculative structures and predictors become more and more attractive. Much as the memory subsystem became divided into a hierarchy of predictors ranging from small and fast first level predictors to larger and slower second and third level structures, we explore dividing the branch prediction architecture into a scalable hierarchy. The first level FTB is small enough to provide a fast cycle time and produce a prediction each cycle. The second level is large enough to provide the capacity for accurate predictions, and is pipelined in accordance with the cycle time of the first level FTB (this assumes that the FTB sets the cycle time of the processor). The FTB we explore in this thesis has 2 levels, but could be composed of any number of levels, much like a cache hierarchy. For the FTBs we consider, the L2 FTB need only be pipelined into two stages.

IX.A.1 FTB Structure

Our Fetch Target Buffer (FTB) design is shown in Figure IX.2. The FTB is accessed with the start address of a fetch target block. Each entry in the FTB contains a tag, taken address, fetch distance, fall-through carry bit, branch type, oversize bit, and conditional branch prediction information. The FTB entry represents the *start* of a fetch block. The fetch distance represents the precomputed lower bits of the address for the instruction following the branch that ends the fetch block. The goal is for fetch blocks to end only with branches that have been taken during execution. If the FTB entry is predicted as taken, the taken address is used as the next cycle's prediction address. Otherwise, the



FTB Entry Format

tag	carry	fetch distance	target	type	over-size	meta	bimod
4-12 bits	1 bit	7 bits	32 bits	2-3 bits	1 bit	2 bits	2 bits

Figure IX.2: The decoupled front-end architecture with fetch target buffer. This figure elaborates upon the high-level pipeline from Chapter I. The branch prediction unit, which feeds into the FTQ, is composed of a two-level fetch target buffer (*FTB*), a gshare predictor with global history (*Br Pred and Hist*), a speculative history queue (*SHQ*), and a speculative return address stack (*S-RAS*). The fetch target buffer predicts fetch addresses using prediction data from the gshare predictor and global history. The various fields of the FTB are shown in the diagram, and will be explained in detail in this section. Since the FTQ enables the FTB to predict far ahead of the current fetch stream, we use the SHQ and S-RAS to track and recover speculative state (from Chapter VIII. Fetch addresses are stored in the FTQ, where they are then consumed by the instruction cache and decode hardware. Instructions are then supplied to the issue buffer, where they are executed and retired by the execution core.

fall-through address (fetch distance and carry bit) is used as the next cycle's prediction address.

The FTB design allows never taken branches to be encapsulated within a fetch block. But, if a formerly never taken branch is taken, the FTB simply decreases the fetch distance of the fetch block that encompassed the formerly never taken branch. The new fetch distance stops at the newly taken branch.

As described earlier, the fall-through address is not stored in its entirety in the FTB entry. It is computed from the address of the branch that ends the fetch block, the fetch block end address. Only N low order bits of the fetch block end address are stored along with a carry bit. If the carry bit is not set, the fetch block end address is calculated by concatenating the upper $address\_size - N$ bits of the current fetch address with the N fetch distance bits stored in the FTB entry. If the carry bit is set, the fetch block end address is calculated by adding one to the upper $address\_size - N$ bits of the current fetch address, and then concatenating this with the N fetch distance bits stored in the FTB entry. The calculation of adding the carry bit to the upper bits of the PC is done in parallel with the FTB lookup. The fall-through address is then calculated by adding 4 to the fetch block end address. If the branch is predicted as not-taken, the carry bit chooses between the two possible values for the upper bits of the fall-through address, and then performs the concatenation.

The size of the N bit fetch distance field determines the size of the sequential fetch blocks that can be represented in the fetch target buffer. If the fetch distance is farther than 2^N instructions away from the start address of the fetch block, the fetch block is broken into chunks of size 2^N, and only the last chunk is inserted into the FTB. The other chunks will miss in the FTB and be predicted not-taken, incrementing the current fetch PC by 2^N, which is the max fetch distance. Eventually, the PC corresponding to the final chunk will be encountered, which will hit in the FTB and provide a branch prediction. Smaller sizes of N mean that fetch blocks will be smaller - thereby increasing the number of predictions that must be made and potentially decreasing the FTQ occupancy. Larger sizes of N will mean less predictions and potentially higher FTQ occupancy, but will also mean that FTB misses will result in large miss fetch blocks which can potentially pollute the instruction cache.

An oversize bit is used to represent whether or not a fetch block spans a cache block [94]. This is used by the instruction cache to determine how many predictions to consume from the FTQ in a given cycle. We simulated our results with a single instruction cache port in this Chapter, but will examine a dual ported cache in Chapter X. The oversize bit is used to distinguish whether a prediction is contained within one cache block or if its fetch size spans two or more cache blocks. If the oversize bit is set, the predicted fetch block will span two or more cache blocks, and a dual ported cache could use its two ports to fetch the first two sequential cache blocks. If the bit is not set, the prediction only requires a single cache block, so the second port could be used to start fetching the target address of the next FTQ entry.

IX.A.2 Branch Direction Predictor

The branch direction predictor shown in the FTB in Figure IX.2 is a hybrid predictor composed of a meta-predictor that can select between a global history predictor and a bimodal predictor. Other combinations are certainly possible, as well as non-hybrid predictors. The global history is XORed with the fetch block address and used as an index into a global pattern history table. The meta-prediction is used to select between the various predictions available, depending on the specifics of the design. The meta-predictor is typically implemented as a counter to select between two predictions or as a per-predictor confidence mechanism to select amongst three or more predictors [71]. The final prediction result is used to select either the target address of the branch at the end of the fetch block or the fetch block fall-through address.

The meta predictor and bimodal predictor values are not updated speculatively, since they are state machines and not history registers. The front-end can only assume it made the correct prediction and thus reinforce bimodal predictions. It has been shown in [42] that better performance results when such predictor updates are delayed until the result of the branch outcome is known, (*i.e.* at execution or retirement).

In the decode stage, the predicted direction of unconditional branches, *e.g.*, jumps, calls and returns, and the targets of direct branches, *e.g.*, PC relative and absolute, are validated. In the writeback stage, the targets of indirect branches and the direction of conditional branches are validated. Fetch block targets and sizes are propagated down the pipeline with instructions. During validation, if a branch target does not match the accompanying fetch block, a branch misprediction recovery sequence is initiated. The FTB entry is updated with the correct fetch block information, mispeculated entries in the speculative history queue are released, and the pipeline is flushed behind the mispeculated branch. The prediction history of branches is also updated at this point. To facilitate the embedding of strongly biased not-taken branches within fetch blocks, not taken branches do not update history or create FTB entries unless they are already contained in the FTB and at the tail of a fetch block. In addition, new FTB entries are only allocated when branches are taken.

In our simulations we do not make use of a two level per-branch predictor. However, if such a predictor were to be used, it need not lengthen the access time of the FTB. On an L1 FTB hit, the prediction from the bimodal predictor could be speculatively used to select the prediction target for the following cycle.

This is the case even if the meta-predictor indicates that the two level per-branch predictor result should be used. This speculative access permits the two level per-branch predictor history table access to be overlapped with the L1 FTB access in the following cycle. By overlapping pattern history table and FTB accesses, we can cycle the L1 FTB at the rate of the L1 FTB table alone, rather than the rate determined by the latency of the FTB table plus the pattern history table. If the meta-predictor indicates that the two level per-branch predictor should be used and the prediction result returned in the following cycle matches the bimodal prediction, the prediction result will be correct and the per-branch pattern history table access will have been successfully overlapped with the following FTB access cycle. Many branches are strongly biased and thus will be predicted by the single-cycle bimodal predictor. Of the remaining branches, many of the bimodal and per-branch predictions will match, again permitting compete overlap. Only in the case where the meta-predictor indicates that per-branch predictor should be used and bimodal prediction does not match the per-branch predictor, does the overlap not occur. In this event, the results of the speculative L1 FTB access are thrown away and the FTB is restarted with the correct address. This misprediction results in the loss of a single FTB cycle. This is similar to what is done in the Alpha 21264 [44].

IX.A.3 Functionality of the 2-Level FTB

The FTQ enables the use of a multi-level branch predictor, since the latency of the predictions from the L2 and higher predictors can be masked by the high occupancy of the FTQ. We will now describe the functionality of a 2-level FTB design.

The L1 FTB is accessed each cycle using the predicted fetch block target of the previous cycle. At the same time the speculative return address stack (S-

RAS) and the global history prediction table are accessed. If there is an L1 FTB hit, then the fetch block address, the oversize bit, the last address of the fetch block, and the target address of the fetch block are inserted into the next free FTQ entry.

If the L1 FTB misses, the L2 FTB needs to be probed for the referenced FTB entry. To speed this operation, the L2 FTB access begins in parallel with the L1 FTB access. If at the end of the L1 FTB access cycle a hit is detected, the L2 FTB access is ignored. If an L1 miss is detected, the L2 FTB information will return in $T - 1$ cycles, where T is the access latency of the L2 FTB (in L1 FTB access cycles). On an L1 FTB miss, the predictor has the target fetch block address, but doesn't know the size of the fetch block. To make use of the target address, the predictor injects fall-through fetch blocks starting at the miss fetch block address into the FTQ with a predetermined fixed length. Once the L2 FTB entry is returned, it is compared to the speculatively generated fetch blocks: if it is larger, another fetch block is generated and injected into the FTQ. If it is smaller, the L1 FTB initiates a pipeline squash at the end of the fetch block. If the fetch target has not made it out of the FTQ, then no penalty occurs. If the fetch target was being looked up in the instruction cache, those instructions are just ignored when the lookup finishes. The final step is to remove the LRU entry from the corresponding L1 FTB set, and insert the entry that was found in the L2 FTB. The entry removed from the L1 FTB, is then inserted into the L2 FTB also using LRU replacement.

If the L2 FTB indicates the requested FTB entry is not in the L2 FTB, the L1 FTB enters a state where it continually injects sequential fetch blocks into the machine until a misfetch or misprediction is detected in the decode or writeback stage of the processor. Once a misfetch or misprediction is detected, the L1 FTB will be updated with the correct information regarding this new fetch block, and then the L1 FTB will once again begin normal operation.

IX.B Results

The results shown use the same base architecture described in Chapter VI. In this Chapter, a single ported 16K 2-way set associative instruction cache is used. We make use of a 32 entry FTQ (except where otherwise noted) and use 7 bits for the fetch distance. The SHQ contains 32 entries. FTB cycle times are as indicated in Chapter VI except for Figures IX.9 and IX.10 (with the exception of these two Figures, we assume ideal interconnect scaling). For IPS results, we assume that the cycle time of the processor is set by the access time to the L1 FTB, and appropriately pipeline the remaining structures of the pipeline (L2 FTB, instruction cache, data cache, etc).

IX.B.1 Predictor Results with Ideal Interconnect Scaling

We first will motivate the potential benefit from pipelining the instruction cache for future processors. Figure IX.3 shows the Billion Instructions per Second (BIPS) results for two FTB configurations with pipelined (2 cycle) instruction caches, and an FTB configuration with a single cycle instruction cache. The FTB designs with pipelined instruction caches use a cycle time equal to the FTB access time, and the non-pipelined I-cache uses a cycle time equal to the instruction cache access time as shown in Table VI.2. The FTB with the single cycle instruction cache avoids lengthening the branch misprediction penalty by a cycle (pipelining the instruction cache in these experiments extends the pipeline by one stage). However, as can be seen, the increase in cycle time has more of an impact on BIPS. In fact, the benefit from this technique could even be greater with a larger instruction cache, as the access time to a larger cache could also fit within two cycles. The larger cache would provide a higher IPC for all configu-

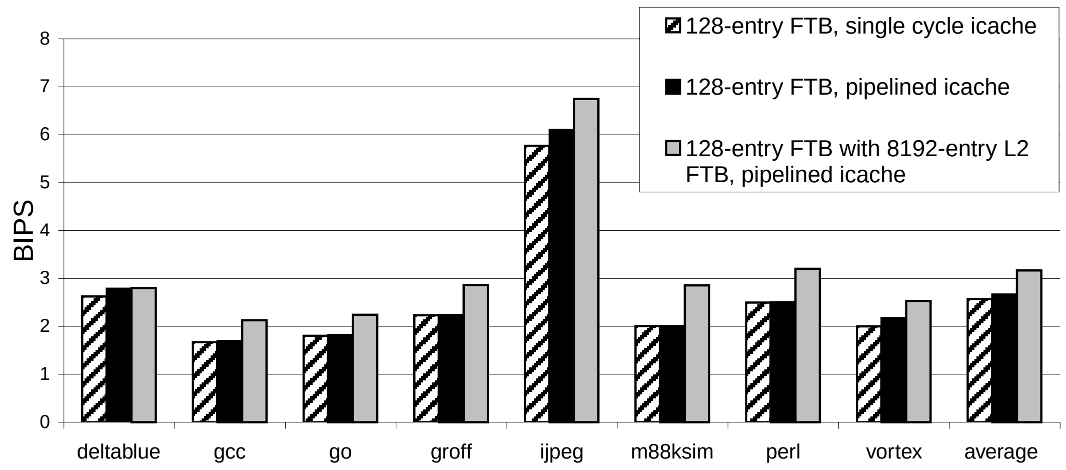


Figure IX.3: BIPS comparison across three FTB/icache configurations. Billion Instructions per Second (BIPS) results were calculated using IPC values from SimpleScalar simulations and CACTI timing data. A 16KB 2-way set-associative instruction cache is used here. The first bar for each benchmark represents an FTB with 128 entries that does not have a pipelined instruction cache. In this case, the cycle time is set to the cycle time of the instruction cache - 0.67ns. The second bar for each benchmark represents a single level FTB configuration with 128 entries. This configuration has a 2 cycle pipelined instruction cache, with a cycle time of - 0.59ns. The third bar for each benchmark represents a single level FTB configuration with 128 entries and a second level with 8192 entries. This configuration also has a 2 cycle pipelined instruction cache, with a cycle time of - 0.59ns. Table VI.2 summarizes the timing data for this Chapter.

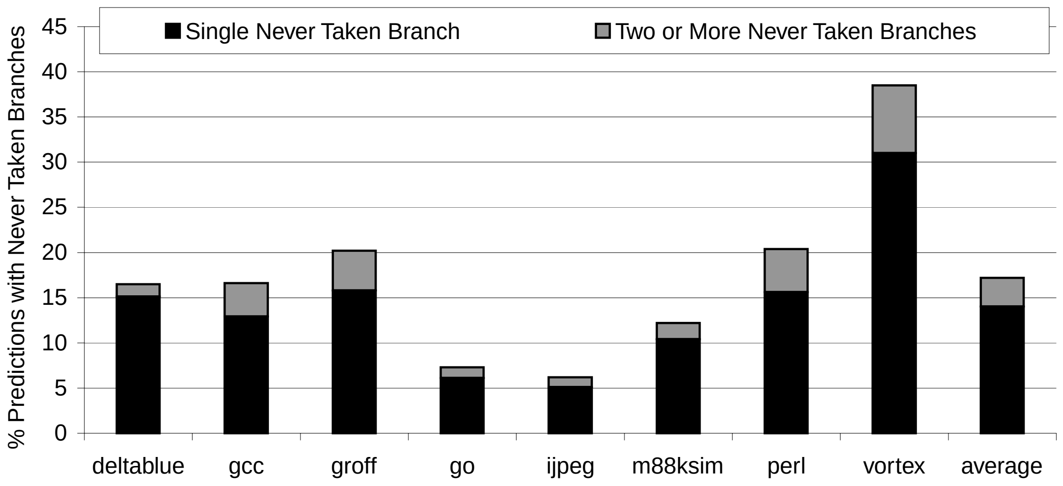


Figure IX.4: Percent of predictions from the FTB that span multiple basic blocks. The x-axis shows the benchmarks we examined, and the y-axis shows the percent of predictions that contain one or more never taken branches. The black bar shows the percent of predictions that contain a single never taken branch and the grey bar shows the percent of predictions that contain two or more never taken branches.

rations, but would decrease the BIPS of the non-pipelined configuration due to an increase in cycle time.

Next, we examine the ability of the FTB to encapsulate never taken branches. Figure IX.4 shows that on average, 14% of predictions include two basic blocks (i.e. include a single never taken branch) and an additional 3.2% of predictions include more than two basic blocks (i.e. include two or more never taken branches). Predictions in `vortex` span multiple basic blocks nearly 40% of the time. Michaud et al. [58] also examined predictors that are capable of bypassing a single not taken branch and found an average fetch rate of 1.3 basic blocks per cycle for an ordinary BTB.

Figure IX.5 shows results in instructions per cycle (IPC) for the benchmarks we examined and an average. For most benchmarks, the increase in predictor size results in increased performance. Benchmarks like `perl` and `m88ksim` are more affected by FTB size due to a large number of branches. For most,

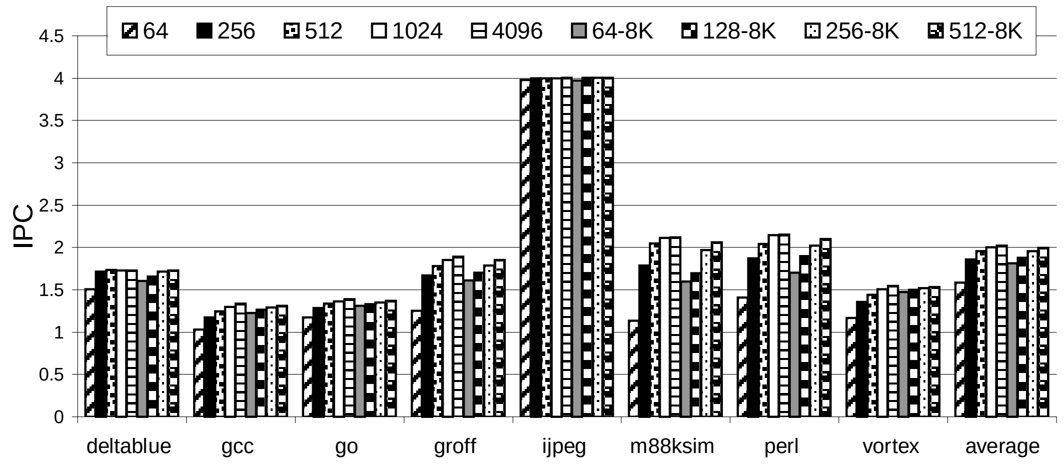


Figure IX.5: IPC comparison across a variety of FTB configurations. The first five bars represent single level FTB configurations: 64, 256, 512, 1024, and 4096 entry first level FTBs. The next four bars represent dual level FTB configurations: 64, 128, 256, and 512 entry first level FTBs, each with an 8192 entry second level FTB.

the difference between a 1024 and a 4096 entry FTB is minimal. The results for `ijpeg` show that this benchmark does not contain a significant amount of branch state. It has few taken branches encountered (only 4.7% of instructions are branches at all) and therefore requires very little space in the FTB. This, in addition to its low instruction cache miss rate, helps to explain the relatively high IPC obtained with this benchmark. The two level results demonstrate that the second level FTB is improving the prediction accuracy of the first level FTB.

Figure IX.6 shows results in Billion Instructions per Second (BIPS) for single level (64 entry to 4096 entry) and two level FTB designs (64 to 512 entry first level table with an 8192 entry 2nd level table). For most benchmarks, the 512 entry FTB is the best single level performer, with a cycle time of 0.64ns. The exception is the performance of `ijpeg` as it does not contain a significant amount of branch state. The second level results are very closely clustered on average. The best performer on average was the configuration with a 128 entry first level FTB and 8192 entry second level FTB. But this varied from benchmark

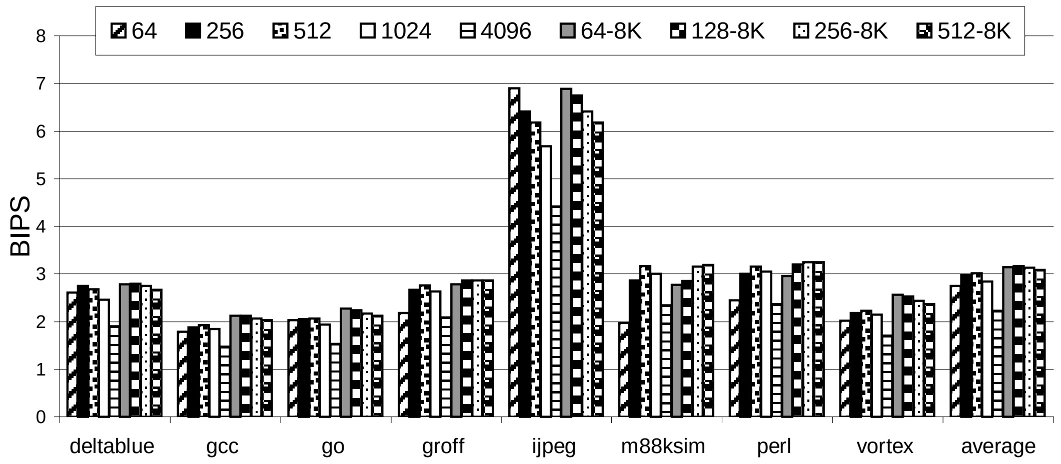


Figure IX.6: BIPS comparison across a variety of FTB configurations. BIPS results were calculated using IPC values from SimpleScalar simulations and CACTI timing data. The bars for each benchmark represent different FTB configurations. The first five bars represent single level FTB configurations: 64, 256, 512, 1024, and 4096 entry first level FTBs. The next four bars represent dual level FTB configurations: 64, 128, 256, and 512 entry first level FTBs, each with an 8192 entry second level FTB.

to benchmark. The benchmark `m88ksim` has a relatively low FTQ occupancy (due to a small average fetch distance), and therefore is not able to tolerate the latency to the second level FTB as well as other benchmarks. These results show that the two level FTB performs slightly better on average than a single level design in the absence of the interconnect scaling bottleneck (i.e. assuming ideal technology scaling in the CACTI timing model).

These results show that IPC does not provide the full picture of processor performance. The BIPS results in Figure IX.6 show that if the FTB access time determines the cycle time of the processor, then a 512 entry FTB provides the best average performance of the single level FTB designs and the 128 entry first level and 8192 entry second level FTB provides the best average performance of the 2-level FTB designs. If one were only to look at IPC, it would appear that the bigger the table is, the better the performance looks, even though the access time for a given cycle would not be realistic.

To measure the accuracy of the FTB, it is necessary to consider both the fetch distance and the branch prediction at the end of the fetch block. The fall-through fetch distance must not encompass a taken branch, and the branch prediction must be correct. Therefore, we define an accurate FTB prediction to be one which satisfies both of these criteria. Figure IX.7 shows the percent of accurate predictions for the benchmarks we examined and an average. Note that this prediction accuracy is in terms of fetch blocks which can span multiple biased not-taken branches to provide a wide fetch block. Therefore, the accuracy will be lower than just looking at conditional branch prediction ratios. This data correlates with the IPC results we measured. Benchmarks like `gcc` and `vortex` show significant differences in accuracy across varying FTB sizes. `Vortex` achieves highly accurate prediction - the two level FTB configurations are all over 90% accurate. On average, we see that the two level FTB configuration provides a boost to accuracy. The 256 entry first level has a 13% accuracy improvement with a second level FTB.

Table IX.1 shows prediction accuracy data on the best single and dual level FTB configurations (from the BIPS data). On average, the results show that 11.2% of predictions from the 2nd level FTB are correctly predicted in the two-level configuration. The benchmark `ijpeg` again proves interesting as 31% of predictions in the single level FTB configuration are correctly predicted and come from misses in the FTB (which would use the default FTB fetch distance and a fall-through prediction). This is due to the nature of the benchmark which contains very few taken branches. Table IX.1 shows that the average FTB prediction size for the single level FTB is around 8 instructions, and 5 predictions occur in a row before reaching a misprediction. This means that on average, a single level FTB can supply around 43 instructions between mispredictions. The

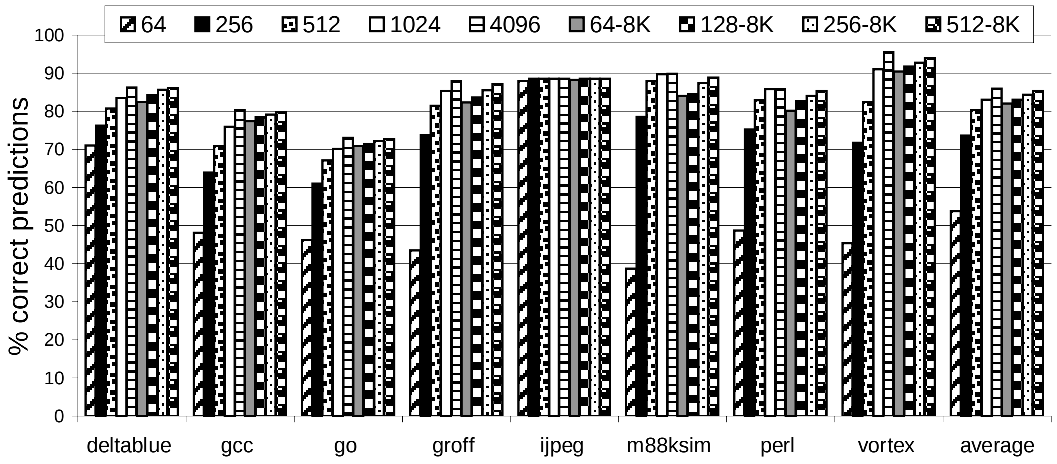


Figure IX.7: Accuracy comparison across a variety of FTB configurations. We measured the accuracy of the FTB by tracking the number of predictions which had both a valid fetch distance and a correct branch prediction. A fetch distance is considered valid if the resulting fetch address stream does not contain any taken branches. The correct branch prediction refers to the prediction of the branch at the end of the fetch block. The first five bars represent single level FTB configurations: 64, 256, 512, 1024, and 4096 entry first level FTBs. The next four bars represent dual level FTB configurations: 64, 128, 256, and 512 entry first level FTBs, each with an 8192 entry second level FTB. This Figure displays the same measure as the 2nd and 6th columns in Table IX.1, but for a wider range of FTB configurations.

Table IX.1: FTB prediction accuracy

	512				128-8K					
program	% FTB acc	% from miss	ave pred size	preds in a row	% FTB acc	% from L1	% from L2	% from miss	ave pred size	preds in a row
deltablue	80.7	0.1	6.1	5.1	84.5	77.2	4.2	3.1	5.8	6.3
gcc	70.9	0.4	7.1	2.9	79.2	59.5	13.2	6.5	6.1	4.5
groff	81.4	0.4	7.3	4.8	84.2	60.2	18.0	6.0	6.4	6.0
go	67.1	1.5	7.8	2.7	72.4	56.1	8.2	8.1	6.8	3.5
ijpeg	88.5	31.0	15.4	8.6	88.6	39.4	0.0	49.2	11.4	8.5
m88ksim	88.0	0.2	4.9	8.0	83.9	60.8	19.0	4.2	4.8	6.0
perl	82.9	0.2	6.6	5.5	83.6	65.9	13.9	3.8	6.3	6.0
vortex	82.5	2.4	10.6	4.8	91.9	60.1	13.1	18.7	8.4	12.1
average	80.3	4.5	8.2	5.3	83.5	59.9	11.2	12.5	7.0	6.6

This table examines two FTB configurations: a single level FTB with 512 entries, and a 128 entry FTB with a second level with 8192 entries. For both configurations, we list the percent of FTB predictions with valid fetch distances and correct branch predictions, shown in columns 2 and 6. Columns 3 and 9 show the percent of correct predictions that resulted from an FTB miss. In this case, the default fetch distance is predicted. In addition, for the two level configuration, we show the percent of correct predictions that were from the first level FTB (column 7) and the percent of correct predictions that were from the second level FTB (column 8). For both configurations, we show the average number of instructions in each fetch block that are predicted in a single cycle (columns 4 and 10). Also, we show the number of predictions in a row that were produced on average before a misprediction (columns 5 and 11). The product of these two numbers provides the average number of instructions between mispredictions.

2-level FTB is able to supply slightly more instructions between mispredictions - around 46 instructions on average. The exception is `m88ksim`, again due to the frequency of taken branches in this benchmark. Without sufficient FTQ occupancy, `m88ksim` is unable to tolerate second level FTB accesses as well as other benchmarks, and a single level configuration is able to perform as well as the two-level configurations.

Figure IX.8 shows the performance of a 2-level FTB with and without an FTQ, and demonstrates how the FTQ enables the 2-level FTB. Using a 32 entry FTQ provides a 16% improvement on average in IPC over a design without an FTQ. Results shown are for a 128 entry first level FTB with an 8192 entry

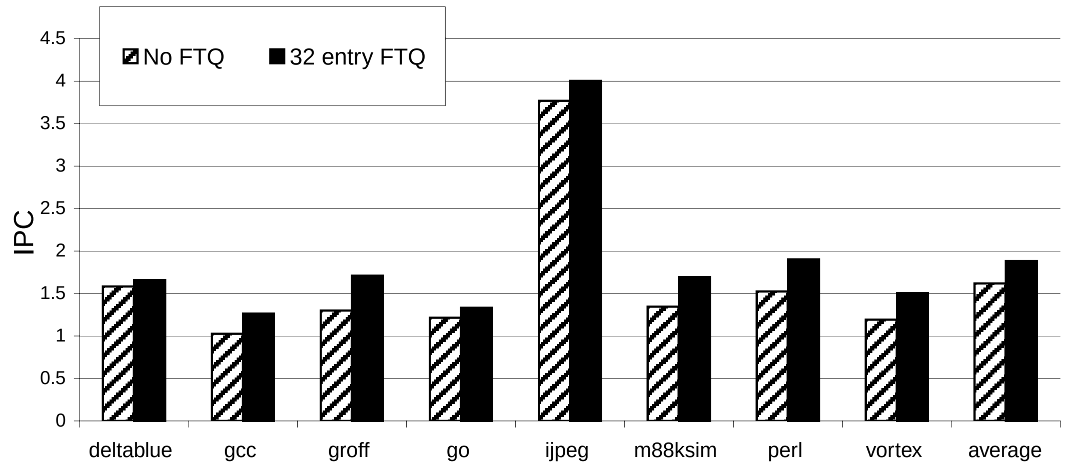


Figure IX.8: IPC comparison with and without FTQ.
This figure shows IPC results for a two-level FTB (128-entry first level, 8192-entry second level) without an FTQ and with a 32-entry FTQ. The x-axis shows the benchmarks we examined and an average. This is similar to Figure VIII.6, but shows results for a two-level FTB.

second level FTB. Without an FTQ, there is limited opportunity for the second level FTB to correct a miss in the first level FTB. The occupancy in the FTQ helps tolerate the latency of the second level FTB access.

IX.B.2 Predictor Results Assuming the Interconnect Scaling Bottleneck

As described in Chapter III, interconnect is expected to scale poorly due to the impact of resistative parasitics and parasitic capacitance. In that Chapter, we concluded that memory structures would be seriously impacted by poor interconnect scaling, and that large memories would scale worse than small memories. In this section, we use a modified version of CACTI 2.0, based on [4]. Table IX.2 summarizes the FTB timing data we used for four process technology sizes: $0.80\mu m$,$0.35\mu m$, $0.18\mu m$, and $0.10\mu m$.

Figure IX.9 also shows results in Billion Instructions per Second (BIPS), but uses the technology scaling calculations from [8] (Table IX.2) showing the

Table IX.2: Timing data for Figure IX.9

FTB num	Access Time (ns)			
entries	$0.80\mu m$	$0.35\mu m$	$0.18\mu m$	$0.10\mu m$
64	4.48	1.75	0.98	0.76
128	4.65	1.83	1.05	0.85
256	4.87	1.98	1.17	0.99
512	5.02	2.10	1.32	1.17
1k	5.40	2.42	1.60	1.50
4k	7.35	3.71	2.60	2.70

This data is only used for the BIPS results in Figure IX.9 and Figure IX.10. It was taken from a modified version of the CACTI 2.0 timing tool using scaling data from [4]. Four process technology sizes are considered.

potential effects of the interconnect scaling bottleneck. In this Figure, we examine data for the $0.10\mu m$ process technology size. These results show an even more substantial difference in BIPS between single and two level FTB designs. When taking into consideration the interconnect scaling bottleneck, we found that the best performing two level FTB design provided a 14% improvement in BIPS over the best performing single level design.

Figure IX.10 shows average results in BIPS for the benchmarks we examined across four different process technology sizes. At the $0.80\mu m$ technology size, all configurations are fairly close in terms of BIPS. But, the performance of the 4096-entry FTB and 1024-entry do not scale with the other configurations. In fact, the 4096-entry FTB actually achieves a lower BIPS at the $0.10\mu m$ technology size than it had at the $0.18\mu m$ technology size. This is due to poor interconnect scaling. The best performing configurations are the two-level predictors, which are able to combine good IPC with small cycle times.

For the remainder of this thesis, we will assume ideal interconnect scaling with the timing parameters from Chapter VI.

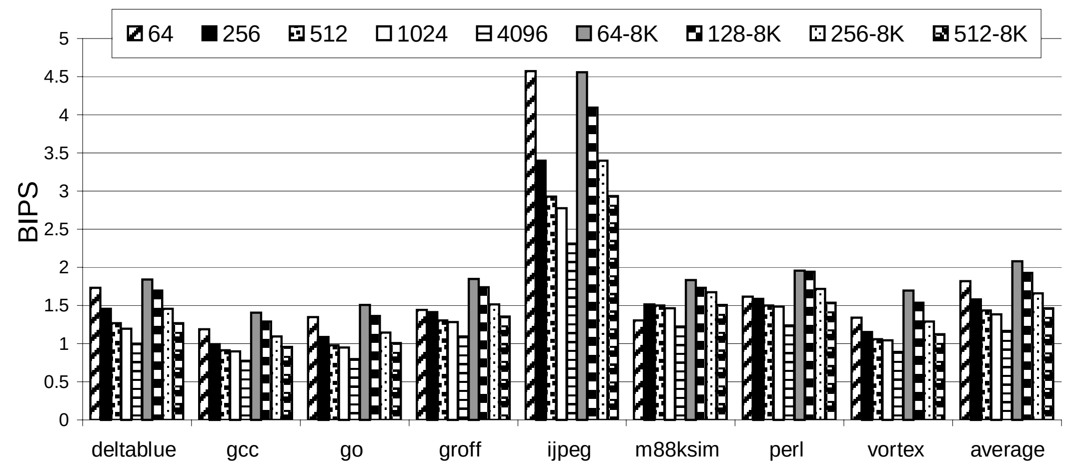


Figure IX.9: BIPS comparison assuming non-ideal interconnect scaling. Fetch Target Buffer performance for $0.10\mu m$ feature size using technology scaling calculations from [8] modeling the potential effect of the interconnect scaling bottleneck.

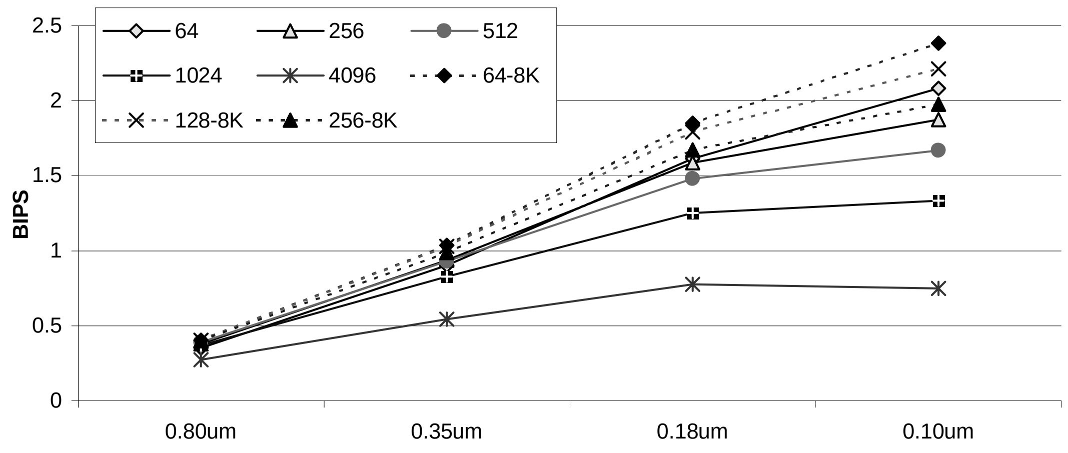


Figure IX.10: BIPS comparison assuming non-ideal interconnect scaling. Fetch Target Buffer performance across four process technology sizes using technology scaling calculations from Table IX.2. The y-axis shows BIPS results. The x-axis ranges across the four process technology sizes. The dotted lines represent two-level FTB configurations. The solid lines represent single-level FTB configurations. Configurations with the same cycle times are shown with the same shaped bullets (*i.e.* the 64-entry FTB and the 64-entry FTB with second level FTB).

IX.B.3 Fetch Distance

Figure IX.11 provides results for the different fetch distance sizes needed for the programs examined. The histogram presents 8 disjoint categories that describe the size of fetch address predictions from the FTB. The categories refer to the number of bits required to exactly capture the fetch distance of a given prediction. This data was generated using a 7-bit maximum fetch distance. This was sufficient to capture most predictions, with the exception of `ijpeg` which still may have had predictions that could be captured with more than 7 bits. As can be seen, 18% of all predictions on average could be captured with only three bits. 26% required exactly five bits. Having 6 bits captures 88% of all fetch distances on average. It is also interesting to note that some programs tend toward larger fetch distances than others, such as the large distances seen with `vortex`. Larger fetch distances will provide more FTQ occupancy and, coupled with high prediction accuracy, will provide more opportunities for FTQ-based optimizations.

IX.B.4 FTQ Occupancy

Figure IX.12 shows two occupancy histograms: (a) corresponds to a large, single level branch predictor (taken from Chapter VIII) and (b) corresponds to a 32 entry L1 FTB with a 512 entry L2 FTB. FTQ occupancy is important in order to provide latency tolerance for multilevel prediction structures and to provide a lookahead into the future fetch stream. As seen in the Figure, the single level configuration has a higher average occupancy than the two level configuration, since the two level configuration must tolerate a second level access from time to time. While the FTQ of the single level configuration is full nearly 24% of the time, the FTQ of the two level configuration is full around 21% of the time. Even with the added latency of the L2 FTB, the two level configuration is

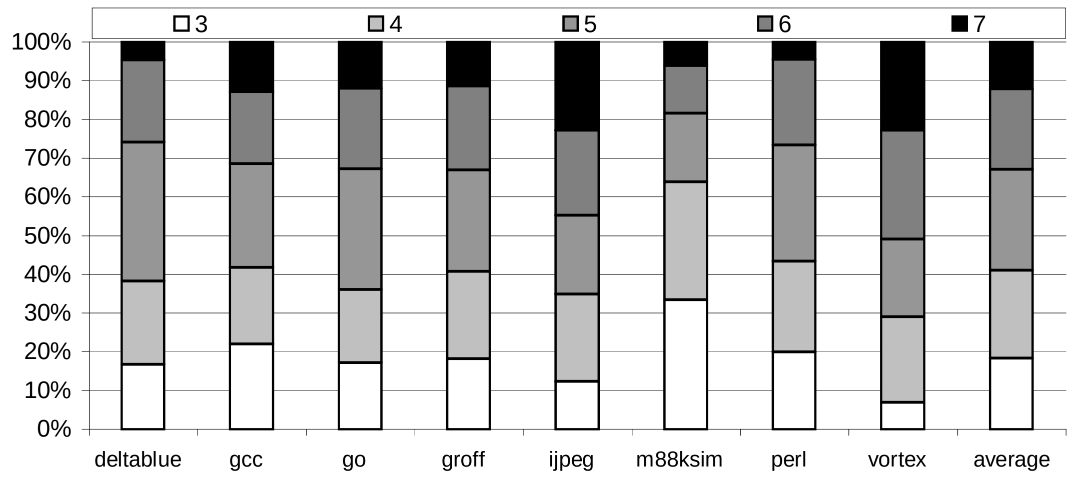
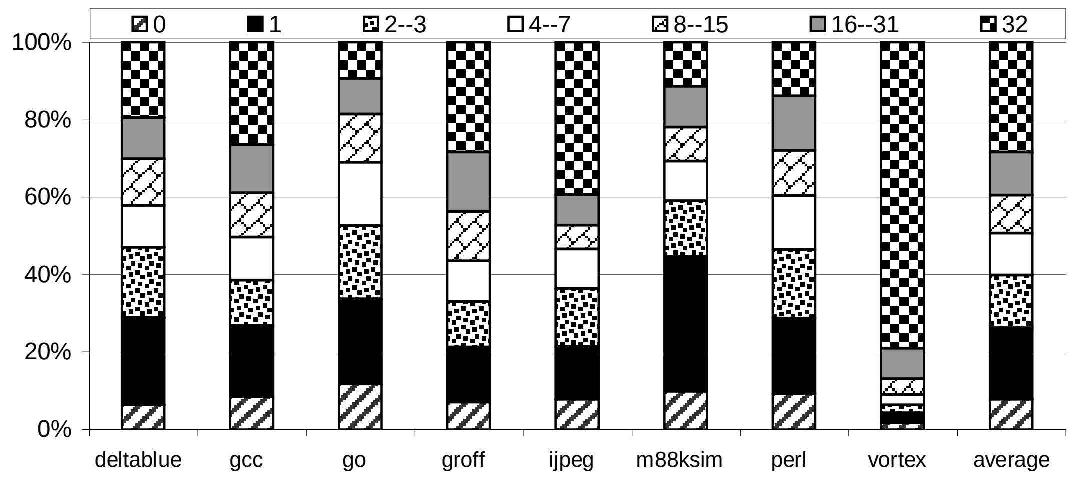
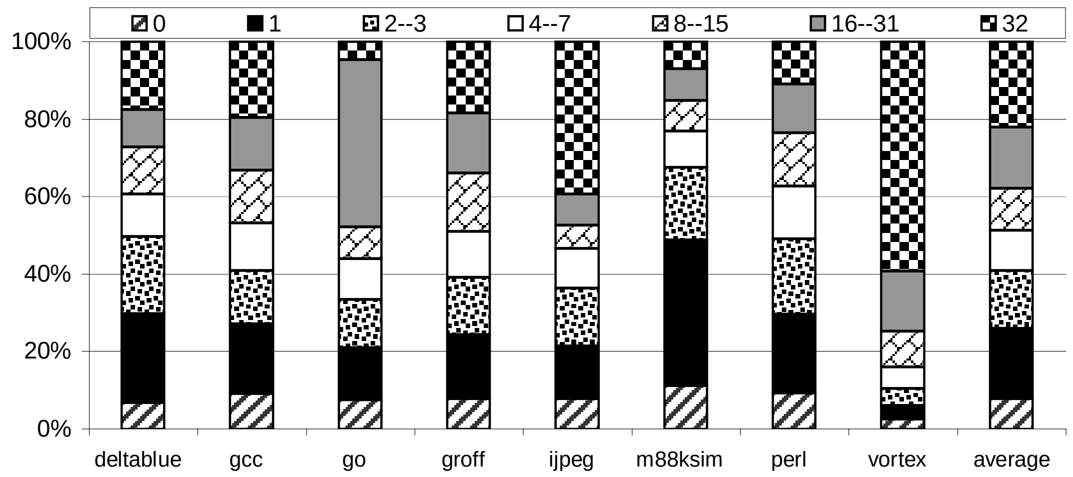


Figure IX.11: Fetch distance histogram.

The Y-axis shows the disjoint percent of predictions that had a fetch distance represented by 2 to 7 bits. The fetch distance is the size of the fetch block prediction generated by the FTB. This histogram represents the distribution of fetch distances from FTB predictions. Only predictions that resulted from FTB hits are included (i.e. no FTB miss default fetch distances). To gather this data, a 512 entry single level FTB was examined, using a 7-bit maximum fetch distance (a prediction could hold 128 instructions). The legend shows the various disjoint fetch distance categories. For example, the middle portion of the bar (3rd from the bottom) shows the percent of predictions that need five bits to capture the fetch distance, but that could not be fit into four bits as a single prediction.



(a)



(b)

Figure IX.12: FTQ occupancy histogram.

This graph shows the percent of time the 32 entry FTQ used in this study had a given number of predictions stored within it after instruction fetch. The disjoint categories of FTQ occupancy are shown in the legend at the top of the graph. For example, the white section represents the percent of time that the FTQ contained between 4 and 7 predictions. The black section represents the percent of time that the FTQ contained exactly one prediction. Graph (a) corresponds to a single level branch predictor, and graph (b) corresponds to a two level branch predictor.

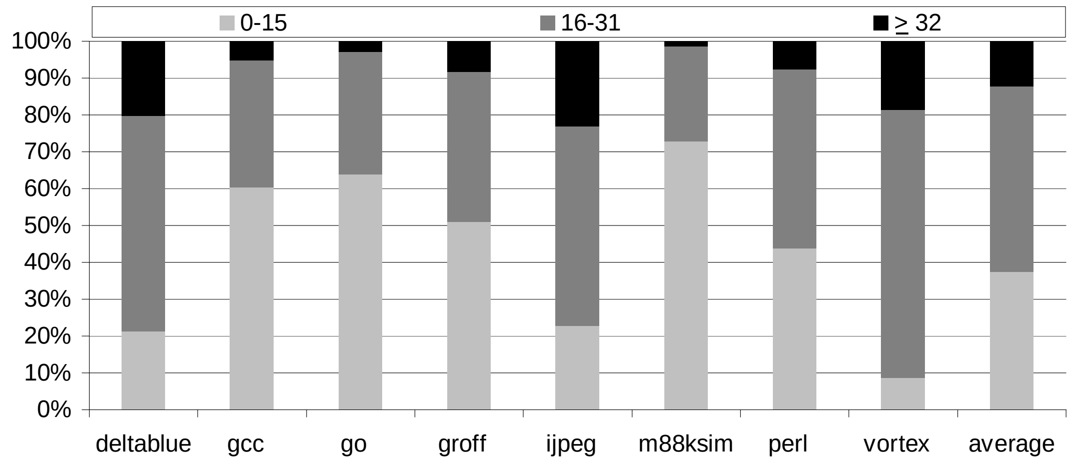


Figure IX.13: Speculative History Queue (SHQ) size.
The Y-axis shows the percent of time that the SHQ contained 0–15, 16–31, or 32 (or more) entries. The SHQ used in this figure has 64 entries. The lightest bar fragment shows the percent of time that the SHQ contained between 0 and 15 entries.

still able to achieve high levels of FTQ occupancy.

IX.B.5 Speculative History Queue Size

Figure IX.13 demonstrates the size of the SHQ for a multilevel branch prediction architecture. For this Figure, we used a 64 entry SHQ to determine what percent of time the SHQ held a given number of predictions. 89% of the time there were less than 32 entries in the SHQ, on average for all benchmarks. There were less than 16 entries in the SHQ around 40% of the time. Benchmarks like *vortex* which have very high FTQ occupancy (meaning that they are able to get many predictions ahead of the current fetch PC) require a larger number of SHQ entries than a benchmark like *m88ksim* which has a much lower FTQ occupancy.

IX.B.6 Further Enhancements

The FTB need not be fully tagged, and the *target* field need not be a complete address. Only enough address bits are required to create a unique L2 FTB index and tag. The L2 FTB need only be tagged if the L2 FTB is set-associative, and then only with enough fetch block address bits to reduce aliasing within sets. For a large L2 FTB, typically $log_2(A)$ bits should suffice, where A is the associativity of the L2 FTB.

IX.C Summary

In this Chapter we have shown how the FTQ enables the use of a two-level branch prediction hierarchy. Predictions enqueued in the FTQ keep the instruction fetch unit busy in the event that the first level FTB misses. This scalable design provides high prediction accuracy, while keeping access times low. Despite the second level, the FTQ still contains a significant amount of occupancy, which will allow the multi-level prediction architecture to work well with other forms of FTQ-based optimizations. This type of design could be further extended to make use of multiple branch predictors: smaller, faster predictors provide the majority of prediction bandwidth and larger, but slower predictors provide a means of verification for the smaller predictors.

Chapter X

Instruction Cache Performance Optimizations

The decoupled front-end design provides an opportunity for the branch prediction architecture to run ahead of the current instruction fetch PC. The stream of fetch addresses contained in the FTQ can be used to direct instruction cache prefetching. Contemporary instruction cache prefetching techniques follow a sequential prefetching path or use a separate predictor to guide prefetch. With the FTQ, we can use the branch prediction architecture, which has benefited from thorough research over the years, to guide our instruction cache prefetch – rather than relying on less intelligent sequential predictors, and less accurate and potentially expensive separate predictors. In this Chapter, we introduce *Fetch Directed Prefetching (FDP)* and compare it to some of the prior work. We also investigate different prefetch filtering mechanisms that provide improvement to both the prior work and FDP.

X.A Prior Instruction Cache Prefetching Work

In this section we describe related work to instruction cache prefetching not covered in Chapter IV. We will compare the performance of two of these (next line prefetching and stream buffers) to FDP in Chapter X.C.7.

X.A.1 Tagged Next Line Prefetching

Smith [81] proposed tagging each cache block with a bit indicating when the next block should be prefetched. When a block is prefetched its tag bit is set to zero. When the block is accessed during a fetch and the bit is zero, a prefetch of the next sequential block is initiated and the bit is set to one. Smith and Hsu [83] studied the effects of tagged next line prefetching and the benefits seen based upon how much of the cache line is used before initiating the prefetch request.

X.A.2 Target and Wrong Path Prefetching

Smith and Hsu [83] also examined the benefits of combining next-line prefetching with target prefetching. For target prefetching, they used a table of branch target addresses, which was indexed in parallel with the instruction cache lookup. If there was a hit, then the target address was prefetched. For a given branch, they examined prefetching both the fall through and the target address.

Pierce and Mudge [66] examined what they called Wrong Path Prefetching, where they examine prefetching both paths of a branch. There are two major differences in their approach and the approach suggested by Smith and Hsu [83]. They only prefetch the taken target if the branch was not-taken, and they only do this after the taken address is calculated in the decode stage. The address is not derived from the branch target buffer. This has the advantage of being able to prefetch branch targets not in the BTB. Their results showed that target prefetching provided only a small improvement over next-line prefetching.

X.A.3 Stream Buffers

Jouppi proposed stream buffers to improve the performance of directed mapped caches [40]. If a cache miss occurs, sequential cache blocks, starting with the one that missed, are prefetched into a stream buffer, until the buffer is filled. A stream buffer is implemented as FIFO queue. The stream buffer is searched in parallel with the instruction cache when performing a lookup. He also examined using multiple stream buffers at the same time.

Palacharla and Kessler [61] improved on this design by adding a filtering predictor to the stream buffer. The filter only starts prefetching a stream if there are two sequential cache block misses in a row. This was shown to perform well for data prefetching. In addition, they examined using non-unit strides with the prefetch buffer.

Farkas et. al. [27], examined the performance of using a fully associative lookup on stream buffers. This was shown to be beneficial when using multiple stream buffers, so that each of the streams did not overlap, saving bus bandwidth.

X.A.4 Other Hardware Based Instruction Prefetching

Joseph and Grunwald [39] examined using a Markov predictor for data and instruction prefetching. We did not examine using the Markov predictor for instruction prefetching, because of the size of predictor and the fact that the small predictors examined in this thesis performed well for instruction cache misses.

Joseph and Grunwald [39] examined storing bits in their Markov predictor table to indicate whether the prefetch address was actually used after being prefetched. If not, it was not used for a given number of times, then the prefetch would not be performed. A similar filter was examined by Luk and Mowry [53] where a pollution counter is kept with each cache block in the L2 cache. This counter keeps track of the number of times a cache block was prefetched from the L2, but not used. When the counter was above a filter threshold the prefetch request would be cancelled. They found this to be a very beneficial for a prefetching architecture that stored prefetched blocks directly into the instruction cache. We did not examine this pollution filter because we store our prefetched blocks into a prefetch buffer before moving those that hit into the instruction cache.

X.A.5 Software Based Instruction Prefetching

Many software techniques have been developed for improving instruction cache performance. Techniques such as basic block re-ordering [65], function grouping [65], reordering based on control structure [56], and reordering of system code [86] have all been shown to significantly improve instruction cache performance.

These code placement techniques can be used to reduce instruction cache misses, but they also allow next-line and stream buffer prefetching architectures to achieve better performance. Xia and Torrellas [93] examined this effect, along with the addition of guarding bits to the ISA to guide instruction cache prefetching. One goal of basic block reordering is to place basic blocks sequentially for the most likely path through a procedure. The guarding bit would be used in conjunction with basic block placement to indicate the end of a sequence (chain) of placed basic blocks. Then when performing next-line prefetching, the next-line prefetcher would stop when it hits a guarding bit. Luk and Mowry [53] examined adding prefetch instructions to prefetch target basic blocks of branches that are not detectable by next-line prefetching. This would be particularly useful for doing interprocedural prefetching for procedures that are likely to be called.

Examining the effects of code placement and software guided prefetching were beyond the scope of what we were able to investigate in this thesis. Using these techniques in combination with fetch directed prefetching and our filtering techniques is a topic for future research.

X.A.6 Lockup-free Caches

Lockup-free caches were originally proposed to increase the performance for a unified instruction and data cache [47], and have been shown to improve the performance of data caches by allowing multiple outstanding loads [26]. Lockup free caches are inexpensive to build, since they only require a few Miss Status Holding Registers (MSHRs) to hold the information for an outstanding miss. When the missed block has been completely fetched from memory it is then inserted into the cache. In this thesis we make use of a lockup-free instruction cache to implement a form of prefetch filtering during an instruction cache miss.

X.B Cache Probe Filtering

There are two sources of wasted prefetches: redundant prefetches and useless prefetches. A redundant prefetch is one that attempts to bring in a cache block that is already contained in the instruction cache. A useless prefetch is a prefetch request that will never be used by the program being executed. Both of these will waste the bus bandwidth from the L2 cache and can severely impact performance by interfering with useful prefetches and demand misses. Useless prefetching can be avoided through more intelligent prefetching strategies and predictors. We now investigate a technique to avoid redundant prefetching.

When the instruction cache has an idle port, the port can be used to check whether or not a potential prefetch address is already present in the cache. We call this technique *Cache Probe Filtering* (CPF). If the address is found in the cache, the prefetch request can be canceled, thereby saving bandwidth. If the address is not found in the cache, then in the next cycle the block can be prefetched if the L2 bus is free. Cache probe filtering only needs to access the instruction cache's tag array. Therefore, it may be beneficial to add an extra cache tag port for CPF, since this would only affect the timing of the tag array access, and not the data array. As we will show, the impact of an extra port on the tag array is relatively minimal.

An instruction cache port can become idle when (1) there is an instruction cache miss, (2) the current instruction window is full, (3) the decode width is exhausted and there are still available cache ports, or (4) there is insufficient fetch bandwidth. To use the idle cache ports to perform cache probe filtering during a cache miss (1), the cache needs to be lockup-free. The cache need not be lockup-free to benefit from (2), (3), or (4) however.

In our simulations we examine performance with and without cache probe filtering. We model cache port usage in our simulations, and *only* allow cache probe filtering to occur when there is an idle cache port. First, we improve stream buffers with cache probe filtering. Then, we will examine a novel prefetching architecture that makes use of cache probe filtering.

X.B.1 Stream Buffer Modifications

We investigated using cache probe filtering with stream buffers. Each stream buffer follows a sequential path of cache blocks. When an idle cache port is available we can use it to verify whether or not a cache block in the path that the stream buffer is following is already in the instruction cache. If it is already in the cache block, we do not prefetch the block. If it is not in the cache, we prefetch the block. This filter can be used in one of two ways. If a cache block is found in the instruction cache, the prefetch can be skipped, and the stream buffer can continue as usual and attempt to prefetch the next cache block. This is labeled as

Table X.1: Port Data for a 16KB 2-way set-associative instruction cache

num read/write ports	extra tag ports	Access Time (ns)	Energy Dissipated (nJ)
1	0	.65	2.4
1	1	.67	2.4
2	0	.77	2.9
2	1	.78	2.9

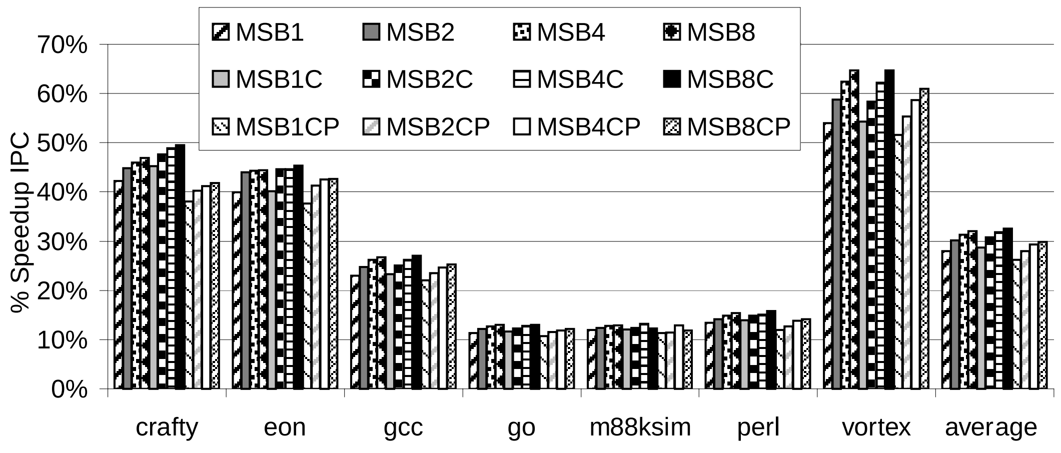
Port comparison for a 16KB 2-way set-associative instruction cache. Results are derived from CACTI timing data for a successful access to the cache.

MSBxC in the remainder of this thesis (where x is the number of stream buffers). Alternatively, once a cache block is found in the instruction cache, the prefetch can be skipped, and the stream buffer can be prevented from producing further prefetches until it is reallocated. We refer to this latter technique as the stop filter, labeled as *MSBxCP*.

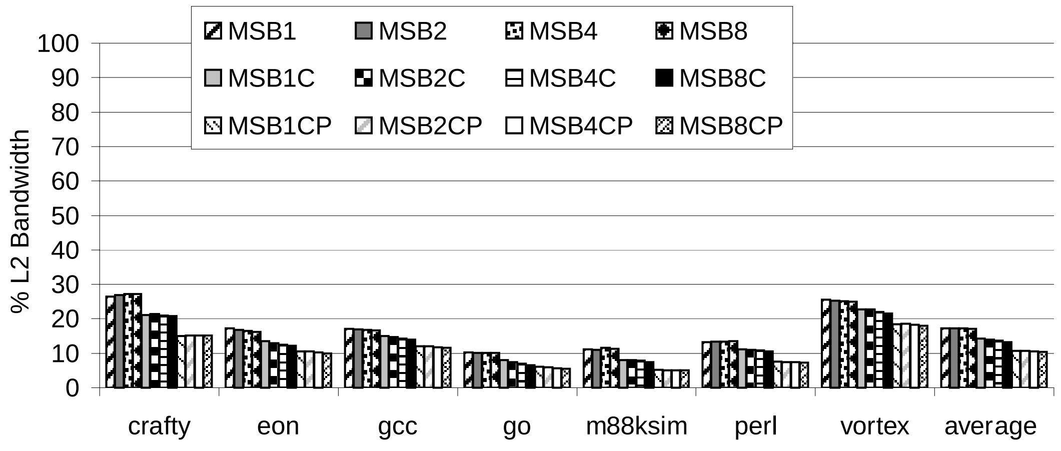
X.B.2 Results

We now compare the relative performance of different stream buffer configurations, with and without cache probe filtering. We used the base architecture from Chapter VI with a 16K 2-way set associative instruction cache with a single read/write port. The cache is lockup-free and also has an extra port on the tag array only. The timing and energy dissipation impact of an extra tag port is negligible, and is summarized in Table X.1. A 128 entry first level FTB is used with a 2048 entry second level FTB.

We implemented stream buffers as described earlier in section X.A.3. In addition, we used the uniqueness filter proposed by Farkas et. al [27]. Therefore, a given cache block will only appear in one stream at a time, and all valid entries in all stream buffers are checked for a hit in parallel with the instruction cache lookup. We found that a four entry stream buffer provided the best performance



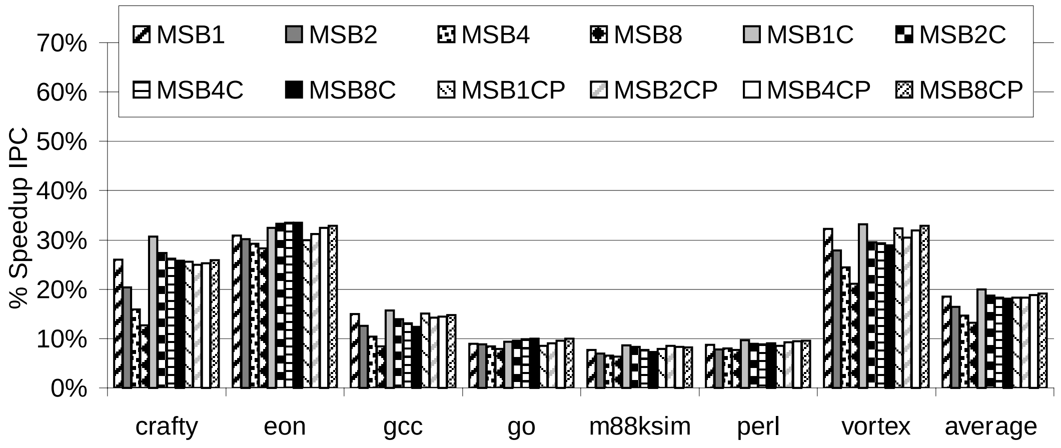
(a)



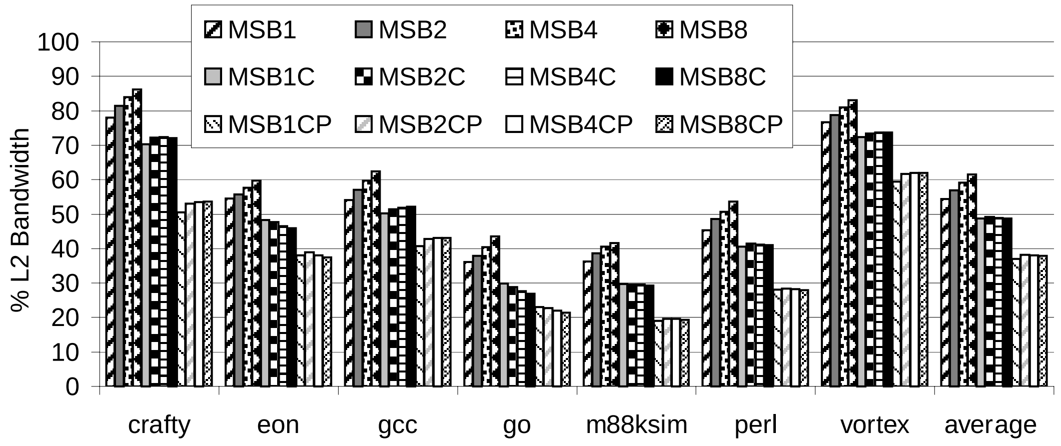
(b)

Figure X.1: Stream buffers (high bandwidth L2 bus).

These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using stream buffers over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and a 32 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The first four bars represent stream buffer configurations with no cache probe filtering (but with the uniqueness filter of [27]. Four configurations are shown: a single stream buffer and multiple stream buffers with 2, 4, or 8 buffers. The next four bars represent the four stream buffer configurations using cache probe filtering. The final four bars represent the four stream buffer configurations with both cache probe filtering and stop filtering.



(a)



(b)

Figure X.2: Stream buffers (low bandwidth L2 bus).
These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using stream buffers over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and an 8 byte/cycle bus. These Figures show the same configurations as Figures X.1(a) and (b), but with a lower bandwidth L2 bus.

for our pipeline architecture. We provide IPC results using a single four entry stream buffer (MSB1), an architecture with 2 four entry stream buffers (MSB2), 4 four entry stream buffers (MSB4), and 8 four entry stream buffers (MSB8) in Figure X.1(a) and Figure X.2(a). We also provide IPC results for the same four architectures using cache probe filtering - MSB1C, MSB2C, MSB4C, MSB8C. And we examine the use of cache probe filtering and stop filtering - MSB1CP, MSB2CP, MSB4CP, MSB8CP. Figures X.1(a) and (b) provide percent speedup in IPC and percent L2 bus utilization respectively for a high bandwidth L2 bus (32 byte/cycle bus). Figures X.2(a) and (b) provide percent speedup in IPC and percent L2 bus utilization respectively for a low bandwidth L2 bus (8 byte/cycle bus).

Stream buffers performed well (ranging from 28% IPC speedup for 1 stream buffer to 32% IPC speedup on average for 8 stream buffers), but performance degraded when used at a lower L2 bus bandwidth. Single stream buffers performed better at the lower bus bandwidth (18% on average), while the 8 stream buffers provided 13% performance improvement on average. The benchmark `crafty` experiences the most dramatic decrease in performance between 1 and 8 stream buffer configurations at the lower bus bandwidth. At the higher bus bandwidth, the stream buffers used around 17% of the L2 bus on average, but used between 54% and 61% of the bus bandwidth on average at the lower bus bandwidth. This helps to explain the performance difference as useful prefetch requests and demand misses conflicted with useless prefetches and redundant prefetches. As we will see, there is sufficient prefetch bandwidth to achieve even higher levels of performance – it is just a matter of intelligently selecting which blocks to prefetch.

When using cache probe filtering, about the same performance is achieved for high bus bandwidth (between 29%-33% IPC speedup on average), but larger speedups are seen for the low bus bandwidth (between 18%-20% IPC speedup on average). CPF reduces the bus bandwidth utilized – from around 17% on average to around 13%-14% on average for the high bandwidth bus results. For the low bandwidth bus, CPF has even more of an impact and reduces utilization to around 49%.

Stop filtering can reduce bus utilization even further: around 10%-11% on average for a high bandwidth bus architecture and around 37%-38% on average for a low bandwidth bus architecture. However, performance degrades using this approach due to a loss of useful prefetches. Still, the 8 stream buffer configuration improves from 13% on average to 19% on average using this technique at a low bus bandwidth.

On average, the best performance is seen using 8 stream buffers with cache probe filtering on an architecture with a high bandwidth L2 bus. For an architecture with a lower bus bandwidth, the best performance is seen using a single stream buffer with cache probe filtering.

X.C Fetch Directed Prefetching

Fetch Directed Prefetching (FDP) [72] follows the predicted stream, enqueuing prefetches from the FTQ. This is made possible if the branch prediction architecture can run ahead of the instruction fetch, which is what the FTQ based branch predictor provides (shown in Chapter IX). One advantage of this design, is that FDP can continue to prefetch down the predicted stream even when the instruction cache is stalled. We now describe our Fetch Directed Prefetching architecture, describe the heuristics that were used to better select which fetch blocks to prefetch, and evaluate their performance.

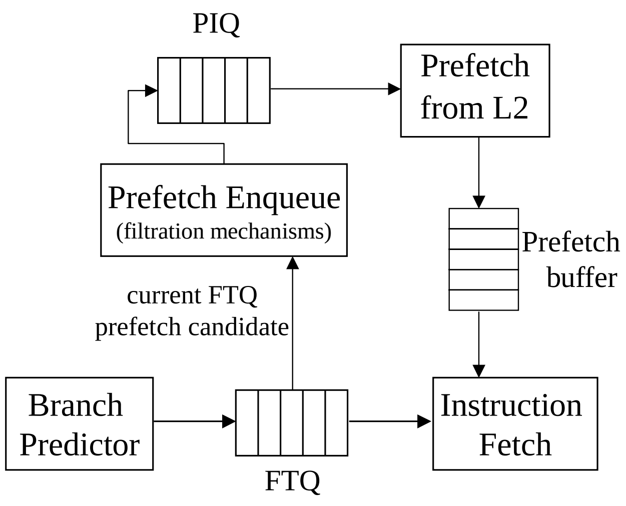


Figure X.3: Fetch Directed Prefetching Architecture.

X.C.1 Fetch Directed Prefetching Architecture

Figure X.3 shows the FDP architecture. As described in section VIII, we use a decoupled branch predictor and instruction cache, where the FTQ contains the fetch blocks to be consumed by the instruction cache. The FDP architecture uses a *Prefetch Instruction Queue* (PIQ), which is a queue of cache block addresses waiting to be prefetched. A prefetch from the PIQ will start when the L2 bus is free, after first giving priority to data cache and instruction cache demand misses.

One of the benefits of the FTB branch predictor design is that it can provide large fetch blocks (Chapter IX). A fetch block from one prediction can potentially span 5 cache blocks. Each fetch block entry contains a valid bit, a *candidate* prefetch bit, and an *enqueued* prefetched bit, for each possible cache block address. The candidate bit indicates that the cache block is a candidate for being prefetched. The bit is set using filtration heuristics described below. The enqueued bit indicates that the cache block has already been enqueued to be prefetched in the PIQ. Candidate prefetches from FTQ entries are considered in FIFO order from the FTQ, and are inserted into the PIQ when there is an available entry. The current FTQ entry, under consideration for inserting prefetch requests into the PIQ, is tracked via a hardware-implemented pointer.

A fetch directed fully-associative prefetch buffer is added to the FDP architecture to hold the prefetched cache blocks. This is very similar to a stream buffer [40], except that it gets its prefetch addresses from the FTQ. Each time a cache block is inserted into the PIQ for prefetching, an entry is allocated for that cache block in the prefetch buffer. If the prefetch buffer is full, then no further cache blocks can be prefetched. When performing the instruction cache fetch, the prefetch buffer is searched in parallel with the instruction cache lookup for the cache block. If there is a hit in the the prefetch buffer, the cache block is removed from the buffer and inserted into the instruction cache. The prefetch buffer contains a replacement bit, which is cleared on a branch misprediction and set when the entry is allocated. The entry is not cleared on a branch misprediction though, and if there is a prefetch buffer hit on an entry with a cleared replacement bit, the replacement bit is set and the entry is preserved. This helps reduce prefetch bandwidth on mispredicted short forward branches.

We examined several approaches for deciding which FTQ entries to prefetch and insert into the PIQ, which we describe in the following sections.

X.C.2 Filter Based on Number of FTQ Entries

The earlier the prefetch can be initiated before the fetch block reaches the instruction cache, the greater the potential to hide the miss latency. At the same time, the farther the FTQ entry is ahead of the cache, the more likely that it will be on a mispredicted path, and the more likely a prefetch from the FTQ entry might result in a wasted prefetch.

We examined filtering the number of FTQ entries to be considered for prefetching based on the position of the fetch block entry in the FTQ. Our prior results showed that when using an FTQ, its occupancy can be quite high (Chapter VIII). We found that starting at the 2nd entry from the front of the FTQ and going up to 10 entries in the FTQ provided the best performance. The FTQ we implemented can hold up to 32 entries, but stopping prefetching at 10 entries provided good performance, since prefetching farther down the FTQ resulted in decreased probability that the prefetch would be useful, and potentially wasted memory bandwidth.

X.C.3 Cache Probe Filtering

Cache probe filtering (CPF) [72] uses idle cache ports to check if a potential prefetch request is in the cache. We examined two approaches to CPF.

The first approach, called enqueue cache probe filtering (Enqueue CPF), will *only* enqueue a prefetch into the PIQ from the FTQ if it can first probe the instruction cache using an idle cache port to verify that the cache block does not exist in the first level instruction cache. This is a very conservative form of prefetching. This is the same type of filtering that was used with stream buffers in Chapter X.B.

The second approach, called remove cache probe filtering (Remove CPF), enqueues all cache blocks into the PIQ by default, but if there is an idle first level cache port, it will check the cache tags to see if the address is already in the cache. If the prefetch is in the cache, the prefetch entry will be removed from the list of potential prefetch addresses. If there are no idle cache ports, then the request will be prefetched *without* first checking the cache.

X.C.4 Fetch Block Evicted Prefetching

The next approach examines keeping track of cache blocks that are evicted from the instruction cache in the branch prediction architecture, and

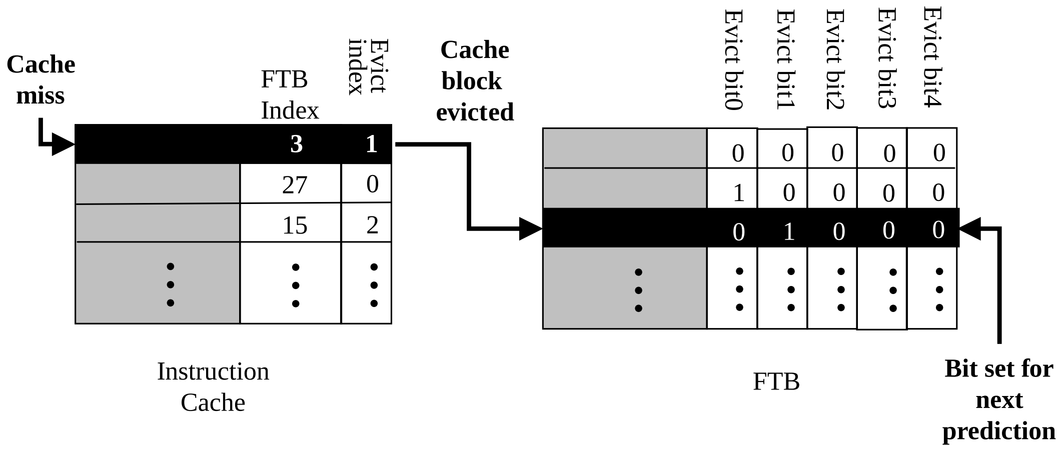


Figure X.4: Eviction Prefetching
This Figure illustrates how the FTB can be augmented with additional evict bits to track potential cache misses. The instruction cache contains an FTB index and an evict index. The FTB index is used to determine what fetch block corresponds to the given cache block (*i.e.* which FTB entry the cache block would reside in). As each fetch block can contain up to 5 cache blocks, the evict index is necessary to complete the cache block mapping. Each FTB entry is augmented with 5 evict bits to represent each of the 5 cache blocks. The evict index in the instruction cache selects which of the 5 evict bits the given cache block corresponds to. When an FTB prediction is made, the evict bits determine whether or not the cache block they represent will be prefetched. In this Figure, we show how an FTB evict bit is set. A cache miss will cause a certain cache block to be replaced in the cache. This is shown on the left side of the Figure: the blackened cache line will be removed from the cache and replaced with the block that missed. The line to be removed contains an index into the FTB, which can then be used to set the corresponding evict bit in the FTB. Later, when the FTB provides this entry for a prediction, the second cache block in the fetch block will be prefetched.

then using this to guide our fetch directed prefetcher. It can be beneficial for the branch predictor to keep track of state for branches that have been evicted from the instruction cache. A multi-level FTB is one possible method of providing a larger branch target buffer.

We store *Evict* bits in each FTB entry and they are set when a cache block of the corresponding fetch block is evicted from the instruction cache. Since a fetch block can potentially span 5 cache blocks, there are 5 evict bits stored in each FTB entry. There also needs to be some way of linking cache blocks in the instruction cache to FTB entires. The implementation we examined stores N bits with each cache block in the instruction cache to identify the FTB entry that last caused the cache block to be brought into the cache. For the implementation we simulated, 12 bits are used to map the block to the FTB entry – 10 bits to index into the set, and 2 bits for the way (4-way associative FTB). Since bits stored in the cache are a direct index into the FTB, there is no guarantee that the entry that loaded the cache block is still in the FTB, but for an FTB which can hold more state and have a larger associativity than the instruction cache, the mapping will likely be correct. Figure X.4 illustrates an example of this mapping.

When a cache block is evicted from the cache, the N bit index is used to access the FTB. The evict bit in the FTB entry corresponding to that index is set, indicating that the cache block will be prefetched the next time it is used as a branch prediction. An eviction can cause at most one prefetch, since the evict bit is cleared for an FTB entry when it is inserted into the FTQ.

X.C.5 Cache Miss Filtering

It is desirable to concentrate on prefetching only those fetch blocks that will most likely miss in the instruction cache in order to reduce bus utilization. If a given cache *set* has a lot of conflict misses, then it can be beneficial to prefetch all blocks that map to that high conflict cache set. To capture this behavior we examine using a confidence counter associated with each instruction cache set to indicate which cache sets miss most frequently. Fetch blocks that access these cache sets will have a greater chance of missing in the instruction cache and will therefore be worth prefetching.

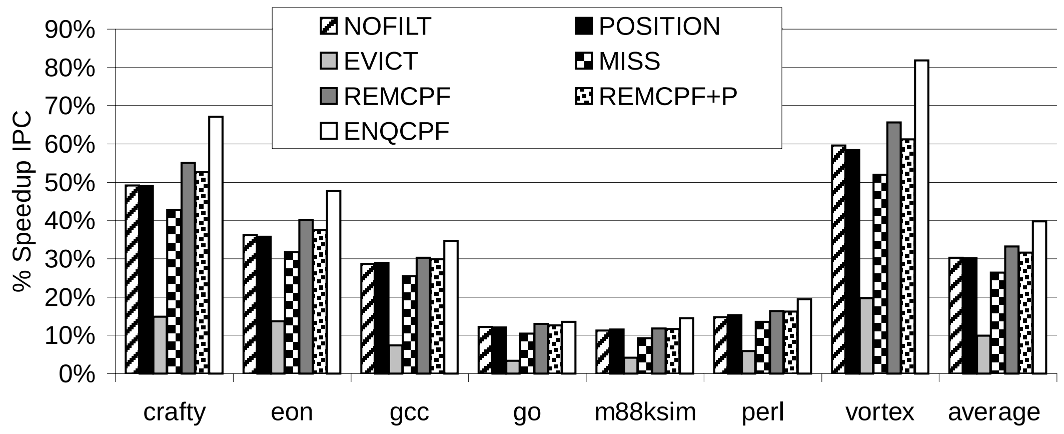
We examined adding a cache miss buffer that contains a 2-bit saturating counter for each instruction cache set. We index the cache miss buffer in parallel with the FTB. If the fetch block being predicted maps to a cache set that has missed frequently in the past, that fetch block is marked to be prefetched.

We examined several different finite state machines for the miss counters, and found the following to work well with fetch directed prefetching. When a cache miss occurs, its corresponding set counter in the miss buffer is incremented. A cache hit does not change the cache set miss counters. Instead, the confidence counters are cleared every million cycles to prevent extraneous prefetching.

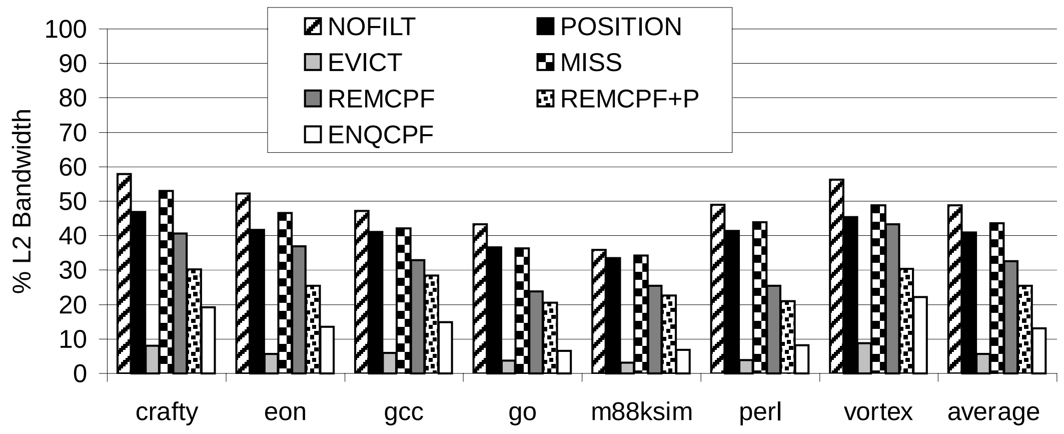
X.C.6 FDP Results

Figures X.5 and X.6 show results for fetch directed prefetching without any filtering (NOFILT), for FTQ position filtering (POSITION), eviction filtering (EVICT), cache miss filtering (MISS), remove cache probe filtering with and without position filtering (REMCPF and REMCPF+P), and enqueue cache probe filtering (ENQCPF). Figure X.5 shows percent IPC speedup (a) and percent L2 bus utilization (b) for a 32 byte/cycle L2 bus. Figure X.6 shows percent IPC speedup (a) and percent L2 bus utilization (b) for a 8 byte/cycle L2 bus.

Fetch directed prefetching, even without any filtering techniques, provides substantial benefits (30% on average for a bandwidth of 32 bytes/cycle). As can be seen in the Figure, this technique uses a great deal of bus bandwidth (49% bus utilization). For an 8 byte/cycle bandwidth, utilization reaches 77% on

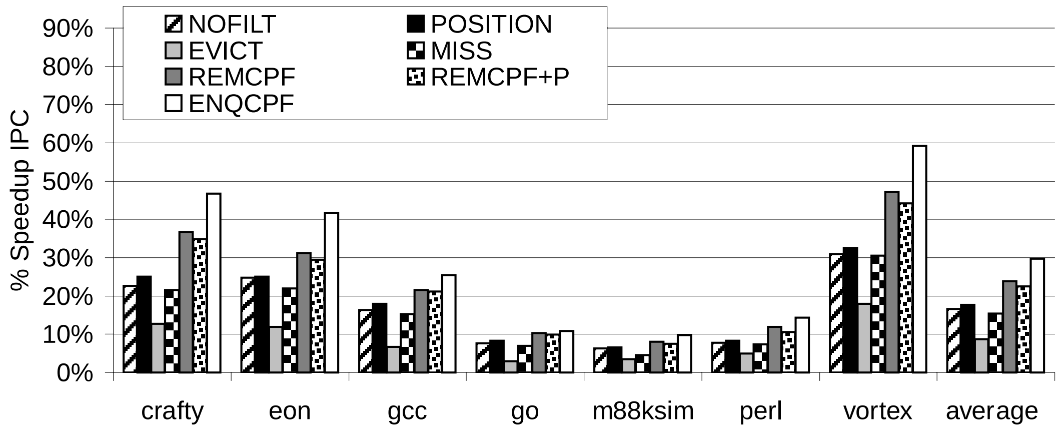


(a)

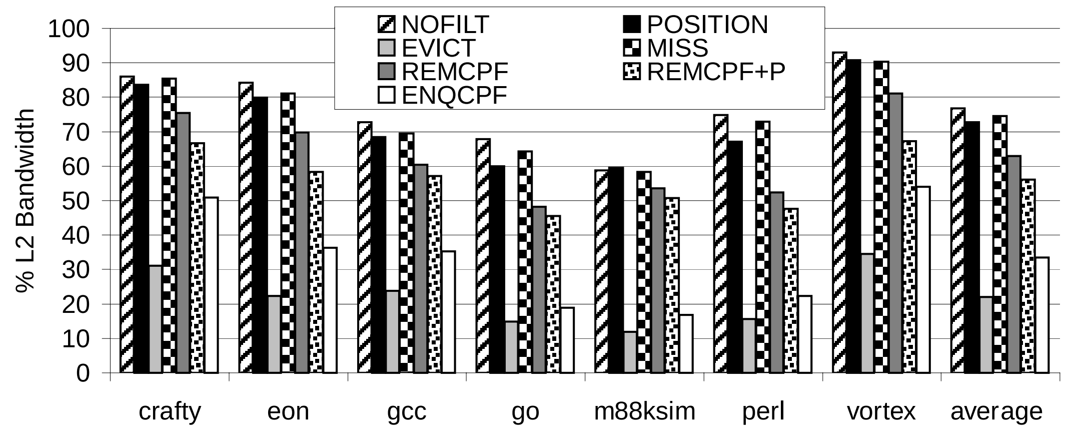


(b)

Figure X.5: Fetch directed prefetching (high bandwidth L2 bus).
These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using fetch directed prefetching (FDP) over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and a 32 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The first bar represents the performance of FDP without any form of filtration (NOFILT). The second bar examines filtering FDP based on the position of the entry in the FTQ (POSITION). We examine just using the next 10 FTQ entries to be consumed. The third bar examines FDP with evict filtering (EVICT). The fourth bar examines FDP with cache miss filtering (MISS). The fifth (REMCPF) and sixth (REMCPF+P) bars are cache probe filtering using the remove filter. The sixth bar also uses position filtering to remove further prefetches. The seventh bar uses cache probe filtering with the enqueue filter (ENQCPF).



(a)



(b)

Figure X.6: Fetch directed prefetch (low bandwidth L2 bus).
These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using fetch directed prefetching (FDP) over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and an 8 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The architectures examined are the same as in Figure X.5.

average, and the IPC speedup obtained drops to 17% on average. For `vortex`, the bandwidth utilized exceeds 90%.

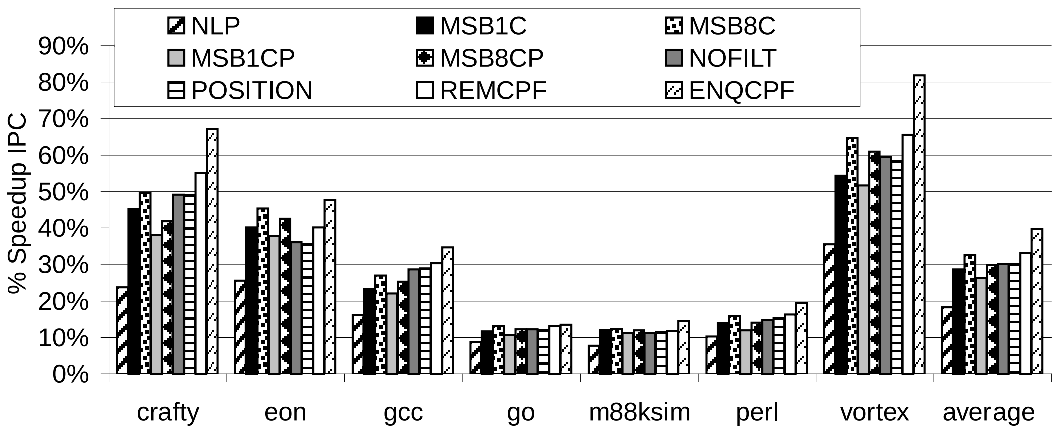
Position filtering reduces the bandwidth used by FDP (to 41% and 73% for the high and low bandwidth L2 buses respectively). The 2-bit cache miss filter does reduce bandwidth, but does not perform as well as the position filter.

Eviction-based prefetching alone provides an average 10% speedup for the high bandwidth case, while only using 6% of the bus bandwidth. For the low bandwidth case, it provides an average 9% speedup in IPC and uses only 22% of the bus bandwidth. This type of prefetching is highly accurate, but is extremely conservative.

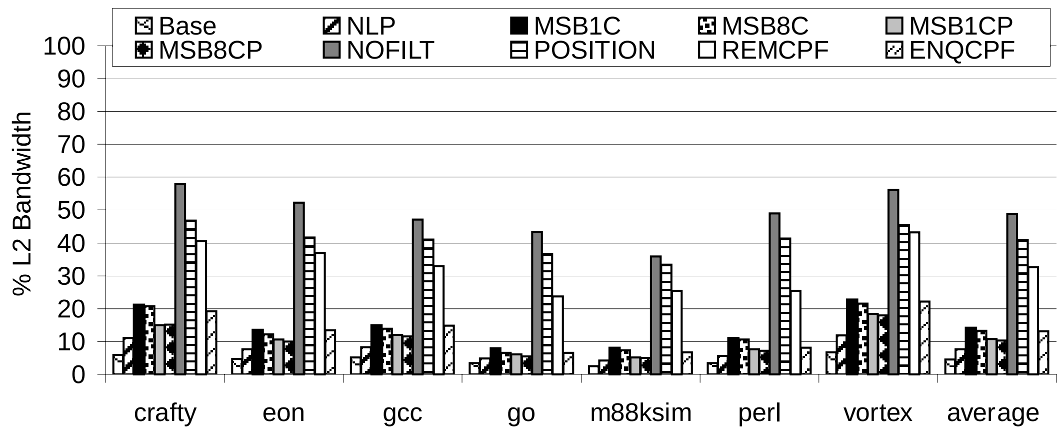
Cache probe filtering provided the best performance out of all filtering techniques examined. Remove CPF provided 33% and 23% speedups in IPC on average for the high and low bus bandwidth architectures (respectively). It reduced the bandwidth utilized to 33% and 63% for these respective architectures. If position filtering is used along with remove CPF, the bandwidth is reduced even further, with minimal performance impact. Enqueue CPF provided the best performance for all configurations and used less bandwidth than any other technique with the exception of evict filtering. For a 32 byte/cycle bus, it provided an additional 33% average improvement in IPC over FDP without filtering while reducing the bandwidth requirements by 73%. The benchmark `vortex` experiences considerable speedup with enqueue CPF due to accurate branch prediction, high FTQ occupancy, and a large instruction cache footprint.

X.C.7 Comparison to prior work

Figures X.7 and X.8 show the best performing configurations from our FDP architectures and our enhanced stream buffers. We also include tagged next-line prefetching (NLP) as described in [81] – except that we prefetch the

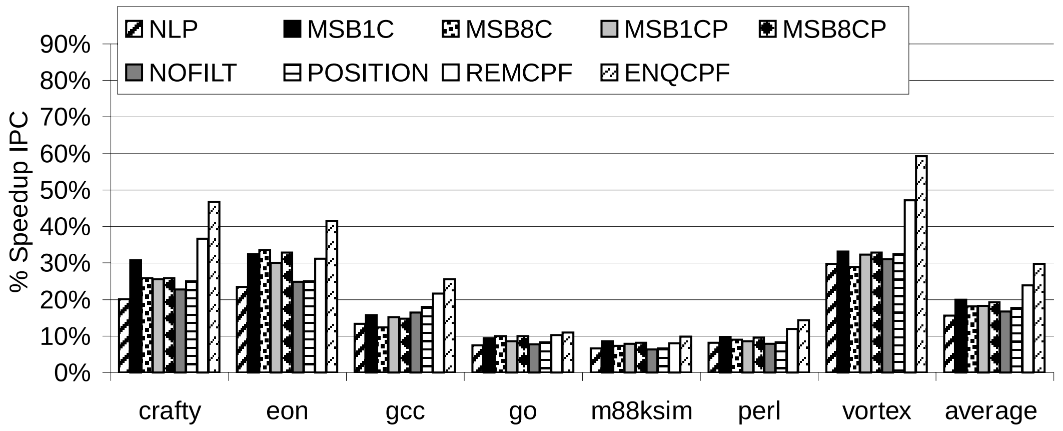


(a)

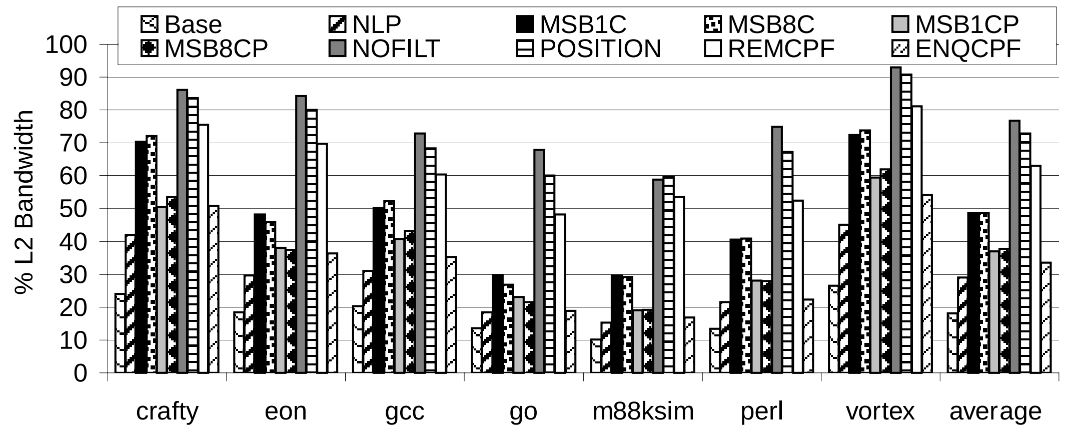


(b)

Figure X.7: Prefetching comparisons (high bandwidth L2 bus).
These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using a variety of prefetching techniques over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and a 32 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The first bar represents tagged next-line prefetching (NLP). The next four bars represent different stream buffer configurations: 1 and 8 stream buffers with cache probe filtering (MSB1C and MSB8C) - and - 1 and 8 stream buffers with cache probe filtering and stop filtering (MSB1CP and MSB8CP). The sixth bar examines the performance of FDP without any form of filtration (NOFILT). The seventh bar examines filtering FDP based on the position of the entry in the FTQ (POSITION). The eighth bar is cache probe filtering using the remove filter (REMCPF). The ninth bar uses cache probe filtering with the enqueue filter (ENQCPF). In (b), an additional bar is added to show the bandwidth requirements of the baseline architecture.



(a)



(b)

Figure X.8: Prefetching comparisons (low bandwidth L2 bus).
These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using a variety of prefetching architectures over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and an 8 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The architectures examined are the same as in Figure X.7.

cache blocks into a fully associative, 32 entry prefetch buffer. This buffer is queried in parallel with the instruction cache during a lookup to find a potential hit. Adding target prefetching to NLP (results not shown) provided minimal improvement, as discovered by [66]. We also examined combining tagged next-line prefetching with stream buffers (results not shown), but this provided only a slight improvement. This is because the stream buffers follow the sequential path after a cache miss, which overlaps the prefetch blocks that next-line prefetching captures.

Figure X.7 shows percent speedup in IPC (a) and percent bus utilization (b) for a 32 byte/cycle bus to the L2 cache using a number of prefetching architectures: tagged next-line prefetching (NLP), a single stream buffer with cache probe filtering (MSB1C), 8 stream buffers with cache probe filtering (MSB8C), a single stream buffer with cache probe filtering and stop filtering (MSB1CP), 8 stream buffers with cache probe filtering and stop filtering (MSB8CP), FDP with no filtering (NOFILT), FDP with position filtering (POSITION), remove cache probe filtering (REMCPF), and enqueue cache probe filtering (ENQCPF). Figure X.7(b) also includes a bar for the base architecture bandwidth utilization. Figure X.7(b) includes bandwidth utilized by both instruction cache demand misses and data cache demand misses as well as by instruction cache prefetching. Figure X.8 shows the same results as Figure X.7, but for an 8 byte/cycle bus to the L2 cache.

NLP provides good speedups for high and low bandwidth (18% and 15%), while using very little bandwidth beyond what the base architecture uses. Enqueue CPF attains the highest speedup for all bus configurations and benchmarks. The benchmark `go` shows little benefit from using enqueue CPF over a simple stream buffer, but this is due to both the relatively poor branch prediction accuracy that is seen with `go` and the relatively low instruction cache miss rate in `go`. The most notable improvements are seen once again in `vortex` (and in `crafty`).

X.C.8 Impact of FTQ size

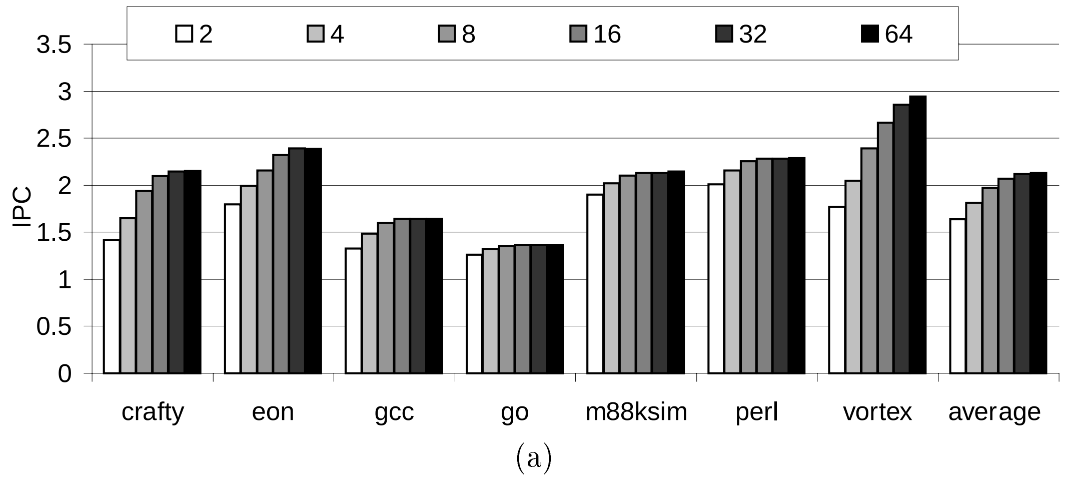
The size of the FTQ can influence how far ahead a fetch directed prefetching mechanism is able to look in the future fetch stream. Instead of using position filtering, it is also possible to vary the size of the FTQ itself to reduce the amount of prefetch bandwidth being used. However, this will also impact other fetch directed predictors which may be using the FTQ. To determine the ideal sized FTQ for fetch directed prefetching, we examined the performance of different FTQ sizes for a number of different architectures.

Figures X.9 and X.10 show the impact of FTQ size on the best performing prefetching scheme: enqueue CPF. For both bus configurations, most of the benefit of enqueue CPF can be captured using a 32 entry FTQ. Only `vortex` experiences a significant speedup using a 64 entry FTQ over the 32 entry configuration. Some benchmarks actually experience a very small degradation in performance with a larger FTQ, like `eon` and `m88ksim`. As would be expected, the bandwidth utilized also increases with the size of the FTQ.

X.C.9 Impact of cache size

In this Chapter, we made use of a 16K 2-way set-associative instruction cache. In Figure X.11, we examine the impact of cache size on the performance of our FDP architectures with CPF. We vary the size of the instruction cache across the x-axis and present the IPC obtained by a base architecture with no prefetching, an FDP architecture with remove CPF, and an FDP architecture with enqueue CPF.

A 16KB 2-way set-associative cache that uses enqueue CPF achieves



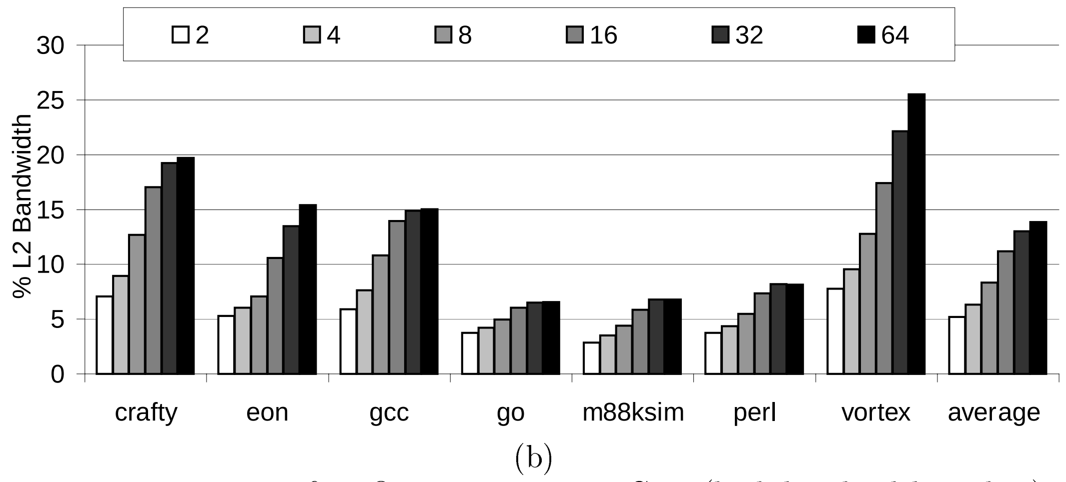
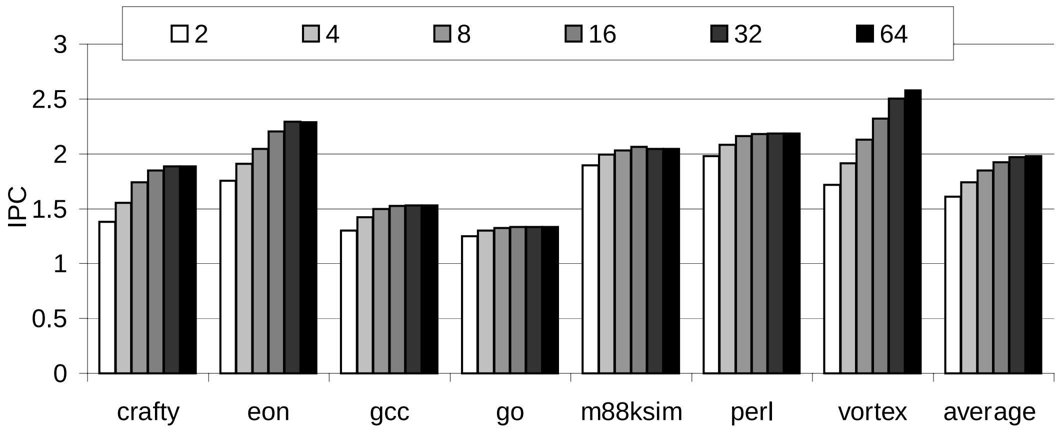
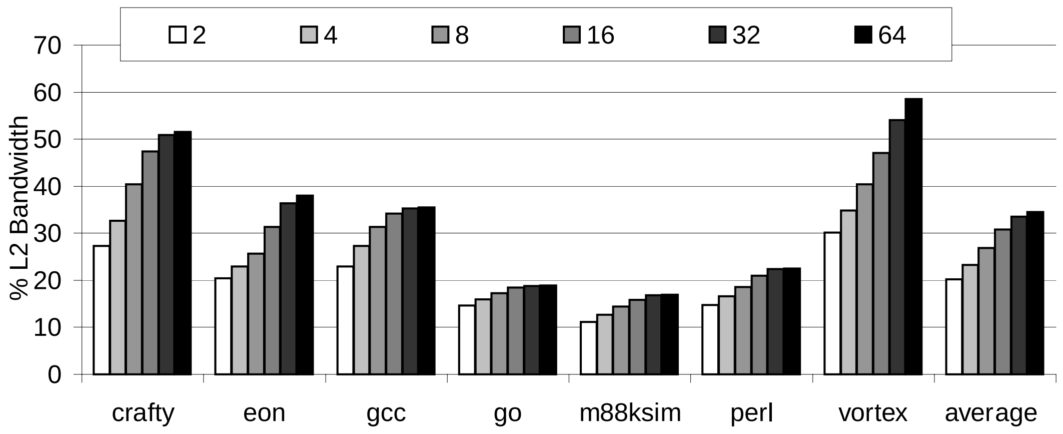


Figure X.9: Impact of FTQ size on enqueue CPF (high bandwidth L2 bus). These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using enqueue CPF over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and a 32 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The bars show varying FTQ sizes, from 2 to 64 entry FTQs.



(a)



(b)

Figure X.10: Impact of FTQ size on enqueue CPF (low bandwidth L2 bus). These Figures show percent speedup in IPC (a) and percent L2 bus utilization (b) using enqueue CPF over a baseline architecture with no instruction prefetching (and a 16K 2-way set associative cache). Here, the L2 cache has a single port and an 8 byte/cycle bus. We show results for 7 benchmarks (along the x-axis). The bars show varying FTQ sizes, from 2 to 64 entry FTQs.

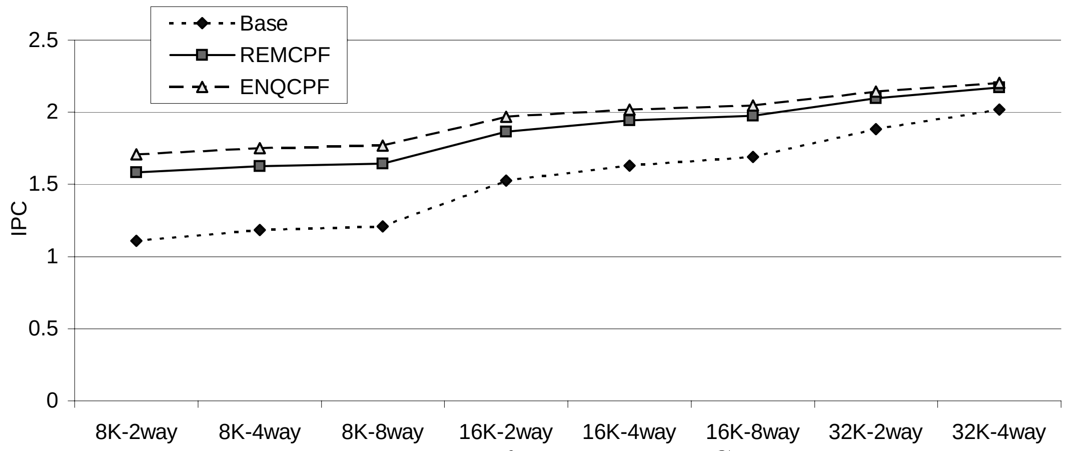


Figure X.11: Impact of cache size on CPF results.
Results here are shown for a low bandwidth (8 byte/cycle) bus to the L2 cache. Three lines are shown: a base architecture with no instruction prefetching, remove cache probe filtering, and enqueue cache probe filtering. Different cache configurations (both size and associativity) are displayed across the x-axis, ranging from an 8KB 2-way set-associative cache to a 32KB 4-way set associative cache. The y-axis shows the IPC obtained on average over 7 benchmarks.

comparable IPC to an architecture with a 32KB 4-way set-associative cache and no instruction cache prefetch. Even with a large 32KB cache, the CPF techniques still provide benefit over the base architecture.

X.C.10 Impact of cache ports

Another important architectural consideration is the number of ports on the instruction cache. In this Chapter, we examined an instruction cache with a single read/write port, but with an additional read port on the tag array. As shown in Table X.1, this additional port has a minimal impact on timing and energy dissipation, and in Figure X.12 we show the impact of this extra port on the performance of the CPF techniques in this Chapter. Figure X.12 shows the percent speedup in IPC obtained when adding an extra port to the tag array of an architecture with a single read/write port. As shown, there is minimal impact on the stream buffer configurations, but the performance of

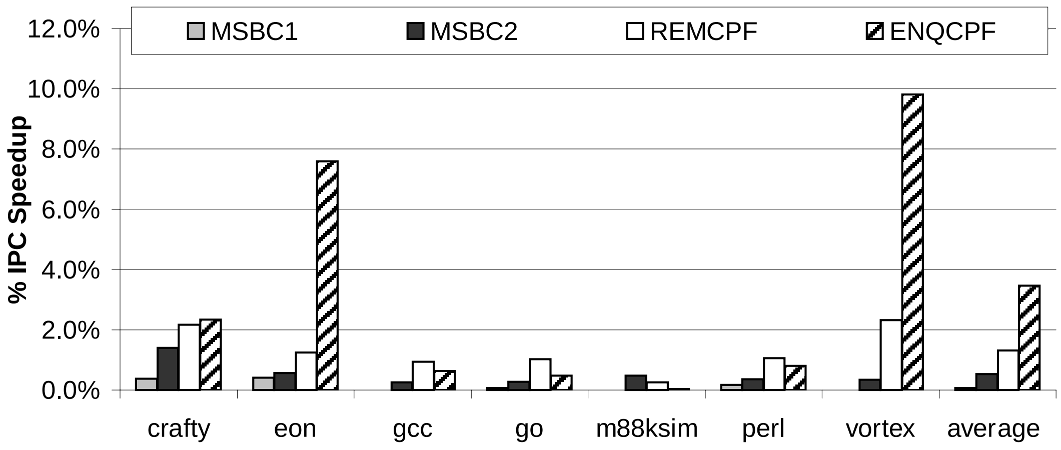


Figure X.12: Impact of extra tag port on single ported instruction cache. This Figure presents the speedup in IPC seen through the addition of an extra read port to the instruction cache tag array. The speedup is measured over an instruction cache with only a single read/write port. Results here are shown for a low bandwidth (8 byte/cycle) bus to the L2 cache. Four bars are shown: a single stream buffer with CPF, two stream buffers with CPF, remove CPF, and enqueue CPF. 7 benchmarks are displayed across the x-axis. The y-axis shows the percent speedup in IPC for each architecture relative to a cache without the extra tag port.

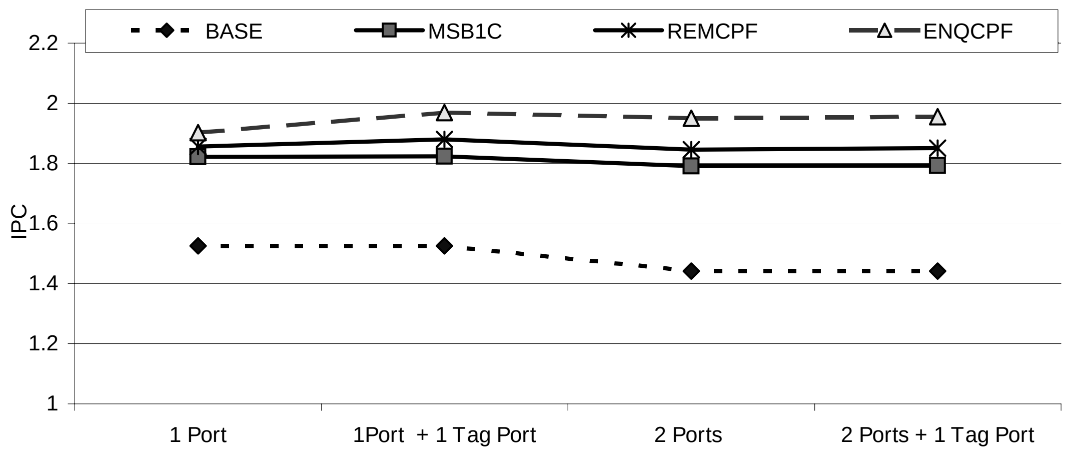


Figure X.13: Average IPC across different port configurations.
This Figure shows how a base architecture with no prefetching (BASE), an architecture with a single stream buffer using CPF (MSB1C), an architecture with FDP filtered by remove CPF (REMCPF), and an architecture with FDP filtered by enqueue CPF (ENQCPF) perform using different port configurations on the instruction cache. The port configurations are shown across the x-axis and IPC is shown on the y-axis. IPC values are averaged over the 7 benchmarks we examine in this Chapter. We make use of a 16K 2-way set-associative instruction cache to obtain these results.

enqueue CPF is greatly enhanced for `vortex`, `eon`, and to some extent `crafty`. The additional port simply provides an opportunity for the filtering mechanism to enqueue prefetch requests even more rapidly. Adding even more tag ports provides minimal additional benefit for these benchmarks and architectures.

Figure X.13 examines four different port configurations on a 16K 2-way set-associative instruction cache. The IPC results in this Figure demonstrate that adding an extra read/write port to our architecture actually degrades performance for the base architecture (with no prefetching). Figure VIII.8 revealed that few stalls occur due to insufficient cache ports for the architectures we examine. And Figure VIII.9 revealed that increasing the number of cache read/write ports actually increases the cycles stalled due to instruction cache misses. The decrease in instruction cache performance comes from branch mispredictions, either due to predictor error or due to a second level FTB access which arrives too late (*i.e.* the fallthrough prediction from the first level FTB has already been consumed by the instruction fetch unit). This data, along with Figure VIII.9 indicates that the extra chip area that would be spent to dual port the instruction cache might be better spent on increasing the size of the instruction cache – for the particular architecture we've examined. Moreover, without a commensurate increase in the performance of the execution core (specifically, the issue width of the execution core), potentially doubling the bandwidth of the instruction fetch unit will have minimal positive impact on the performance of the processor as a whole. While the performance of the execution core may be limited by the front-end of the processor (as shown in Chapter I), the performance of the processor itself is fundamentally limited by the performance of the execution core.

As first shown in Figure X.12, the single ported cache obtains benefit from an additional tag array port. However, the dual ported cache receives almost no benefit from an additional tag array port as shown in Figure X.13.

For the results in this Chapter, we have used a two-level FTB configuration (using a 128 entry first level FTB). However, to ensure that there is not a better configuration which will provide more performance for a dual ported instruction cache, we also examine results for a variety of FTB configurations using a 16K 2-way set-associative dual ported instruction cache. Figure X.14 presents BIPS results for a number of different single and multi level FTB configurations. As in Chapter IX, the best performing single level predictor has 512 entries, and the best performing two level predictor has a 128 entry first level. The two level predictors still outperform the single level predictors, even with a dual ported instruction cache.

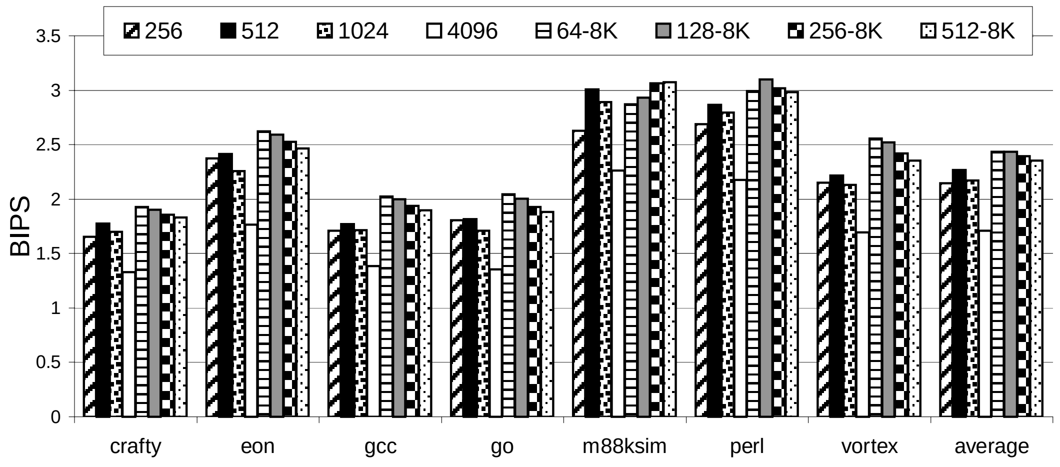


Figure X.14: Dual ported cache results with varied FTB configurations
This Figure provides BIPS results for a variety of FTB configurations using a dual ported instruction cache. Four single level configurations (256, 512, 1024, 4096 entry first levels) and four two level configurations (64, 128, 256, 512 entry first levels with 8192 entry second levels) are shown. This Figure uses $0.10\mu m$ technology data from CACTI and assumes no effects from the interconnect scaling bottleneck. The 7 benchmarks we examined in this Chapter are shown along the x-axis.

X.C.11 Impact of reorder buffer size

Finally, we explore the impact of reorder buffer size on some of the prefetching results. We used a 128 entry reorder buffer in this Chapter. Figure X.15 shows results for three reorder buffer configurations: 32, 128, and 512 entry reorder buffers. As can be seen in Figure VIII.8, there are very few stalls due to a full instruction fetch queue, and therefore very few stalls due to a full reorder buffer. So, for the 512 entry reorder buffer case, we also double the number of instructions that can issue in a single cycle (*i.e.* use a 16-way machine). Figure X.15 demonstrates how we can still provide considerable improvement using CPF with either a higher bandwidth execution core (larger ROB and fetch width) or a lower bandwidth execution core (smaller ROB).

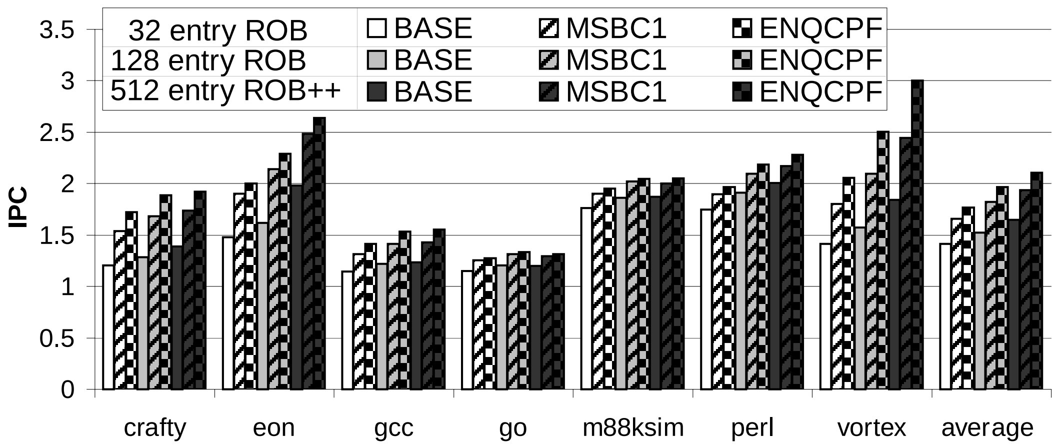


Figure X.15: Impact of reorder buffer size

This Figure presents three reorder buffer (ROB) configurations: a 32-entry ROB, a 128-entry ROB, and a 512-entry ROB. The 32-entry and 128-entry ROBs are used in 8-way out-of-order architectures. The 512-entry ROB is used in a 16-way out-of-order architecture. The y-axis gives the IPC of the three prefetching architectures we examine for each ROB configuration: a base architecture with no prefetching (BASE), a single stream buffer with CPF (MSBC1), and an FDP architecture with enqueue CPF (ENQCPF). The 7 benchmarks we examine in this Chapter are shown along the x-axis.

X.D Summary

In this Chapter, we have examined a number of instruction cache prefetching techniques. Prefetches need to be both accurate and timely, and should not interfere with demand misses. We have shown how various filtering techniques can be used to reduce demand miss interference, and can improve the timeliness of prefetching by making the most effective use of the available bandwidth. We can make the following observations about some of the more effective cache prefetching techniques for an 8 byte/cycle L2 bus:

- *Tagged Next-line Prefetching* – this technique uses very little L2 bus bandwidth, and can provide an average 16% speedup in IPC over a base architecture. It is a simple technique, and is easily implemented. However, it has no notion of prefetch usefulness. It can only follow a sequential prefetching path, but has no notion of whether or not that path will be taken by the processor. Moreover, it has limited prefetch timeliness, as it initiates the prefetch of a given block upon the use of the block immediately before.

- *Stream Buffers* – this technique is able to take advantage of the latency of a cache miss to hide a number of sequential path prefetches. Therefore, it exhibits excellent prefetch timeliness. However, it can have a relatively high amount of prefetch bandwidth. We show how cache probe filtering can reduce this however, and can substantially improve performance. Nevertheless, stream buffers still can only follow a number of sequential prefetching paths, and like NLP, the technique has no notion of prefetch usefulness. The best stream buffer configuration (using CPF) provided a 20% speedup in IPC on average.

- *Fetch Directed Prefetching* – this technique follows the predicted stream of fetch addresses stored in the FTQ. It can perform timely prefetches by either using the latency of a cache miss (much like stream buffers) or through sufficient occupancy in the FTQ. However, because it could potentially enqueue every cache block executed by the processor, it uses a substantial amount of L2 bandwidth. When using cache probe filtering, this bandwidth can be greatly reduced, and the performance can be greatly increased. Our best performing FDP technique provided 30% speedup in IPC on average. This technique is strongly linked to the accuracy of the branch predictor.

Chapter XI

Instruction Cache Energy and Complexity Optimizations

In Chapter X we explored using predictions stored in the FTQ to direct instruction cache prefetching. We used cache probe filtering (CPF) in Chapter X.C.3 to eliminate redundant prefetches and make more efficient use of the bus bandwidth to the L2 cache. In this Chapter, we focus on techniques to reduce both the complexity and energy dissipation of our prefetching architecture. We examine three different cache designs, explore way prediction, and propose a novel decoupled instruction cache design which allows complexity-effective and energy-efficient fetch directed instruction prefetching.

Throughout this Chapter, we make use of a 128-entry FTB with a 2048-entry second level FTB. This predictor couples a small first level table with a larger, but relatively energy efficient second level predictor.

XI.A Cache Design Tradeoffs

Instruction cache performance is vital to the processor pipeline. Associativity is a useful technique to improve cache performance by reducing conflict

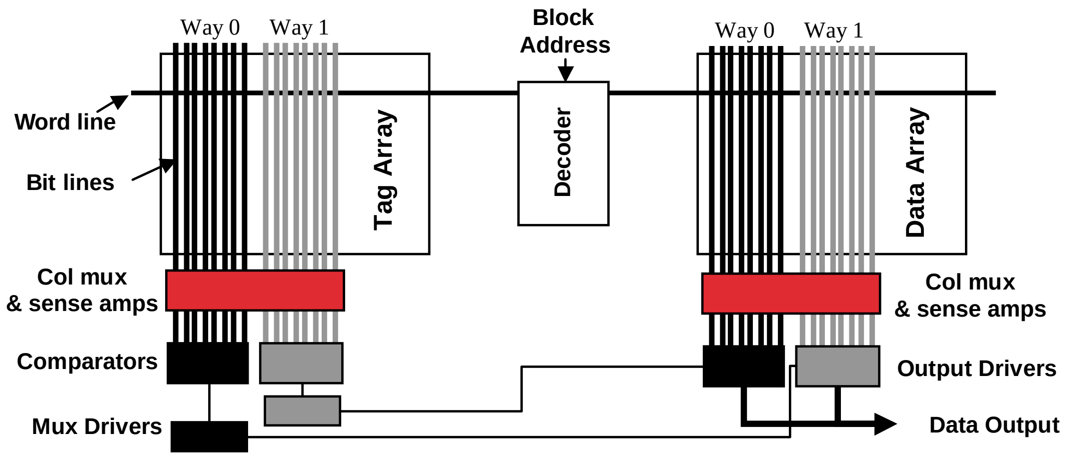


Figure XI.1: Cache configuration taken from CACTI [91]
The tag and data arrays are accessed in parallel. In a set associative cache, all ways of a cache set are driven on the data path, but a single way is selected for data output once the tag path completes. The ways of a particular cache set share a common wordline.

misses in the cache. However, direct mapped caches expend less energy on a given access than associative caches. Moreover, they typically have faster access times as the data output drivers do not need to wait for the tag comparators. Next line set prediction [15] has been proposed as a means of providing the timing and energy benefits of a direct mapped cache configuration, combined with the improved hit rate of an associative cache configuration. One type of cache configuration that is used with next line set prediction is the predictive sequential associative (PSA) cache [16]. We expand upon the PSA cache and introduce the multi-component serial cache, which provides the same energy benefits as the PSA cache, but has a slightly faster access time.

XI.A.1 Set-Associative Instruction Cache

A set-associative cache is split into two components, a tag component and a data component. These are indexed in parallel to reduce the access time of the cache. Throughout this paper we will refer to this traditional cache design as the *parallel cache.* All ways of a particular cache set are driven from the data component, and the desired way is selected based upon the comparator output of the tag component. Figure XI.1 shows a 2-way set associative cache with the bitlines representing way 0 of a cache set colored in black and the bitlines representing way 1 of a cache set colored in grey. In the cache we simulate, there are two bitlines (*bit* and $\overline{bit}$) for each bit of the cache block. These lines are precharged high, and on each access to a particular tag or data array, either *bit* or $\overline{bit}$ will be brought low for each bit on the driven wordline. For the data array, if the number of bits output by the cache is 256 (assuming an entire 32-byte block of the instruction cache is output) and the cache is 2-way set associative, then 512 bitlines will need to be charged before every access (one line from each bitline pair, as either *bit* or $\overline{bit}$ will be high already). Moreover, there will be 512 sense amps hooked up to *bit* and $\overline{bit}$, which consume a significant amount of power. For a 16KB 2-way set associative cache with 1 read/write port, over 94% of energy dissipated by the cache on a successful cache access is in the sense amps and the bitlines of the data component (using the CACTI 2.0 tool [74]). There are fewer bits in the tag array than the data array, and therefore the data array dominates the energy dissipation of the cache. Figure XI.1 also includes column multiplexors before the sense amps that are used when multiple tag or data arrays are used by CACTI. The sense amps feed the output drivers, which selectively output a particular way of the cache depending on values from the tag comparison (made in parallel with the data lookup). Only 256 bitline pairs and sense amps will actually determine the output of the cache, so around 50% of the energy dissipated in the bitlines and sense amps is wasted. Even on a cache miss, the bitlines and sense amps still consume power, as the data and tag components occur in parallel – the data component does not see the miss until the output driver stage. In this case, 100% of energy consumed in the data component of

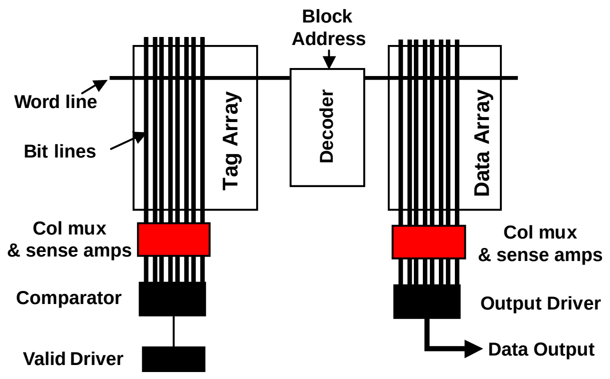


Figure XI.2: The direct mapped cache model.

the instruction cache is wasted. The effect of this is worse in caches with even higher associativities – a 4-way set associative cache has 1024 bitlines that must be precharged before every access, and makes use of 1024 sense amps – wasting around 75% of the total energy that is consumed in the sense amps and bitlines.

As we model an instruction cache that outputs an entire cache line on a successful access, we scaled the multiplexor driver on the tag path of [74] to be able to handle the large number of output drivers associated with this kind of cache. We were also able to eliminate some of the extra multiplexing done at this stage as the entire cache block is output.

XI.A.2 Predictive Sequential Associative Cache

Calder et al. [16] proposed the predictive sequential associative (PSA) cache to provide the performance of a set-associative cache with the access time of a direct mapped cache. The PSA cache is essentially a direct mapped cache, but stores cache blocks that map to the same cache way in sequential cache entries. For example, a 2-way PSA cache has half the number of sets as an equivalent direct mapped cache – each set is composed of two consecutive cache entries. In [16], a predictor determines which cache entry contains the desired block. The tag and data arrays are accessed in parallel, and on a misprediction, the correct block can be provided from the data array in the next cycle. Figure XI.2 shows the CACTI model for a direct mapped cache that is used in our timing analysis.

The PSA cache effectively uses a form of way prediction to determine what cache line to drive out of the direct mapped cache. This is similar to the way prediction technique used in the NLS architecture [15], but in this case, the way predictor is associated with the cache itself, rather than the branch prediction architecture. This idea was also explored by Inoue et al [38].

The PSA cache was designed to provide 2-way associativity using a direct mapped structure, but could be expanded to provide higher degrees of associativity. This would require either multiple tag arrays or multiple tag ports – or a set-associative tag component paired with a direct mapped data component. We explore the latter design in the next section.

XI.A.3 Multi-component Serial Cache

Another energy efficient cache design is the serial cache. A serial instruction cache lookup can be broken into two components – the tag comparison and the data lookup. The data component is responsible for the majority of the power consumed in the access. By accessing the less power intensive tag component first, it is possible to eliminate unnecessary power consumption in the bitlines and sense amps of the data component during a cache access. The tag component will indicate what way of the data component needs to be accessed, thereby avoiding unnecessarily driving the bitlines of other ways of the cache and decreasing the number of necessary sense amps. If the data is not present in the cache, the data component access will be avoided, and no unnecessary power dissipation will occur in the bitlines or sense amps of the data component.

The type of serial cache that we examine is the *multi-component serial*

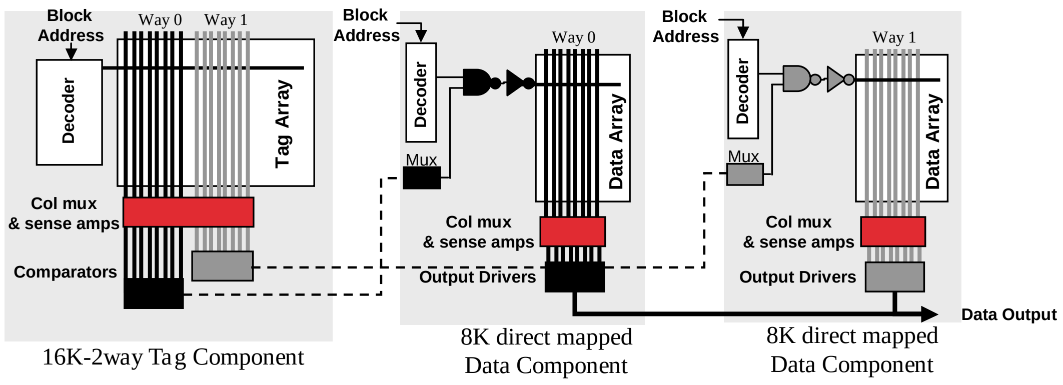


Figure XI.3: A 16KB 2-way set-associative multi-component (MC) serial cache. This has the same tag component as the instruction cache, but with multiple data components. Each data component is a direct mapped cache. For a cache of size C that is A-way set associative, there are A direct mapped caches of size $\frac{C}{A}$ forming the data component.

cache (MC), as shown in Figure XI.3. This cache has the same tag component arrangement as a regular set-associative instruction cache, but rather than a single set-associative data component, there are a number of direct mapped data components. Figure XI.3 shows the arrangement of a 16KB 2-way set associative multi-component serial cache. The tag component chooses one of the two direct mapped data components to drive. The data components collectively comprise the data portion of the MC cache, and so in our 16KB 2-way associative configuration, each data component is only 8KB in size. A 16KB 4-way set associative MC cache would have four 4KB direct mapped data components. Each direct mapped data component has its own decoder, sense amps, and other auxiliary structures. At most one data component is enabled at each access, depending on tag information. To further optimize this design, we have moved the multiplexor drivers from the tag component in Figure XI.1 to the data component in Figure XI.3. These drivers choose which wordlines to enable in the direct mapped data components. The multiplexor drivers can be driven in parallel with the data decoders – the multiplexor drivers determine which data component is enabled and depend on the tag path, while the data decoders only depend on the incoming block address. In addition to the selection logic to determine which data component to activate, there is additional muxing done at the data output of all data components to the output bus of the MC cache. This must be done as several data components will potentially be sharing a common cache output bus.

The PSA cache is very similar to our MC cache. Either design can use some form of prediction to allow parallel access of tag and data components – or can simply use serial access (tag then data) to determine where the data exists. The MC cache is better able to scale to higher associativities however, as it can make use of smaller and faster multiple data components. Also, the tag array on the MC cache is better able to support higher associativities. If the PSA cache is to provide higher associativities it will likely need a set-associative tag component to perform all the tag checks in the initial access.

XI.A.4 Performance and Energy Comparisons

Table XI.1 and Table XI.2 provide results for the different cache configurations we examine. Table XI.1 compares data obtained from our modified version of CACTI 2.0 [74] – the access times (tag and data) and energy consumption for a successful access to the parallel, PSA, and MC caches. Results are shown for 8KB and 16KB caches with 2-way and 4-way set associativity using a single ported data component and a dual ported tag component. We examine cache configurations with dual ported tag components to provide more opportunities for prefetching and speculative fetching. The additional tag port does not significantly impact power or timing of either the MC or parallel cache and can significantly contribute to the the performance of the techniques we examine in the remainder of the paper. In the 16KB 2-way set associative case, the parallel cache (Figure XI.1) has an access time of 0.69 ns, which includes both tag and data component accesses (occurring in parallel). The MC cache (Figure XI.3) is composed of two stages, a tag component taking 0.48 ns followed by a data component taking 0.49 ns. The parallel cache access time is computed by taking the maximum time between the tag and data component, and then adding the delay of the output driver, as in [91]. The tag component of the MC cache does not include the delay of the output driver, and does not include the delay of the multiplexor driver, as both of these structures are no longer on the tag path of the cache (as seen in Figure XI.3). Therefore, the delay of the tag component of the MC cache is less than that of the overall access time to the parallel cache. The data component of the MC cache does include the output driver, but because the MC cache has two direct mapped data components rather than a single set-associative component, the delay of each direct mapped data component is less than that of the overall access time to the parallel cache. However, the overall access time to the parallel cache is less than the sum of the tag and data component access times of the MC cache.

For the rest of this thesis, we only consider the use of the MC cache in the place of the PSA cache, as it provides a slightly faster access time (0.48 ns versus 0.43 ns in the case of a 16K 2-way associative cache) and uses slightly less energy than the PSA cache(1.0 nJ versus 0.9 nJ for the 16K 2-way associative cache).

Table XI.2 compares data obtained from SimpleScalar [12] simulations with the CACTI 2.0 timing and energy data from Table XI.1. We examine two different cycle times in this table. The first set of four columns represents simulation results assuming a cycle time equal to the access time of the parallel cache. In this case, the parallel cache would be accessed in a single cycle, while the MC cache would require two cycles to access. The second set of four columns

Table XI.1: CACTI 2.0 Data

	Parallel		PSA			MC		
Cache Config	Access Time (ns)	Energy per Access (nJ)	Tag Time (ns)	Data Time (ns)	Energy per Access (nJ)	Tag Time (ns)	Data Time (ns)	Energy per Access (nJ)
8KB 2-way	0.65	2.3	0.43	0.43	0.8	0.38	0.41	0.8
8KB 4-way	0.67	4.6				0.38	0.38	0.8
16KB 2-way	0.69	2.5	0.48	0.49	1.0	0.42	0.43	0.9
16KB 4-way	0.71	5.0				0.42	0.41	0.9

CACTI 2.0 timing and energy data for different parallel, PSA, and MC cache configurations. The first column gives the cache configuration (the number of kilobytes and the degree of associativity). The caches examined have a single port on the data component and a dual ported tag component. The next two columns represent the access time and energy dissipation of an access that hits for a parallel cache of the given configuration. The next three columns provide access time to the tag component, access time to the data component, and energy dissipation for the predictive sequential associative (PSA) cache. Note that data is only shown for 2-way PSA configurations as this cache is only designed for 2-way associativity as explained in [16]. The next three columns provide access time to the tag component, access time to the data component, and energy dissipation for the multi-component (MC) cache. Data here is shown for the $0.10\mu m$ process technology size.

represents a cycle time equal to the access time of the data component of the MC cache. Here, both caches require two cycles for a complete access. Each case assumes that other architectural structures in the processor can be pipelined to accommodate the given cycle time. For example, the data cache in the parallel cycle time results takes two cycles for a complete access, while the data cache in the MC cycle time results takes three cycles for a complete access. For each set of cycle time results, both billions of instructions second (BIPS) and total front-end energy dissipation (in Joules) are shown. Results shown are the averages obtained over the seven benchmarks we examined. Chapters VI and V describe in detail the metrics used and the methodology for our experimentation.

For the 16K 2-way set associative case, the MC cache is able to reduce total front-end energy dissipation by 33.2%, with only a 4% decrease in BIPS in the parallel cycle time configuration. The MC cache configuration has a lower BIPS since the instruction cache in this case has an additional pipeline stage. If we match the cycle time of the processor to the data component of the MC cache (the MC cycle time configuration), then both caches take two cycles to access and have identical BIPS. In this case, the MC cache is able to reduce total front-end energy dissipation by 27.4%. The large per access power reduction for a cache hit all comes from the data component, since the tag component for the parallel cache consumes very little power (approximately .1 nJ).

XI.B Single Cycle Instruction Cache Architectures

As discussed in Chapter III, one of the major means of improving processor performance has been to reduce the cycle time of the processor. The instruction cache is a critical component of the front-end architecture, and is integral to maintaining high performance in the front-end. In order to reduce the processor cycle time beyond that of the access time of the instruction cache, it is necessary to

Table XI.2: Cycle Time Data

	Parallel Cycle Time				MC Cycle Time			
	Parallel		MC		Parallel		MC	
Cache Config	BIPS	Energy (J)	BIPS	Energy (J)	BIPS	Energy (J)	BIPS	Energy (J)
16KB 2-way	2.297	0.286	2.210	0.191	3.112	0.263	3.112	0.191
16KB 4-way	2.398	0.447	2.361	0.182	3.325	0.411	3.325	0.182

Simulation results for different parallel and MC cache configurations. The first column gives the cache configuration (the number of kilobytes and the degree of associativity). The caches examined have a single port on the data component and a dual ported tag component. Results are shown for two different processor cycle times. The first is set to the access time of the parallel cache and is shown in columns 2-5. The second is set to the access time of the data component of the MC cache and is shown in columns 6-9. Columns 2,4,6, and 8 show average billion instruction per second results and columns 3,5,7, and 9 show total fetch engine energy dissipated on average for the benchmarks we examined. Data here is shown for the $0.10\mu m$ process technology size.

either reduce the size of the instruction cache or to pipeline the instruction cache access over several cycles. The former will greatly impact the performance of the processor (see Figure X.11), while the latter will lengthen the branch misprediction penalty. The Next Cache Line and Set Prediction (NLS) [15] architecture can reduce the impact of the longer branch misprediction penalty when used with the MC or PSA cache. We examine the NLS architecture, and its limitations, and propose how we can use similar principles with an FTB architecture.

XI.B.1 Next Cache Line and Set Architecture

The NLS architecture [15] features a small, tagless predictor which is used to determine the instruction cache line to be provided in the next cycle and the instruction cache way (they refer to this as the set) to be used in the current cycle. This predictor takes the place of a BTB in a contemporary architecture. Each cycle, the current cache line is used to index into the NLS predictor. The cache line that the predictor returns is the predicted target of the current cache

Table XI.3: FTB Partial Tag Timing Data

Number of Predictor Entries	Access Time (ns)
64	0.44
128	0.45
256	0.47
512	0.48
1024	0.50

CACTI 2.0 timing data for partially tagged FTB configurations. Only 8 bits are allocated for each tag entry in the FTB.

block. The cache way that the predictor returns is a prediction of what way the current cache block is in.

XI.B.2 FTB Architecture

The NLS architecture has three main benefits: a small, tagless predictor which will scale well to future technology sizes; an energy efficient cache design that avoids driving all ways of a particular cache line; and a way prediction mechanism that can reduce the instruction cache access to a single cycle (thereby reducing the branch misprediction penalty). The FTB, two-block ahead predictor [77], and trace cache [75] are all able to make predictions beyond a single cache block. We would like to combine the high bandwidth achievable with these approaches with the benefits of the NLS architecture. In this Chapter, we focus again on the FTB.

In Chapter IX, we demonstrated how the multilevel branch predictor hierarchy allows us to provide scalable and highly accurate branch prediction. We can even reduce the access time to the FTB even further by making use of partial tags. Table XI.3 provides timing data for a variety of FTB structures

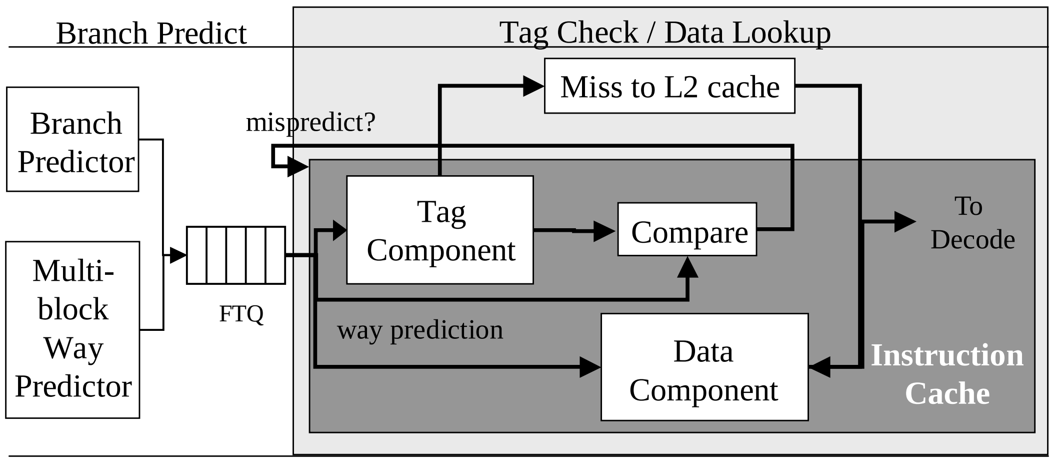


Figure XI.4: MC cache base pipeline with way prediction.
The base pipeline for the MC cache front-end augmented with way prediction. The pipeline consists of two stages: branch prediction and a combined tag check and data lookup stage. The branch predictor (in this case the FTB) feeds the FTQ with fetch predictions. A mutiple way predictor (accessed in parallel with the FTB) augments the fetch predictions with per cache block way predictions. These fetch predictions are then consumed by the instruction cache tag component during the tag check stage. The way prediction is consumed by a data component and the way compare hardware. The data component selected by the way predictor will drive the cache block corresponding to the fetch prediction. The way compare hardware will determine whether or not the way indicated by the tag component matches the way prediction (*i.e.* whether or not the correct data component was enabled). If a misprediction has occured, the correct data component will be accessed in the following cycle. If the way prediction was correct, the instruction cache access will have only taken a single cycle. On a tag miss, the block is brought in from the L2 cache.

which only store 8 bits of the fetch block tag. The use of a partially tagged FTB reduces IPC by less than 1% for all benchmarks examined, but has a significant impact on the access time of the FTB.

As shown in Tables XI.1 and XI.2, the MC cache is able to provide the same energy and timing benefits as the PSA cache. For this Chapter, we will use the MC cache design with our FTB architecture and with the NLS architecture.

XI.B.3 Way Prediction

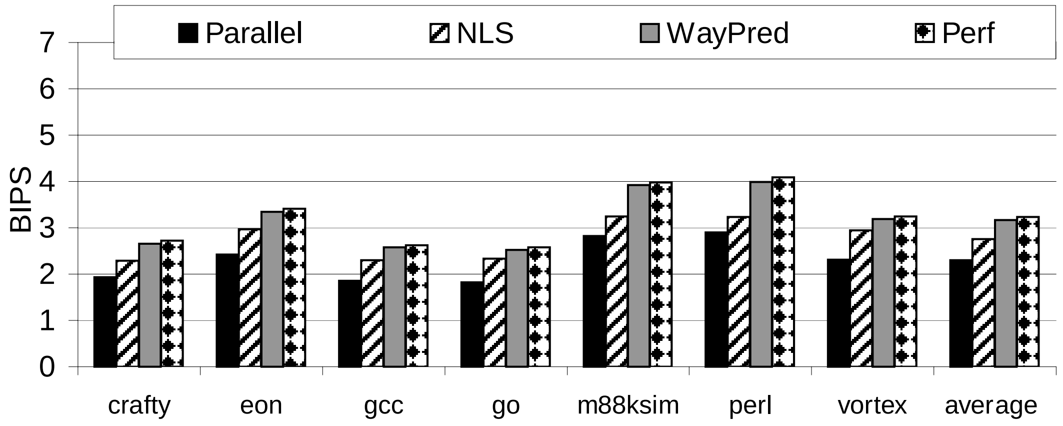
While we have addressed the scalability and energy efficiency of the NLS architecture, we still do not have any form of way prediction that would allow us to collapse the tag check and data lookup stages of the instruction cache access into a single stage – while still using the MC cache. Calder et al. [16] have suggested using a separate way predictor with a data cache to perform such an access in parallel. The predictor we use is a simple last value predictor. However, as each FTB prediction can potentially span up to 5 cache blocks, we would like to provide up to 5 way predictions per cycle. We examine the use of a multiple way predictor to provide this. Our tagless predictor has 8192 entries, and each entry contains 5 2-bit predictors. The way predictor and FTB are accessed in parallel, and the way prediction is stored in the FTQ until it can be consumed by the instruction cache. One alternative to this is to place the way prediction directly in the FTB, trading the additional area required by the way predictor for the access time impact that would result from widening the FTB.

Figure XI.4 demonstrates the addition of such a predictor to our FTB architecture. The tag component grabs the current cache block to be fetched from the FTQ. The data component grabs the way prediction associated with this cache block, which will determine what direct mapped data component to access (*i.e.* what way contains the cache block). The tag component searches all cache ways of the line corresponding to the current cache block. On a correct way prediction, the predicted data component can output its data to the decode stage, and the instruction cache access takes a single cycle. When a way misprediction occurs, a single cycle stall is incurred while the correct data component is accessed (determined via the tag component access from the prior cycle). This configuration has the same branch misprediction penalty as the NLS architecture.

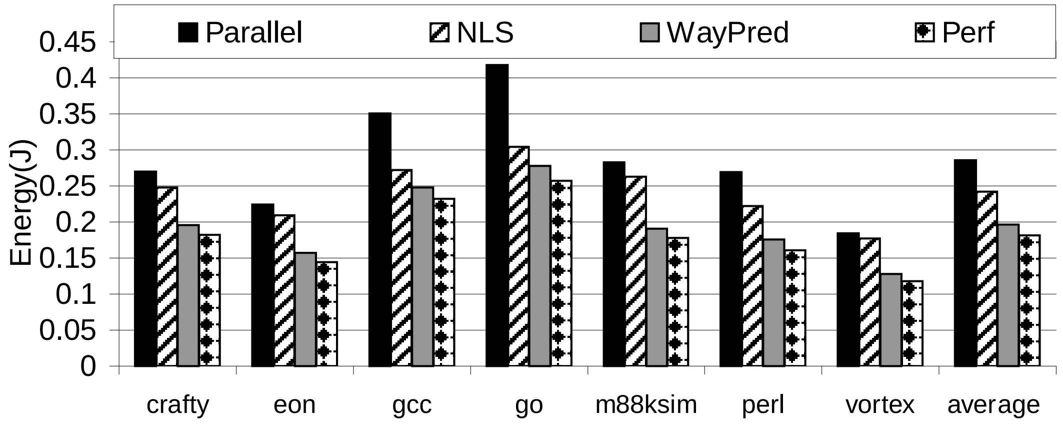
XI.B.4 Results

In this section, we compare performance and energy data of the different cache architectures we have examined so far. The simulation methodology is as described in Chapter VI. We either use a 16KB 2-way set-associative parallel cache with a single read/write port and extra tag port or a 16KB 2-way MC serial cache with single ported data components and dual ported tag components. The L2 bus can provide 8 bytes/cycle. Figures XI.5(a) and (b) show BIPS and energy dissipation results respectively for four architectures. The first bar (Parallel) uses a single cycle parallel instruction cache with the cycle time of the parallel cache. This configuration uses a 2-level FTB with a 128-entry first level. The remaining bars use MC caches. The next bar (NLS) represents a NLS architecture capable of predicting a single block per cycle. The MC instruction cache for this configuration has a single cycle access on a successful NLS way prediction. The NLS predictor can only provide a single cache block per cycle. The third bar (WayPred) represents the MC way prediction configuration from Figure XI.4. This architecture has a single cycle MC instruction cache access on a successful way prediction. Finally, the last bar represents the MC way prediction configuration with a perfect way predictor (Perf). Here, every MC instruction cache access takes a single cycle. These last two bars make use of a 2-level FTB with a 128-entry first level.

The NLS architecture is able to outperform the parallel cache due to the lower cycle time that is achievable through pipelining the instruction cache. However, the larger fetch bandwidth of the FTB allows the WayPred configuration to outperform the NLS configuration (an improvement of around 15% on average). The perfect way predictor provides the best performance for all configurations as it never mispredicts the way of a cache access. Nevertheless, the WayPred architecture is very close to the performance of the perfect predictor.



(a)



(b)

Figure XI.5: BIPS and Energy results for NLS and Way Prediction
This Figure presents BIPS and Energy results for four architectures: a parallel cache with a single cycle instruction cache (Parallel), an NLS architecture that can provide a single cache block each cycle using an MC cache (NLS), an MC cache with a 2-level FTB and a way predictor (WayPred), and an MC cache with a 2-level FTB and a perfect way predictor (Perf). The x-axis shows the 7 benchmarks we examined.

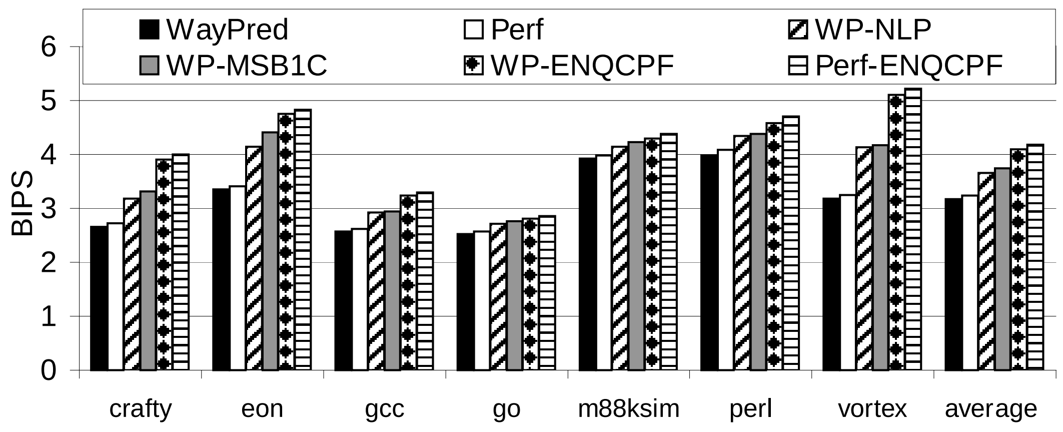
The NLS and Parallel configurations both consume more energy than the WayPred or Perf architectures. The Parallel configuration suffers from a less energy efficient cache design, and the NLS architecture is forced to make more branch predictions to achieve the same fetch bandwidth as the FTB techniques.

XI.C Way Prediction with Prefetching

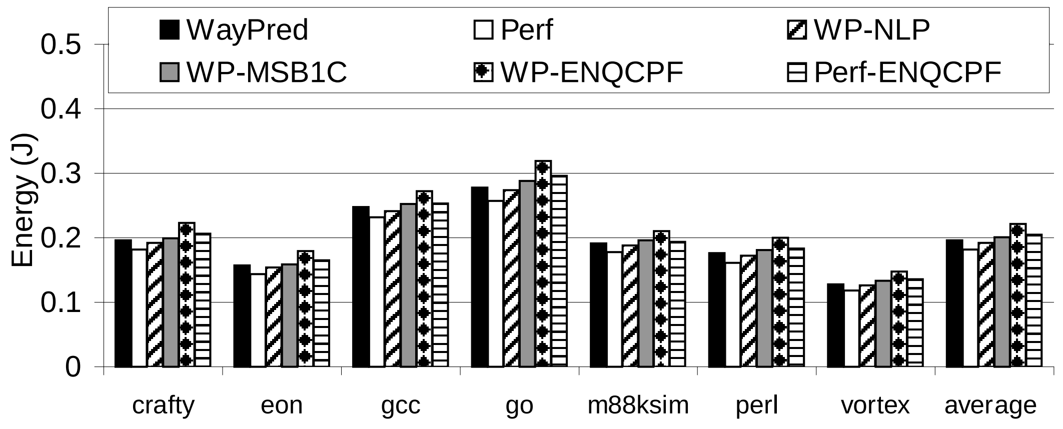
In Chapter X we introduced fetch directed prefetching (FDP) to improve instruction cache performance. In this Section, we enhance our energy efficient architecture with FDP.

The simulation methodology is as described in Chapter VI. We use a 16KB 2-way MC serial cache with single ported data components and dual ported tag components. The L2 bus can provide 8 bytes/cycle. Figures XI.6(a) and (b) provide performance and energy data for different way prediction architectures with prefetching techniques from Chapter X. WayPred is the base configuration, without any form of instruction cache prefetching. WP-NLP uses tagged next-line prefetching and WP-MSB1C uses a single stream buffer with CPF filtering. WP-ENQCPF uses FDP with enqueue CPF filtering. Perf is the base configuration with perfect way prediction. Perf-ENQCPF is the perfect way predictor with FDP and enqueue CPF filtering.

These results agree with Chapter X and demonstrate that enqueue CPF outperforms other prefetching approaches. The way predictor achieves 86% accuracy on average, and provides performance that is very close to perfect predictor accuracy.



(a)



(b)

Figure XI.6: BIPS and Energy results for Way Prediction with FDP
This figure presents BIPS and Energy results for six architectures: an MC cache with a way predictor (WayPred), an MC cache with a perfect way predictor (Perf), an MC cache with a way predictor using NLP (WP-NLP), an MC cache with a way predictor using a single stream buffer with FDP (WP-MSB1C), an MC cache with a way predictor using FDP with enqueue CPF (WP-ENQCPF), and an MC cache with a perfect way predictor using FDP with enqueue CPF (Perf-ENQCPF). The x-axis shows the 7 benchmarks we examined. These results assume an 8 byte/cycle bus to the L2 cache. All architectures use 2 level FTBs. On successful way predictions, these architectures all have single cycle instruction cache pipelines.

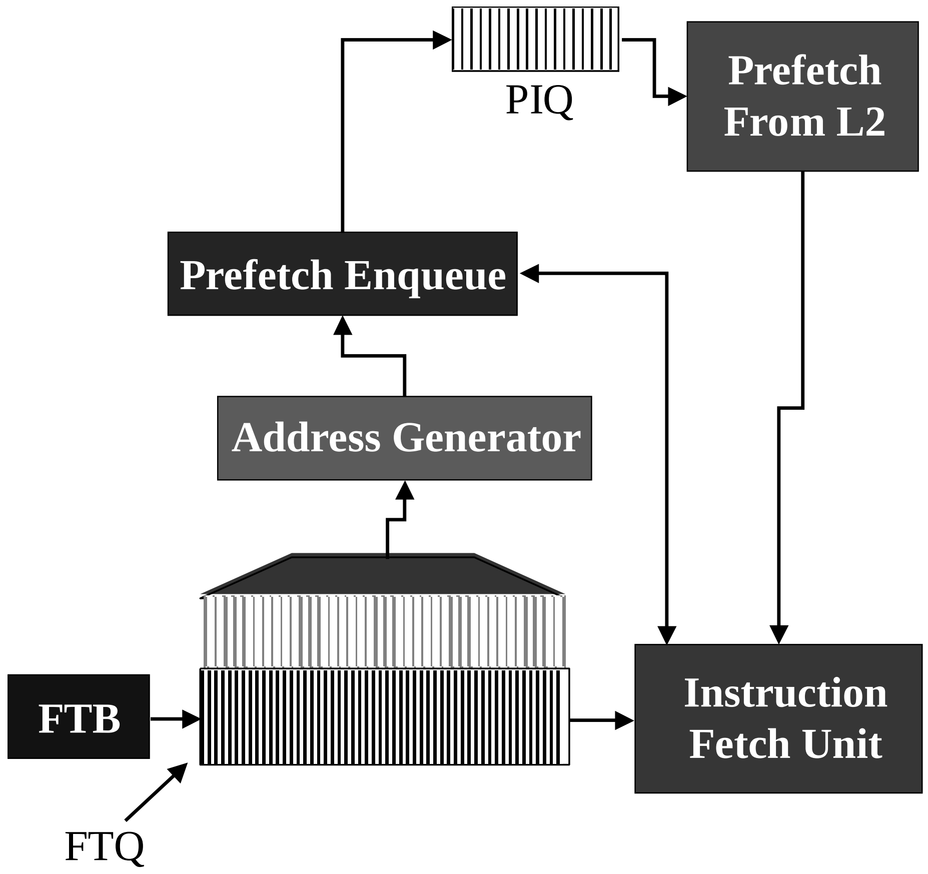


Figure XI.7: FDP complexity concerns.
This Figure is similar to Figure X.3 but expands upon the connection between the FTQ and other structures. Each FTQ entry contains a fetch address and a fetch distance – but can potentially span up to 5 cache blocks. FDP requires some form of address generation to derive up to 5 cache blocks from a single FTQ entry. Here, we assume that the FTQ entry contains 4 additional bits – collectively indicating how many cache blocks the fetch prediction stored in the FTQ entry spans. These bits are then used in the generation of up to five cache block addresses in the address generator. To perform FDP on an arbitrary FTQ entry, there must be a connection from each entry to the address generator via the multiplexor shown above the FTQ. Each grey line leaving the FTQ and connecting to the multiplexor actually represents 31 wires (as a single cache block address to be probed or enqueued is represented by 27 bits – plus the 4 additional cache block indicator bits). So, for a 32 entry FTQ, there would need to be 992 additional wires added to the FTQ to implement FDP from any arbitrary FTQ entry.

XI.D Speculative Fetch Architecture

We have seen how FDP can enhance the performance of the front-end, and how it can be integrated into an energy efficient front-end. However, there are some complexity concerns in this design. Figure XI.7 provides a closer look at the architectural implementation. Here, the branch predictor feeds fetch blocks into the FTQ where they are consumed by the instruction fetch unit (which contains the instruction cache). The fetch block prediction stored in a given FTQ entry may span up to five instruction cache blocks. In order to extract these addresses, the FTQ needs to make use of an address generator. We can utilized extra bits in the FTQ entry to indicate how many cache blocks the entry spans, and can use these in conjunction with the address generator to provide instruction cache block prefetch addresses. With FDP, any entry in the FTQ can potentially initiate a prefetch to the instruction cache. This would require a connection from each FTQ entry to the address generator via a multiplexor. As shown in Figure XI.7, this requires a considerable amount of wire surrounding a small structure. This could have substantial design and performance implications.

Rather than allowing a prediction to proceed from any entry in the FTQ, we could connect a single FTQ entry to both the prefetch enqueue hardware and the instruction cache tag ports. However, this would prove too restrictive, as an entry at the head of the FTQ (near the instruction fetch unit) might not be able to look far enough ahead to provide a timely prefetch. And, an entry at the tail of the FTQ (near the branch predictor) might not be occupied a sufficient fraction of the time to provide maximum benefit.

Additionally, FDP can result in two cache tag checks for a single instance of a cache block: one to verify whether or not the cache block should be prefetched (done in the FTQ) and one as part of the normal cache access (done in the instruction fetch unit).

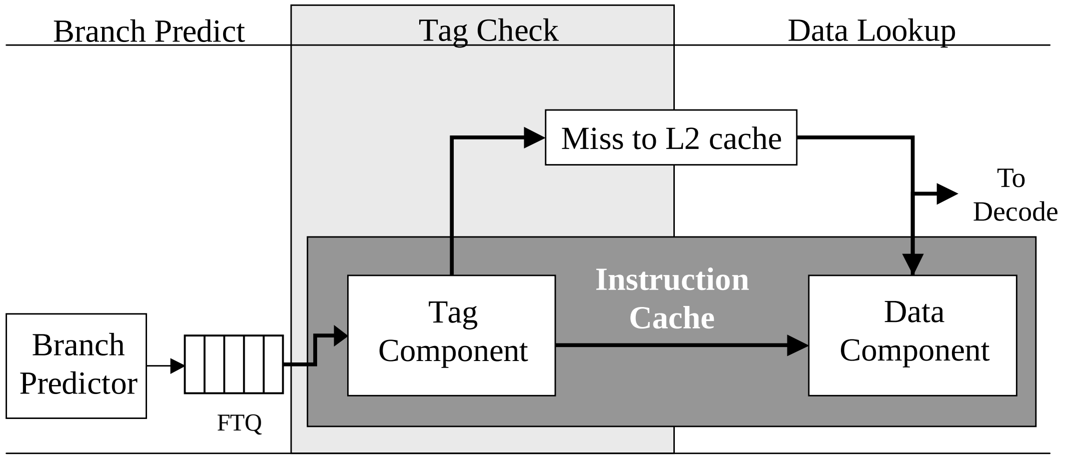


Figure XI.8: MC cache base pipeline.
The base pipeline for the MC cache front-end. The pipeline consists of three stages: branch prediction, tag check, and data lookup. The branch predictor (in this case the FTB) feeds the FTQ with fetch predictions. They are then consumed by the instruction cache tag component during the tag check stage. On a tag hit, the appropriate data component is activated in the data lookup stage. On a tag miss, the block is brought in from the L2 cache.

In order to make this design more power efficient and more complexity effective, we propose an architecture that:

1. performs a tag comparison at most once for a cache block instance
2. integrates prefetching into the normal operation of the instruction cache
3. makes use of an instruction cache with serial lookup to conserve power
4. performs cache block verification and prefetch requests from a single location (rather than from any arbitrary FTQ entry)
5. requires no form of way prediction (and therefore no mechanism for way misprediction recovery), but has a pipelined instruction cache.

Figure XI.8 presents the base configuration of a decoupled branch predictor with an MC cache. This architecture features a front-end that has been split into 3 separate pipeline stages: (1) Branch Prediction, (2) Tag Comparison,

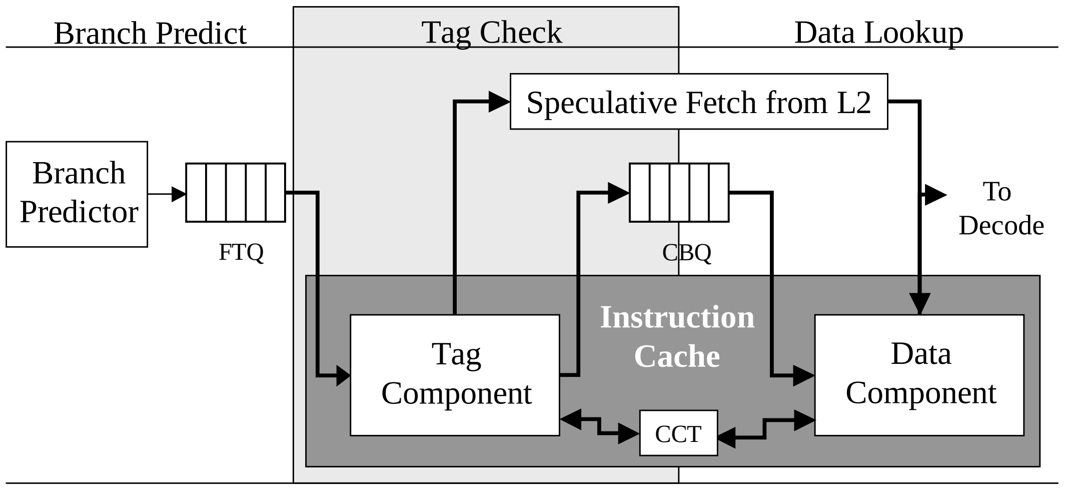


Figure XI.9: The speculative fetch architecture.
The pipeline for the speculative fetch architecture. The pipeline consists of three stages: branch prediction, tag check, and data lookup. The branch predictor (in this case the FTB) feeds the FTQ with fetch predictions. They are then consumed by the instruction cache tag component during the tag check stage. The tag component inserts an entry into the CBQ that contains information about the location of the desired cache block. On a tag hit, the appropriate data component is activated in the data lookup stage. On a tag miss, the block is brought in from the L2 cache. The CBQ decouples the tag component from the data component and allows these structures to operate relatively independently. The tag array is dual ported and can run ahead of the data component. The CCT maintains consistency between the data and tag components – and enables an intelligent replacement algorithm.

and (3) Data Lookup. The branch predictor (in our case, the FTB) supplies the FTQ with fetch block predictions, which are in turn consumed by the tag component of the MC instruction cache during the tag check stage. The tag component then selects an appropriate data component to enable in the data lookup stage. On a tag component miss, the cache block is brought in from the L2 cache.

To achieve the five goals mentioned above, we decouple the tag component from the data component of the instruction cache. Figure XI.9 shows the architecture we examined, the speculative fetch architecture. There are three pipeline stages: branch prediction, tag check, and data lookup, as in Figure XI.8. A non-blocking, MC instruction cache is used, and split into tag and data components separated by a queue of cache block requests called the *cache block queue (CBQ)*. The tag component of the instruction cache gets fetch block addresses from the FTQ and verifies whether or not the cache blocks within the fetch block are already in the instruction cache. If a cache block is not in the instruction cache, the tag array can speculatively fetch the block from the L2 cache. The tag component inserts an entry into the CBQ that corresponds to a single cache block request. The entry contains information about the cache block and instructions requested in the block, as well as what *way* of the cache contains the requested block (or whether it will be brought in from the L2 cache). The data component then consumes this entry and if the block was found to be in the cache (i.e. a known cache hit), it uses the additional state in the CBQ entry to drive the appropriate associative way.

Note that there is no way prediction in this architecture. The tag component is accessed first, and then the corresponding data component is accessed - if there is a tag hit. This avoids the need for any way misprediction recovery hardware.

The FTB can provide up to five cache blocks in a single fetch block, but it is expensive (both in terms of power and access time) to add additional ports to the instruction cache to handle multiple cache blocks. Simply adding extra ports to the tag component of the instruction cache is cheaper than multiporting the data component. This allows the tag component to run ahead of the data component to examine the future cache block request stream of the processor. The tag component can also run ahead of the data component if the data component has stalled due to a full instruction window or it is waiting on data because of a cache miss. We can use the entries stored in the CBQ to guide cache replacement and to speculatively fetch instruction cache blocks that miss in the tag component from lower levels of the memory hierarchy.

However, there is a serious consistency issue which much be addressed. If the tag component is allowed to run ahead of the data component, it is possible that a cache entry can be evicted *after* the tag component has already verified that it is in a particular way of the instruction cache. Because we only perform the tag check once, the data component would grab the wrong instruction cache block. To avoid this, we propose the use of a cache consistency mechanism called the cache consistency table (CCT) that will be explained in Chapter XI.D.3.

Because branch mispredictions do occur, it is not always desirable to bring cache blocks directly into the instruction cache, as they may be on a mis-predicted path and could potentially replace a useful block. Still, as Pierce and Mudge [66] have found, wrong path prefetching can be extremely useful. To resolve this, we make use of a separate fully associative structure to hold cache blocks, the *Speculative Fetch Buffer* (SFB), which holds cache blocks (analogous to the prefetch buffer from Chapter X). The SFB works in parallel with the instruction cache for both tag checks and data lookups. The tag components of the instruction cache and SFB consume an entry from the FTQ and check each cache block in the fetch block to determine whether it is a hit in the instruction cache or SFB — or if it missed in both and must be retrieved from the lower levels of the memory hierarchy. A new entry is inserted in the CBQ for each cache block that is verified from the FTQ. This entry has bits indicating which structure (instruction cache or SFB) contained the cache block that it represents. Then, when the entry is consumed from the CBQ, the bits contained in the entry will indicate which data component should be accessed, and if that component is part of the instruction cache or SFB. Figure XI.10 shows this final modification to the speculative fetch architecture.

If a cache block is not found in the instruction cache or SFB, then it is speculatively fetched from the lower levels of the memory hierarchy and

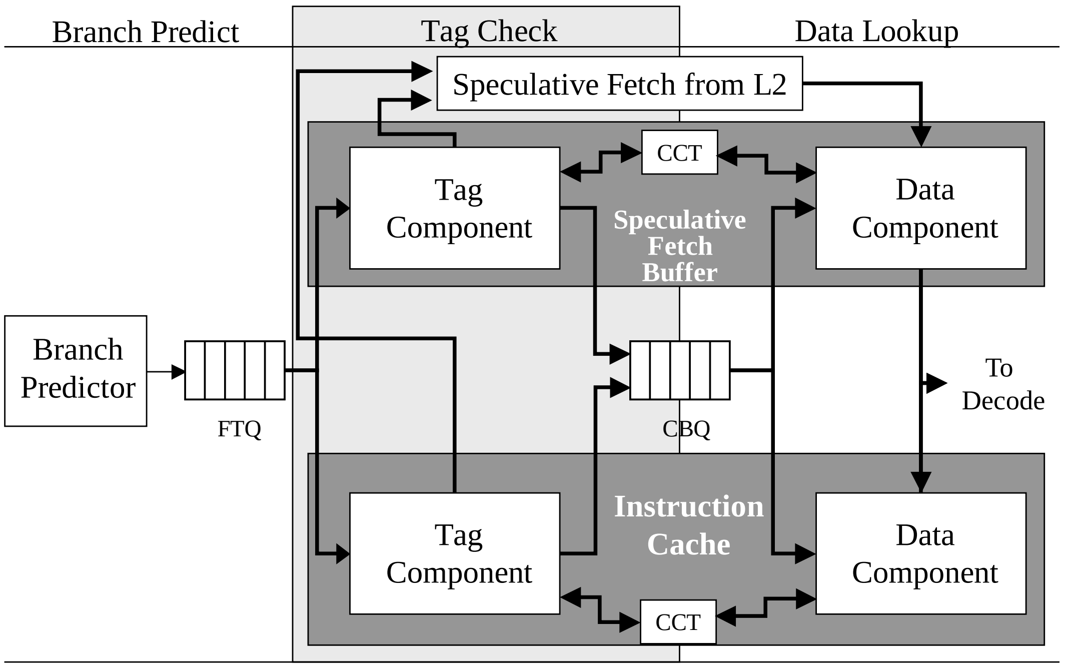


Figure XI.10: The speculative fetch architecture with SFB.
The FTB provides fetch blocks to the FTQ where they are then consumed by the tag components of the speculative fetch buffer and instruction cache. If neither structure has the cache block, the block is speculatively fetched from the lower levels of the memory hierarchy. A new CBQ entry is allocated for each cache block processed by the tag component. It will contain the address of the cache block, the location of the block (or if it missed), and additional way information bits to determine what way of the instruction cache hit (if any). This entry is consumed by the data components of the speculative fetch buffer and instruction cache and used to provide data to the decode stage. At most one data component will actually be accessed. The CCT maintains consistency between the tag and data components and guides cache replacement. Speculative fetches are placed in the speculative fetch buffer until they are used. Then they are placed in the instruction cache depending upon the CCT replacement policy.

brought into the SFB. We will now examine the structures of the speculative fetch architecture in more detail.

XI.D.1 Cache Block Queue

The CBQ holds a cache block address, *block location* bits, *way* bits, and *SFB index* bits. The block location bits represent whether the cache block that the entry represents is in the instruction cache, in the SFB, or is to be brought into the SFB from another level of the memory hierarchy. The way bits are fed into the data component of the instruction cache on an instruction cache hit. These bits indicate which direct mapped component should be activated (i.e. which data way the tag component found the data in – the output from the comparators of the tag array). When the cache block hits in the SFB, the SFB index is used to keep track of the location of the cache block in the SFB. If the cache block is speculatively fetched from lower levels of the memory hierarchy, the SFB index holds the location where the speculatively fetched cache block will be stored.

The size of the CBQ can be used to control the amount of speculative fetching (prefetching) that occurs. The larger the queue, the more the tag comparator can be allowed to run ahead of the data output components, and the more cache blocks (not found in the instruction cache by the tag component) that can potentially be brought into the speculative fetch buffer. The further ahead the tag component runs of the data components, the earlier the speculative fetch can occur and the more memory latency that can be hidden. However, there is also a greater chance of speculating down a mispredicted control path. Therefore the size of the CBQ trades the benefit obtainable from speculative fetching with the amount of power potentially wasted on mispredicted control paths (similar to the tradeoffs inherent in FTQ size). The CBQ is flushed on a branch misprediction.

XI.D.2 Speculative Fetch Buffer and Cache Misses

The SFB is a small 32 entry buffer, which holds cache blocks that have been speculatively fetched from lower levels of memory. As with the instruction cache, the tag component on this buffer is probed when a fetch block is consumed from the FTQ in the tag check stage. The SFB data component is only accessed on a known tag hit, like the MC cache, to save power.

If a cache block is not found in either the instruction cache or SFB tag components, a SFB tag entry is allocated using the consistency mechanism described below. If a tag entry in the SFB is allocated, a speculative fetch is initiated to the lower levels of the memory hierarchy. If an SFB tag entry cannot be allocated, then the tag lookup pipeline stage stalls until an entry in the SFB can allocated.

If a cache reference is not found in the instruction cache tag component, the CBQ entry that is to be created will be marked as a cache miss, and it will specify the entry in the SFB where the cache block is to be found. When the CBQ entry is consumed by the data lookup, the SFB cache block is used and potentially brought into the instruction cache depending upon the cache consistency table, which is described in more detail below.

This approach does not allocate blocks into the instruction cache until they are used by the data component. This reduces cache pollution caused by branch mispredictions. We found this to perform better than allocating the cache block during the tag component pipeline stage when the initial miss occurs.

XI.D.3 Consistency Mechanism

Because the tag component can verify cache blocks far in advance of the data component and we perform replacement in the data lookup pipeline stage, we need some consistency mechanism to guarantee that cache blocks verified by the tag component are not evicted during cache replacements before they can be accessed in the data lookup stage. We maintain an extra table, called the *cache consistency table (CCT)*, to guarantee this. The CCT is a tagless buffer, with one entry for every cache block in the instruction cache, but with much smaller blocks – each CCT block only holds an N-bit counter. For example, assuming the use of a 3-bit counter, a 16KB 2-way associative instruction cache would only need a 320 byte table (a structure roughly 1% of the size of the instruction cache).

The counter stored in each CCT entry represents the *number of outstanding verified cache block requests* sitting in the CBQ for the corresponding entry in the instruction cache. When the tag component verifies a cache block in the tag check stage, the N-bit counter in the CCT corresponding to that cache block is incremented. When a data component accesses an instruction cache block, the N-bit counter in the CCT corresponding to that cache block is decremented. On a misprediction or misfetch, the CCT is flushed (set to zero), just as the CBQ is also flushed. The mapping from instruction cache to CCT is implicit and does not require tags – since both structures have the same number of entries and associativity, they have identical decoders.

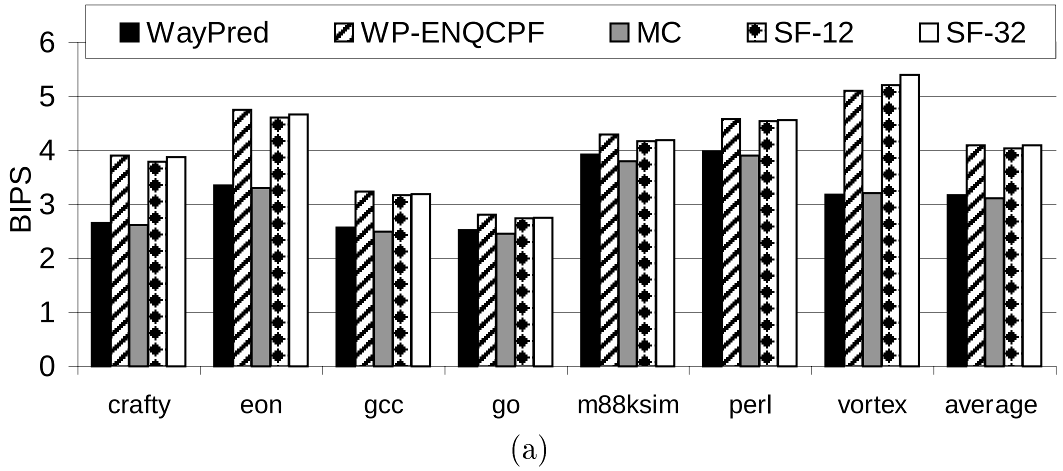
This consistency mechanism also extends to the speculative fetch buffer. The SFB has its own dedicated CCT as shown in Figure XI.10. If the desired cache block is not found in the instruction cache, but is in the speculative fetch buffer, a N-bit counter associated with this structure is incremented each time that block is placed in the CBQ. This is necessary to preserve cache consistency so that a cache block in the SFB that a CBQ entry points to is not replaced.

The CBQ provides a means to look ahead at the behavior of the instruction cache. As mentioned before, this enables the front-end to save power by only accessing structures that are already known to have the desired cache block and to guide the speculative fetch of cache blocks that do not hit in any first level structure. In addition, the CBQ can also help guide instruction cache or SFB replacement, with the help of the CCT. When a cache block is brought into the instruction cache from the SFB, a block is chosen for replacement from the set that has a zero CCT entry. This ensures that a cache block that a later CBQ entry wants to use does not get removed from the cache. This policy overrides the standard LRU replacement policy of the instruction cache. If all cache blocks in a particular cache set are marked in the CCT (meaning no replacement for the new cache block exists), then the block is not put into the instruction cache, and instead just stays in the speculative fetch buffer until used again or replaced. Similarly, the replacement policy ensures that entries in the SFB with non-zero N-bit counters are not replaced by new speculative fetches until they are used by consuming a CBQ entry. In this manner, the speculative fetch buffer acts as a flexible depository of instruction cache blocks for contended cache sets – directed by the CBQ and CCT.

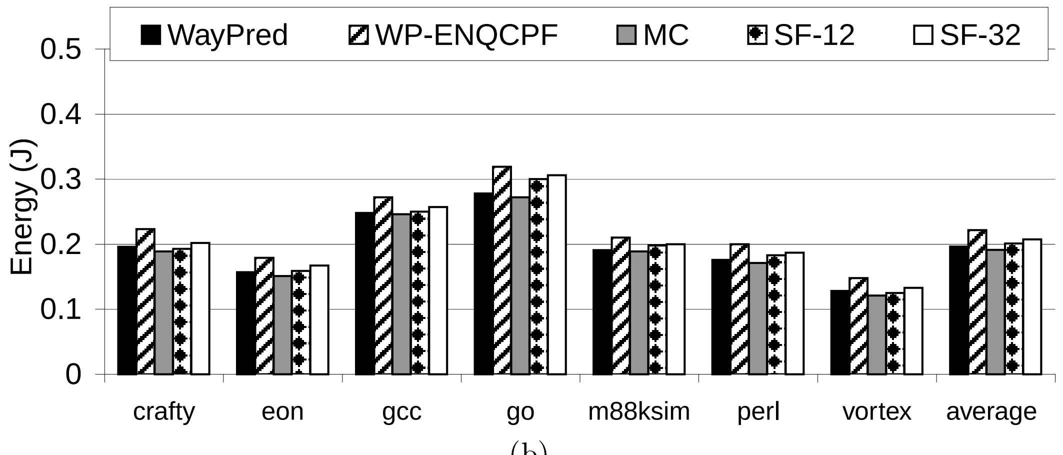
If the N-bit counter associated with a cache block in any structure is saturated, and another instance of the saturated cache block is encountered, the tag component stalls until the counter is decremented (the data output component consumes an instance of the cache block) or until a branch misprediction occurs. In this paper, we use 5-bit CCT counters when using a 32 entry CBQ, and 3-bit CCT counters when using a 12 entry CBQ.

XI.D.4 Results

Figures XI.11(a) and (b) present BIPS and Energy results for five architectures. The first two bars represent the MC cache with way prediction – with and without FDP using enqueue CPF (WayPred and WP-ENQCPF). (MC) represents the MC cache without either way prediction or prefetching. The final two bars represent the speculative fetch architecture with a 12-entry CBQ (SF-12)



(a)



(b)

Figure XI.11: BIPS and Energy results for speculative fetch.
This figure presents BIPS and Energy results for five architectures: an MC cache with a way predictor (WayPred), an MC cache with a way predictor using FDP with enqueue CPF (WP-ENQCPF), an MC cache without a way predictor and without prefetching (MC), the speculative fetch architecture with a 12-entry CBQ (SF-12), and the speculative fetch architecture with a 32-entry CBQ (SF-32). The x-axis shows the 7 benchmarks we examined. These results assume an 8 byte/cycle bus to the L2 cache. All architectures use 2 level FTBs.

and a 32-entry CBQ (SF-32).

For most benchmarks, the SF-32 configuration performs slightly worse than the WP-ENQCPF architecture. The benchmark `vortex` is the exception however, as SF-32 outperforms WP-ENQCPF for this program. WP-ENQCPF shortens the branch misprediction penalty by a single cycle, but does not have the same intelligent replacement algorithm that SF-32 has. Therefore, programs with a significant number of instruction cache conflict misses will see more benefit from the SF-32 architecture – and programs with substantial branch mispredictions can see more benefit from the WP-ENQCPF architecture. It depends on what the particular benchmark is more constrained by: branch misprediction or instruction cache misses. Additionally, the way predictor performs relatively worse when considering caches with higher degrees of associativity. This can be further illustrated by Figures XI.12 and XI.13. These show the performance of SF-32, WP-ENQCPF, and Perf-ENQCPF (perfect way prediction with enqueue FDP) for an 8KB 2-way MC cache and an 8KB 4-way MC cache respectively. Here, `vortex` sees so much benefit from the intelligent replacement algorithm that it is able to outperform even the perfect way prediction configuration. The benchmark `perl` sees more improvement from the SF-32 architecture than the WP-ENQCPF architecture under the increased cache space constraint. Most other benchmarks obtain about the same amount of performance from both techniques, although `go` still sees slightly more benefit from WP-ENQCPF due to its relatively poor branch prediction.

Figure XI.14 demonstrates the ability of the speculative fetch buffer to provide extra associativity to a cache design. Here, we consider an 8KB direct-mapped cache configuration. This cache is plagued with a considerable amount of conflict misses. We show only two architectures in this Figure, the speculative fetch architecture (SF-32) and way prediction using FDP with enqueue CPF

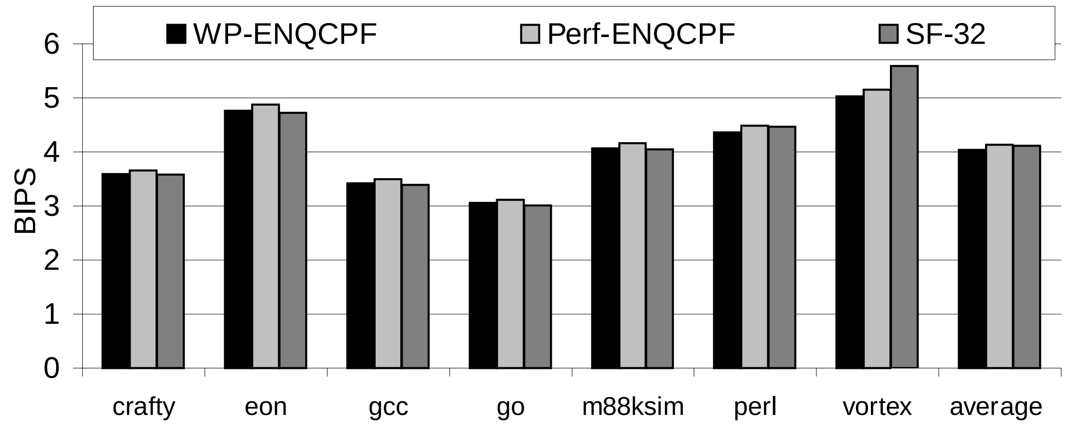


Figure XI.12: 8KB 2-way set-associative cache results for speculative fetch. This figure presents BIPS results for three architectures: an MC cache with a way predictor using FDP with enqueue CPF (WP-ENQCPF), an MC cache with a perfect way predictor using FDP with enqueue CPF (Perf-ENQCPF), and the speculative fetch architecture with a 32-entry CBQ (SF-32). The x-axis shows the 7 benchmarks we examined. These results assume an 8 byte/cycle bus to the L2 cache. All architectures use 2 level FTBs.

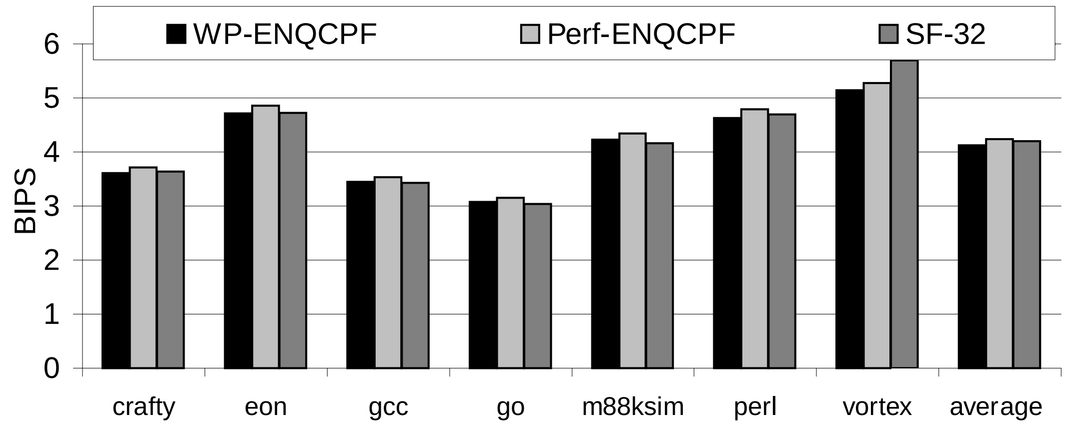


Figure XI.13: 8KB 4-way set associative cache results for speculative fetch. This figure presents BIPS results for three architectures: an MC cache with a way predictor using FDP with enqueue CPF (WP-ENQCPF), an MC cache with a perfect way predictor using FDP with enqueue CPF (Perf-ENQCPF), and the speculative fetch architecture with a 32-entry CBQ (SF-32). The x-axis shows the 7 benchmarks we examined. These results assume an 8 byte/cycle bus to the L2 cache. All architectures use 2 level FTBs.

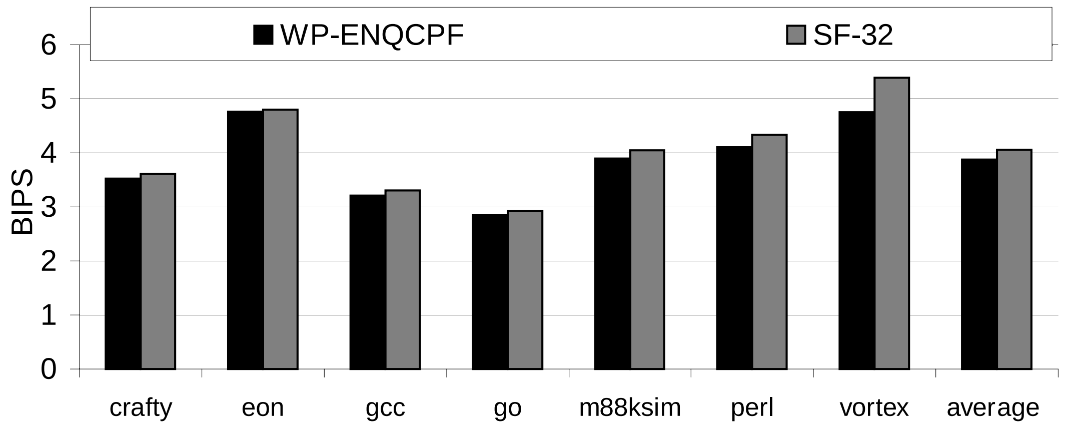


Figure XI.14: 8KB direct-mapped cache results for speculative fetch. This figure presents BIPS results for two architectures: an MC cache with a way predictor using FDP with enqueue CPF (WP-ENQCPF) and the speculative fetch architecture with a 32-entry CBQ (SF-32). The x-axis shows the 7 benchmarks we examined. These results assume an 8 byte/cycle bus to the L2 cache. Both architectures use 2 level FTBs.

(WP-ENQCPF). There is no need to show perfect way prediction as this is a direct mapped configuration and there is only a single way to search (*i.e.* WP-ENQCPF is the same as perfect way prediction in this instance). Here, the SF-32 case is able to outperform WP-ENQCPF for all benchmarks – even though the instruction cache in the SF-32 case still takes two cycles to access. This can be fully attributed to the benefit obtainable from the intelligent replacement policy.

Figure XI.11(b) shows that all architectures examined dissipate relatively small amounts of energy. The speculative fetch architecture is able to reduce energy consumption somewhat more than the WP-ENQCPF architecture due to the more intelligent replacement policy and due to the absence of the way predictor. If cache blocks can be kept around longer when they will be needed in the near future, we can avoid energy intensive L2 accesses.

The difference between the SF-12 and SF-32 is relatively small for most benchmarks, with the exception of `vortex`.

XI.E Summary

In this Chapter, we have presented a complexity-effective and energy-efficient approach to instruction cache prefetching, speculative fetch. The speculative fetch architecture combines an energy efficient cache design with enqueue cache probe filtered fetch directed prefetching and an intelligent cache replacement algorithm that can look ahead at the cache block request stream. We have compared this technique to a similar FDP scheme that makes use of a way predictor. The use of the way predictor reduces the branch misprediction penalty, but adds complexity to the overall design as there must be some way misprediction recovery mechanism. The speculative fetch architecture performs a serial cache lookup, and requires no way prediction, but is able to provide the same energy savings and the same average performance. This design is flexible enough to accommodate more storage structures, such as a victim cache, which could also be decoupled and augmented with a CCT.

Chapter XII

Conclusions

In this thesis, we have investigated a decoupled front-end architecture that uses a fetch target queue (FTQ), which is a small FIFO queue used to store predicted fetch addresses from the branch prediction unit. The FTQ provides a decoupled front-end that allows the branch predictor and instruction cache to run more independently. Decoupling the components of the front-end architecture enables a number of optimizations, and we demonstrated three of these in this study: the construction of a multilevel branch predictor, effective instruction cache prefetching, and an energy-efficient cache replacement policy.

To better evaluate our architecture, we made use of results in both IPC and BIPS (billion instructions per nanosecond). While IPC provides some notion of performance, BIPS gives a complete picture as it takes into account the cycle time of the processor. It is essential for architects to study both the improvement in performance obtained through a particular technique and the impact the technique (in terms of the size of structures used) will have on the cycle time of the processor. To obtain cycle time information, we made use of the CACTI 2.0 [74] timing model. This also provided us with energy data for Chapter XI.

The FTQ enables a number of optimizations based on the predicted fetch stream it contains. We first showed that a multilevel branch predictor can be an effective technique to achieve high prediction accuracy while retaining small cycle times. In our results, we saw that the best performing two level branch predictor (a 128 entry first level fetch target buffer with an 8192 entry second level) achieved a 5% speedup on average in BIPS over the best performing single level FTB configuration (a 512 entry FTB). These results assume no interconnect scaling bottleneck. In the event of poor interconnect scaling, the gap in cycle times between the single level tables and the multilevel tables can be significant: the best two-level we examined configuration provides a 14% speedup on average over the best single level configuration we examined.

We also demonstrated how the FTQ can be used to direct instruction cache prefetching. With this technique, we were able to provide a speedup in IPC of 17% for an 8 byte/cycle bus to the L2 cache. We showed how cache probe filtering could be an effective technique for reducing bus utilization and attaining higher levels of performance. Our best performing filtration technique, enqueue cache probe filtering, provided an average 30% speedup over a base configuration without prefetching for the 8 byte/cycle bus configuration. However, the performance of fetch-directed prefetching is essentially tied to the accuracy of the prediction stream and the occupancy of the FTQ.

Finally, we explored a number of complexity and energy dissipation issues in the design of the instruction cache. We proposed the MC cache design, a technique that showed a 27% reduction in energy dissipation from a contemporary cache design. We then examined the addition of way prediction to our architecture in order to reduce the branch misprediction penalty by enabling a single cycle instruction cache access (at a reduced cycle time). This design provided a 10% improvement over the use of an NLS architecture, thanks to the FTB. Finally, we explored the speculative fetch architecture. This architecture has the potential to provide even higher levels of BIPS performance for benchmarks which are heavily constrained by instruction cache conflict misses. This is accomplished through the use of an intelligent replacement policy which can reduce instruction cache block thrashing. Moreover, this technique reduced the complexity of fetch directed prefetching by consolidating the prefetch verification and enqueuing mechanism to a single site (rather than any arbitrary FTQ location) and by integrating prefetching into the normal operation of the instruction cache. For a space-constrained instruction cache (an 8KB direct-mapped cache), the speculative fetch architecture provides an additional 5% improvement over the FDP architecture with way prediction. The 16KB 2-way set associative instruction cache that we examined throughout this thesis sees around the same performance on average for both of these techniques.

These techniques prove useful when examining ideal interconnect scaling. However, the potential benefit is even greater in the face of the interconnect scaling bottleneck. In order to reach future processor performance goals, Agarwal et al. [2] claim that architects may no longer be able to both improve IPC and increase the clock speed of the processor. To increase the clock speed, it may be necessary to decrease the size of the structures that are used in the processor pipeline, which can impact performance. However, the use of multilevel hierarchies and intelligent prefetching strategies can allow us to use smaller structures in the pipeline – which in turn allows us to increase the clock speed. And as we have shown, these designs need not add complexity or increase energy dissipation in the processor.

The design of decoupled components in the processor pipeline provides us with two advantages: latency tolerance and processor look-ahead. Components that are decoupled from one another can operate relatively independently and can more easily be changed to a multilevel hierarchy. In this thesis, we have shown how decoupling the branch predictor from the instruction cache enables a multilevel branch predictor hierarchy. This helps to simplify the core pipeline by enabling a smaller first level branch predictor. Because decoupling allows components to run ahead of one another, we can also get insight into the future resource needs of the processor. In this thesis we examined using the stream of fetch addresses contained in the FTQ to guide instruction cache prefetching. This can allow us to use smaller first level instruction caches, which can again simplify the core pipeline. Such techniques can aid in keeping large memory structures and complex logic off of the critical path of the processor core, and will greatly aid in scaling these processor to future technology sizes.

Chapter XIII

Future Work

There are a number of other optimizations which could be enabled by the FTQ design. In addition to instruction cache prefetching, the FTQ could be used to direct data cache prefetch in a manner similar to that proposed in [20]. The FTQ could also be used to index into other PC-based predictors (such as value or address predictors) further ahead in the pipeline.

XIII.A Branch Prediction

The use of a multi-level branch predictor, like the FTB, provides us with a significant amount of instruction state. Extra bits could be stored in the branch predictor to help guide later stages of the pipeline. For example, we used extra state in the multi-level branch predictor to perform eviction prefetching in Chapter X. The state can be used to help guide cache replacement policies, and to track branch confidence, value predictability, or any other useful metric.

Other future work on the FTB will involve the use of a more intelligent replacement mechanism for the first level FTB. By storing different types of branch confidence and usefulness measures in the FTB, we will be able to keep the most important fetch blocks in the first level FTB at all times. Fetch blocks that are not fetch critical (i.e. they occur when the FTQ already has considerable occupancy, occur infrequently, occur during a cache miss, etc) will be kept in the second level FTB and will not replace a more critical fetch block in the first level FTB. This technique should help benchmarks like `m88ksim` which do not have good average FTQ occupancy and cannot tolerate second level FTB accesses for critical fetch blocks.

Finally, it would prove interesting to use other branch predictor architectures in place of the FTB on our decoupled front-end. It may also be useful to combine these predictors together in the front-end. While a trace cache [75] is able to provide high bandwidth fetching, it is a large structure that may not scale well to future technology sizes. The trace cache may need to be kept small, and only used to store the most important instructions in a program (possible those used in a frequently executed loop). These important instructions can then be dynamically optimized by the trace cache. The FTB can be used to handle the remainder of the program.

XIII.B Value Prediction

Value and address prediction have been shown to be effective at reducing instruction latency in the processor pipeline [51, 52, 29, 31, 76, 89, 71]. A value predictor attempts to predict the result of or the inputs to a particular instruction. If the result of an operation is predicted, instructions that are currently waiting on the completion of the operation may speculatively execute. If the inputs to a particular instruction are value predicted, the instruction itself can speculatively execute. An address predictor attempts to predict the memory location that a load or store will access. The load or store will speculatively execute with the predicted address.

The predictor tables used can be quite complex and fairly large. The fetch addresses stored in the FTQ can initiate the predictor access earlier in the pipeline, even before the corresponding instruction cache blocks are fetched, storing the result with the fetch block in the FTQ. This can allow for larger and more accurate predictors to be used, and even allow these predictors to be located off-chip.

One example of this is context prediction [76]. In a context value predictor that predicts the results of an operation, the last n values of an instruction are stored in hashed form in a first level table that is hashed by instruction PC. These values are used to index into a second level table that contains the actual value to be predicted. In [76], a first level table with 2^{16} entries and a second level table with 2^{20} entries were used. Assuming the predictor stores 32-bit values in the second level table, the second level will be at least 32 MB in size, and will likely require multiple cycles to access alone (not including the number of cycles necessary to access the first level table to generate the index to the second level table). But with the fetch addresses stored in the FTQ, we could conceivably initiate the predictor access earlier in the pipeline - storing the result in the FTQ itself if the prediction has not yet been consumed. This could allow even larger and more accurate predictors to be used, potentially even allowing these predictors to be located off-chip.

Bibliography

[1] A. Agarwal and S. Pudar. Column-associative caches: A technique for reducing the miss rate of direct mapped caches. In *20th Annual International Symposium on Computer Architecture*, 1993.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th Annual International Symposium on Computer Architecture*, 2000.

[3] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32st International Symposium on Microarchitecture*, November 1999.

[4] T. Austin, B. Calder, and G. Reinman. Modified cacti model source and documentation distribution. http://www-cse.ucsd.edu/users/calder/bacti, October 1998.

[5] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronics and Design*, 1998.

[6] H. Bakoglu and J. Meindl. Optimal interconnect circuits for VLSI. *IEEE Transactions on Computers*, 32(5):903–909, May 1985.

[7] N. Bellas, I. Hajj, and C. Polychronopoulos. A new scheme for i-cache energy reduction in high-performance processors. In *International Symposium on Low Power Electronics and Design*, 1998.

[8] M. Bohr. Interconnect scaling - the real limiter to high-performance ulsi. In *Tech. Dig. of the International Electron Devices Meeting*, pages 241–244, December 1995.

[9] M. Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[10] J. O. Bondi, A. K. Nanda, and S. Dutta. Integrating a misprediction recovery cache (MRC) into a superscalar pipeline. In *Proceedings of the 29th Annual*

International Symposium on Microarchitecture, pages 14–23, December 2–4, 1996.

[11] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, 2000.

[12] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

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

[14] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, October 1994.

[15] B. Calder and D. Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[16] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *International Symposium on High Performance Computer Architecture*, 1996.

[17] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[18] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 274–283, June 1997.

[19] I.K. Chen, C.C. Lee, and T.N. Mudge. Instruction prefetching using branch prediction information. In *International Conference on Computer Design*, pages 593–601, October 1997.

[20] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[21] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 51–61, October 1992.

[22] Y. Chou and J. Shen. Instruction path coprocessors. In *27th Annual International Symposium on Computer Architecture*, 2000.

[23] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[24] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[25] J. Montanaro et al. A 160 mhz 32b 0.5w cmos risc microprocessor. In *Digital Technical Journal*, August 1997.

[26] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[27] K.I. Farkas, N.P. Jouppi, and P. Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, January 1995.

[28] J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.

[29] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[30] K. Ghose and M. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *Proceedings of the International Symposium on Low Power Design*, 1999.

[31] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[32] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. In *IEEE Journal of Solid State Circuits*, 1996.

[33] C. Thomas Gray, Wentai Liu, and Ralph K. Cain III. *Wave Pipelining: Theory and CMOS Implementation.* Kluwer Academic Publishers, Norwell, MA, 1993.

[34] L. Gwennap. Power issues may limit future cpus. Microprocessor Report, August 1996.

[35] E. Hao, P. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 191–200, December 1996.

[36] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1996.

[37] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1999.

[38] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *International Symposium on Low Power Electronics and Design (ISLPED'99)*, 1999.

[39] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[40] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[41] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, April 1994.

[42] S. Jourdan, T. Hsing, J. Stark, and Y. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[43] M. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. *International Symposium on Low Power Electronics and Design*, 1997.

[44] R. Kessler. The alpha 21264 microprocessor. In *IEEE Micro*, April 1999.

[45] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache:an energy efficient memory structure. In *IEEE International Symposium on Microarchitecture*, December 1997.

[46] U. Ko, P. Balsara, and A. Nanda. Energy optimization of multi-level processor cache architectures. In *Proceedings of the International Symposium on Low Power Design*, 1995.

[47] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium of Computer Architecture*, pages 81–87, May 1981.

[48] D. Lammers. IBM's copper interconnects hit the market. EETimes, 9/3 issue, September 1998.

[49] D. Lammers. TI's 0.13-micron process speeds system-on-a-chip designs. EE-Times, 10/23 issue, October 1998.

[50] S. Lee, Y. Wang, and P. Yew. Decoupled value prediction on trace processors. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[51] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, October 1996.

[52] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[53] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *31st International Symposium on Microarchitecture*, December 1998.

[54] J. McCormack, R. McNamara, C. Gianos, L. Seiler, N. Jouppi, and K. Correll. Neon: a single-chip 3d workstation graphics accelerator. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics Hardware*, 1999.

[55] G. McFarland and M. Flynn. Limits of scaling mosfets. CSL TR-95-62, Stanford University, November 1995.

[56] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[57] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[58] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[59] S. Oh, K. Rahmat, O. Nakagawa, and J. Moll. A scaling scheme and optimization methodology for deep sub-micron interconnect. In *IEEE International Conference on Computer Design*, pages 320–325, October 1996.

[60] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[61] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[62] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[63] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. CSE-TR-335-97, University of Michigan, May 1997.

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

[65] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[66] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *29th International Symposium on Microarchitecture*, pages 165–175, December 1996.

[67] Fred Pollack. New microarchitectural challenges in the coming generations of cmos process technologies. Slides from Fred Pollack's Micro32 keynote speech, November 1999.

[68] T.R. Puzak. Analysis of cache replacement-algorithms. Ph.D. Dissertation, University of Massachusetts, Amherst MA, 1985.

[69] J. Rabaey. *Digital Integrated Circuits.* Prentice Hall Electronics and VLSI Series., 1996.

[70] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[71] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, December 1998.

[72] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *32st International Symposium on Microarchitecture*, November 1999.

[73] G. Reinman, B. Calder, and T. Austin. Optimizations enabled by a decoupled front-end architecture. In *IEEE Transactions on Computers*, 2001.

[74] G. Reinman and N. Jouppi. Cacti version 2.0. http://www.research.digital.com/wrl/people/jouppi/CACTI.html, June 1999.

[75] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.

[76] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[77] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, October 1996.

[78] T. Sherwood and B. Calder. The time varying behavior of programs. Technical Report UCSD-CS99-630, University of California, San Diego, August 1999.

[79] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 259–271, December 1998.

[80] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. Technical Report TR-589-98, Princeton Dept. of Computer Science, December 1998.

[81] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[82] J. Smith. Instruction-level distributed processing. In *IEEE Computer*, April 2001.

[83] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[84] J. Stark, P. Racunas, and Y. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 34–45, December 1997.

[85] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *Proceedings of the International Symposium on Low Power Design*, 1995.

[86] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, January 1995.

[87] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 345–356, June 1995.

[88] S. Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 94–103, 1997.

[89] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[90] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison Wesley, 1993.

[91] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Compaq WRL TR-93-5, July 1994.

[92] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, May 1996.

[93] C. Xia and J. Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *23rd Annual International Symposium on Computer Architecture*, June 1996.

[94] T. Yeh. Two-level adpative branch prediction and instruction fetch mechanisms for high performance superscalar processors. Ph.D. Dissertation, University of Michigan, 1993.

[95] T. Yeh and Y. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, December 1992.